**IMDB Clone API**

**Imp Notes**

Pk = self.kwargs[‘pk’]

Review.objects.filter(review\_user=pk)

**And here review\_user is known as Foreign key**

**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.

**Very IMP**

**Generic (Concrete Based Class View)**

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

<https://www.django-rest-framework.org/api-guide/generic-views/>

* Generic class have all the power of mixings and APIView. Bcz
* Generic inherited the mixings
* Mixings implemented all the get post put methods.
* Mixings inherited the APIView class.

**Views.py**

**Import the generic**

from rest\_framework import generics

class ReviewList(generics.ListCreateAPIView):

    queryset = Review.objects.all()

    serializer\_class = ReviewSerializer

class ReviewDetail(generics.RetrieveUpdateDestroyAPIView):

    queryset = Review.objects.all()

    serializer\_class = ReviewSerializer

Now I want all the reviews of a particular movie only

Like this url movie/all\_Reviews

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

So I have to override the Queryset method and perform specific task.

class ReviewList(generics.ListCreateAPIView):

    # queryset = Review.objects.all()

    serializer\_class = ReviewSerializer

    def get\_queryset(self):

        pk = self.kwargs['pk']

        return Review.objects.filter(watchlist=pk)

Now if I visit <http://127.0.0.1:8000/watch/3/reviews/> this link for movie 3 I will get all the reviews of movie 3.

Now I want to if I want to give review of any movie then I don’t want to pass in post request.

It should select automatically.

path('<int:pk>/review-create', ReviewCreate.as\_view(), name="reviewCreate"),

**I want this kind of url where I can create review for any particular movie.**

For more reference you can see the E-commerce website. Reviews

Views.py

* **First remove the create in ReviewList and use only ListAPIView**
* **Use the perform\_create method**
* **We have to exclude watchlist in reviewSerializer bxz we don’t want to show to user**.

class ReviewCreate(generics.CreateAPIView):

    serializer\_class = ReviewSerializer

    def perform\_create(self, serializer):

        pk = self.kwargs['pk']

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

        serializer.save(watchlist=watchlist)

class ReviewSerializer(serializers.ModelSerializer):

    class Meta:

        model = Review

        # fields = "\_\_all\_\_"

        exclude = ('watchlist',)

<http://127.0.0.1:8000/watch/3/review-create>

    # Reviews URLs

    # I want all the reviews of a purticular movie only

    path('<int:pk>/review-create', ReviewCreate.as\_view(), name="reviewCreate"),

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

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

**User Model**

Who written the review.

1. First delete all the reviews from admin panel
2. Import the User model
3. Then add the new field in model -- review\_user (Foreign key)
4. Then register in serilizers.py

from django.contrib.auth.models import User

class Review(models.Model):

    review\_user = models.ForeignKey(User, on\_delete=models.CASCADE)

class ReviewSerializer(serializers.ModelSerializer):

    review\_user = serializers.StringRelatedField(read\_only=True)

    class Meta:

        model = Review

        exclude = ('watchlist',)

But here is one problem that same user can give multiple reviews for same movie.

Which is not good.

One User can give only one review for a movie.

For that

1. When we submitting the review so it come to reviewCreate view.
2. First we have to get the username
3. We have to filter user and movie. Basically we are checking that the user is gave review for same movie or not.
4. Then we have to check that review is present or not in review database.
5. We have to override the get\_queryset method
6. Import the exception
7. Pass the review\_user in serializer.save()

from rest\_framework.exceptions import ValidationError

class ReviewCreate(generics.CreateAPIView):

    serializer\_class = ReviewSerializer

    def get\_queryset(self):

        return Review.objects.all()

    def perform\_create(self, serializer):

        pk = self.kwargs['pk']

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

        # Checking if same user posting review 2nd time for same movie

        review\_user = self.request.user

        review\_queryset = Review.objects.filter(watchlist=watchlist, review\_user=review\_user)

        if review\_queryset.exists():

            raise ValidationError("You already gave the Review!")

        serializer.save(watchlist=watchlist, review\_user=review\_user)

**Temporary Login and Logout**

* Provided by Django itself
* Create the new user in admin panel
* You can see this new user is not a staff member and not isAdmin.
* Now logout and try to login with new user.
* You will not able to login because new user is not an admin nor staff member.

Django has Temporary login page.

Go to the main urls.pyAnd add

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

api-auth it is just a name it can be anything. Refresh the page and now check at right side you have option to logout and login now you can login thru new user.

**Django Permissions**

Permissions are used to grant or deny access for different classes of users to different parts of the API.

The simplest style of permission would be to allow access to any authenticated user, and deny access to any unauthenticated user. This corresponds to the IsAuthenticated class in REST framework.

A slightly less strict style of permission would be to allow full access to authenticated users, but allow read-only access to unauthenticated users. This corresponds to the IsAuthenticatedOrReadOnly class in REST framework.

## [How permissions are determined](https://www.django-rest-framework.org/api-guide/permissions/#how-permissions-are-determined)

Permissions in REST framework are always defined as a list of permission classes.

Before running the main body of the view each permission in the list is checked. If any permission check fails, an exceptions.PermissionDenied or exceptions.NotAuthenticated exception will be raised, and the main body of the view will not run.

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

<https://github.com/encode/django-rest-framework/blob/master/rest_framework/permissions.py>

There are 2 types of permissions.

1. It will allow on every class of project - write in settings.py
2. We can apply on particular class – (object level permissions)

And we can also use the custom permissions where we can restrict according to our requirement like user isAuthenticate, isAdmin etc.

**Settings.py**

# Django permissions

# It will apply on every single class of project

REST\_FRAMEWORK = {

    'DEFAULT\_PERMISSION\_CLASSES': [

        'rest\_framework.permissions.IsAuthenticated',

    ]

}

Object level permission

I want to apply condition on watchlist where anyone can see the movies but unauthorized user cannot post the new movie.

1. Import the permissions
2. Apply on any class where you want to restrict someone.

from rest\_framework.permissions import IsAuthenticatedOrReadOnly

class WatchListView(APIView):

    permission\_classes = [IsAuthenticatedOrReadOnly]

For example

Only Authenticate user can post the review.

from rest\_framework.permissions import

class ReviewCreate(generics.CreateAPIView):

    serializer\_class = ReviewSerializer

    permission\_classes = [IsAuthenticated]

# [**Custom permissions**](https://www.django-rest-framework.org/api-guide/permissions/#custom-permissions)

To implement a custom permission, override BasePermission and implement either, or both, of the following methods:

* .has\_permission(self, request, view)
* .has\_object\_permission(self, request, view, obj)

The methods should return True if the request should be granted access, and False otherwise.

If you need to test if a request is a read operation or a write operation, you should check the request method against the constant SAFE\_METHODS, which is a tuple containing 'GET', 'OPTIONS' and 'HEAD'. For example:

if request.method in permissions.SAFE\_METHODS:

# Check permissions for read-only request

else:

# Check permissions for write request

1. **Create folder permissions.py in api folder**
2. **Here create your custom classes**
3. **Then import them in your view and use them**

**Permissions.py**

from rest\_framework import permissions

class IsAdminOrReadOnly(permissions.IsAdminUser):

    def has\_permission(self, request, view):

        admin\_permissions = bool(request.user and request.user.is\_staff)

        return request.method == 'GET' or admin\_permissions

# or you can use if else also

#    def has\_permission(self, request, view):

#       if request.method in permissions.SAFE\_METHODS:

#            return True

#        else:

#            return bool(request.user and request.user.is\_staff)

Views.py

from watchlist\_app.api.permissions import IsAdminOrReadOnly

class ReviewDetail(generics.RetrieveUpdateDestroyAPIView):

    queryset = Review.objects.all()

    serializer\_class = ReviewSerializer

    permission\_classes = [AdminOrReadOnly]

<http://127.0.0.1:8000/watch/reviews/6/>

**go to this kind of url and now everyone can see the review but only admin can update and delete**

Only review owner can update and delete the review other can only see the review.

# Only review user can edit or delete this

class IsUserReviewOrReadOnly(permissions.BasePermission):

    def has\_object\_permission(self, request, view, obj):

        if request.method in permissions.SAFE\_METHODS:

            return True

        else:

            return obj.review\_user == request.user

**views.py**

import watchlist\_app.api.permissions import IsUserReviewOrReadOnly

class ReviewDetail(generics.RetrieveUpdateDestroyAPIView):

    queryset = Review.objects.all()

    serializer\_class = ReviewSerializer

    # permission\_classes = [AdminOrReadOnly]

    permission\_classes = [UserReviewOrReadOnly]

Rating calculations

1. Add 2 new fields in WatchList model
2. Add the logic in ReviewCreate class.
3. Suppose we deleted any review so it should decrease the number\_review.
4. Write the logic in delete review method or use serialization Field and core argument

class WatchList(models.Model):

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

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

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

    active = models.BooleanField(default=False)

    average\_rating = models.FloatField(default=0)

    number\_rating = models.IntegerField(default=0)

    def perform\_create(self, serializer):

        pk = self.kwargs['pk']

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

        # Checking if same user posting review 2nd time for same movie

        review\_user = self.request.user

        review\_queryset = Review.objects.filter(watchlist=watchlist, review\_user=review\_user)  # here i am cheking on both movie and

                                                                                                # user.It means same movie or same user

        if review\_queryset.exists():

            raise ValidationError("You already gave the Review!")

        # Reting calculation after the isExist condition bcz we want to first check that is they Gave review or not otherwise it affect our calculations

        if watchlist.number\_rating == 0:

            watchlist.average\_rating = serializer.validated\_data['rating']

        else:

            watchlist.average\_rating = (watchlist.average\_rating + serializer.validated\_data['rating']) / 2

        watchlist.number\_rating += 1

        watchlist.save()

        serializer.save(watchlist=watchlist, review\_user=review\_user)

**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](https://www.django-rest-framework.org/api-guide/permissions/) and [throttling](https://www.django-rest-framework.org/api-guide/throttling/) policies can then use those credentials to determine if the request should be permitted.

**Note:** Don't forget that **authentication by itself won't allow or disallow an incoming request**, it simply identifies the credentials that the request was made with.

There are 4 types of authentications.

1. Basic Authentication
2. Token Authentication
3. Session Authentication
4. Remote User Authentication

Token Authentication is Most Important

**Comment the Temporary Django Login Url in main url**

**Basic Authentication**

It is used only for testing purpose

Paste in the settings.py

REST\_FRAMEWORK = {

    'DEFAULT\_AUTHENTICATION\_CLASSES': [

        'rest\_framework.authentication.BasicAuthentication',

    ]

}

Views.py

IsAthentication using permission class. Now when you visit this you will now get pop up login box.

class ReviewList(generics.ListAPIView):

    # queryset = Review.objects.all()

    serializer\_class = ReviewSerializer

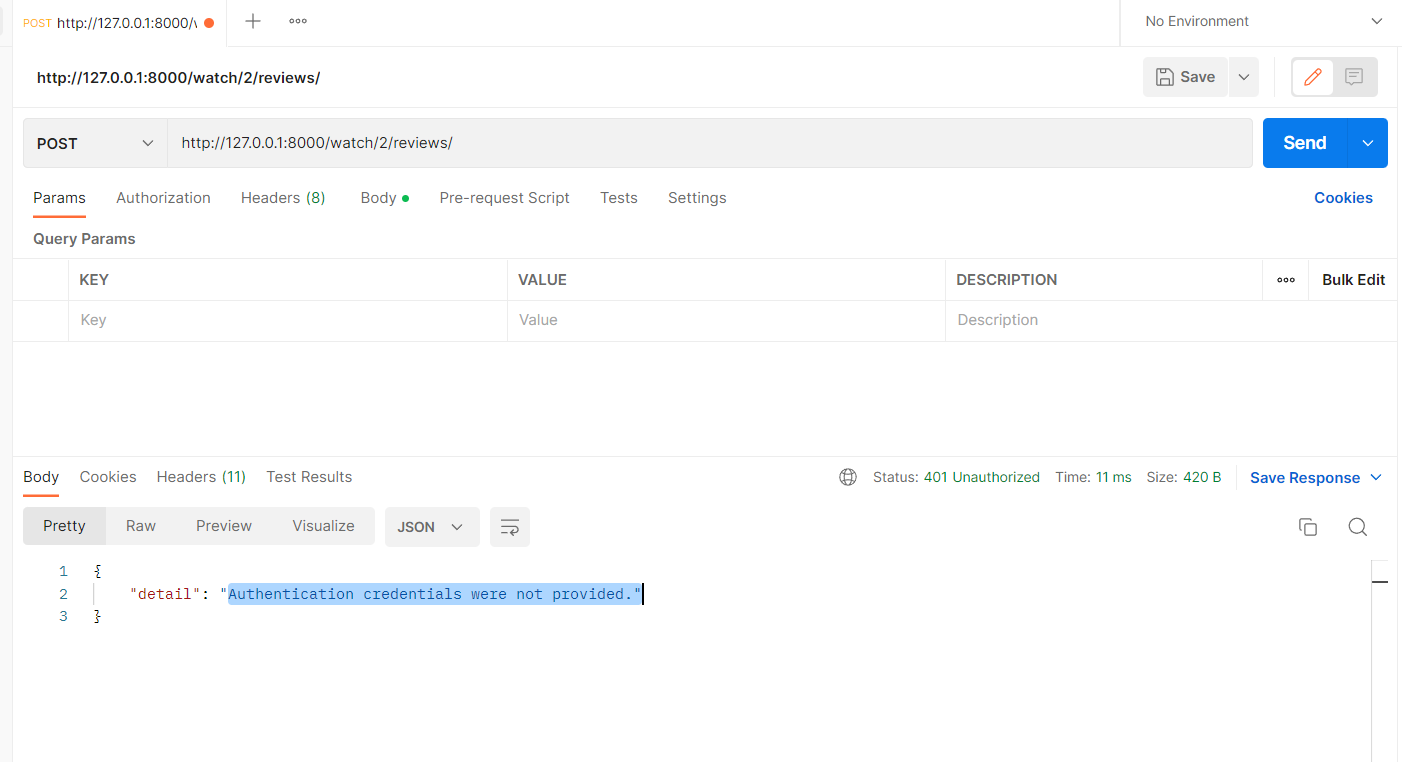
    permission\_classes = [IsAuthenticated]

<http://127.0.0.1:8000/watch/2/reviews/>

Here you will get pop up box.

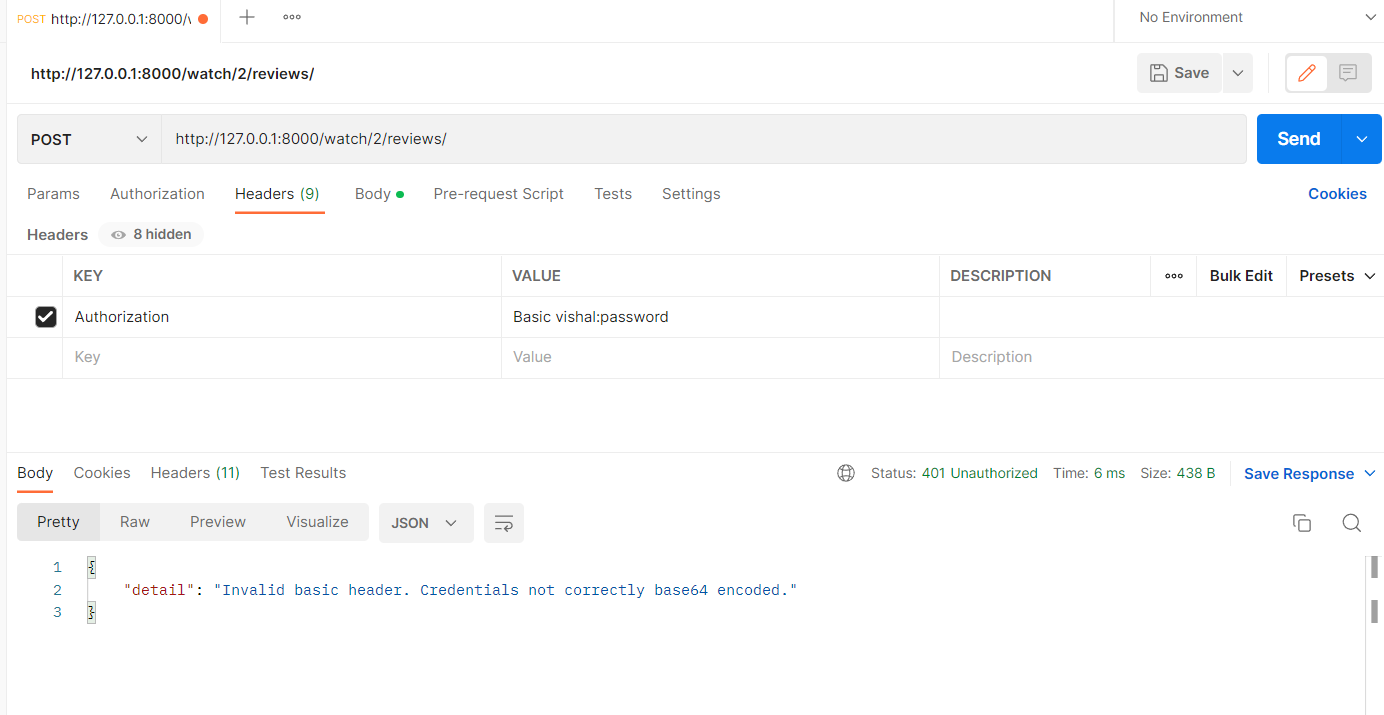
Basic Authentication using Postman

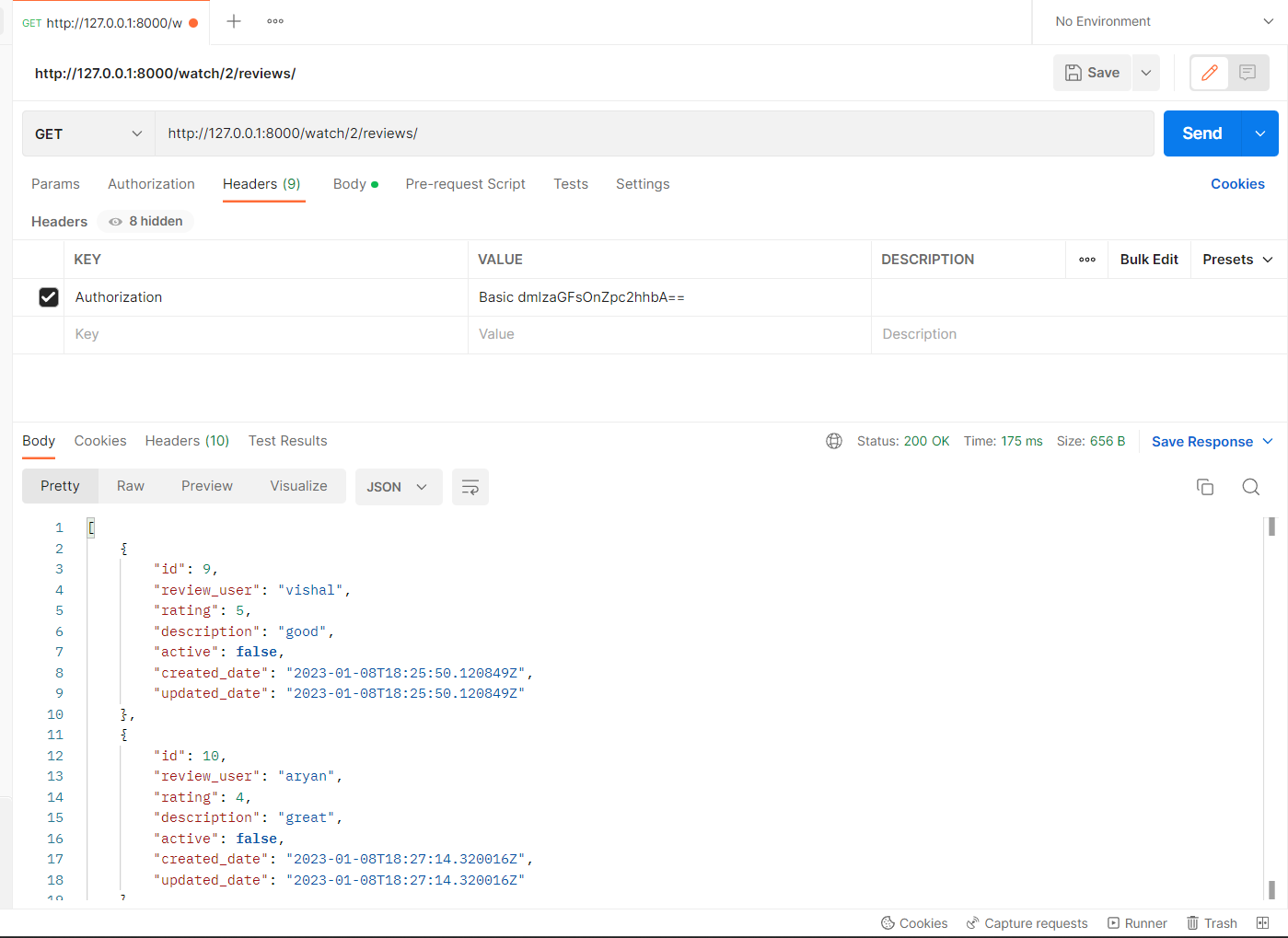
<http://127.0.0.1:8000/watch/2/reviews/>



So here first we have to give user name and password

1. Go to the header
2. Key - > Authorized
3. Value - > Basic username:password but password should be password
4. For eg you password in vishal but you have to write password so you will get this kind of error.
5. Now you can see base64 so you have to encode your username:password then paste it. <https://www.base64encode.org/> (vishal:vishal) encode it





**Token Authentication**

Token authentication refers to exchanging username and password for a token that will be used in all subsequent requests so to identify the user on the server side. The token authentication works by providing token in exchange for exchanging usernames and passwords.

Settings.py

REST\_FRAMEWORK = {

    # Basic Authentication

    # 'DEFAULT\_AUTHENTICATION\_CLASSES': [

    #     'rest\_framework.authentication.BasicAuthentication',

    # ],

    # Token Authentication

    'DEFAULT\_AUTHENTICATION\_CLASSES': [

        'rest\_framework.authentication.TokenAuthentication',

    ]

}

And register the authtoken in installed app and migrate it.

INSTALLED\_APPS = [

'rest\_framework.authtoken'

]

Make sure to run manage.py migrate after changing your settings.

Python manage.py migrate

<http://127.0.0.1:8000/admin/>

login and left top you can see the Auth token now add the token for users.

Now go to the postman

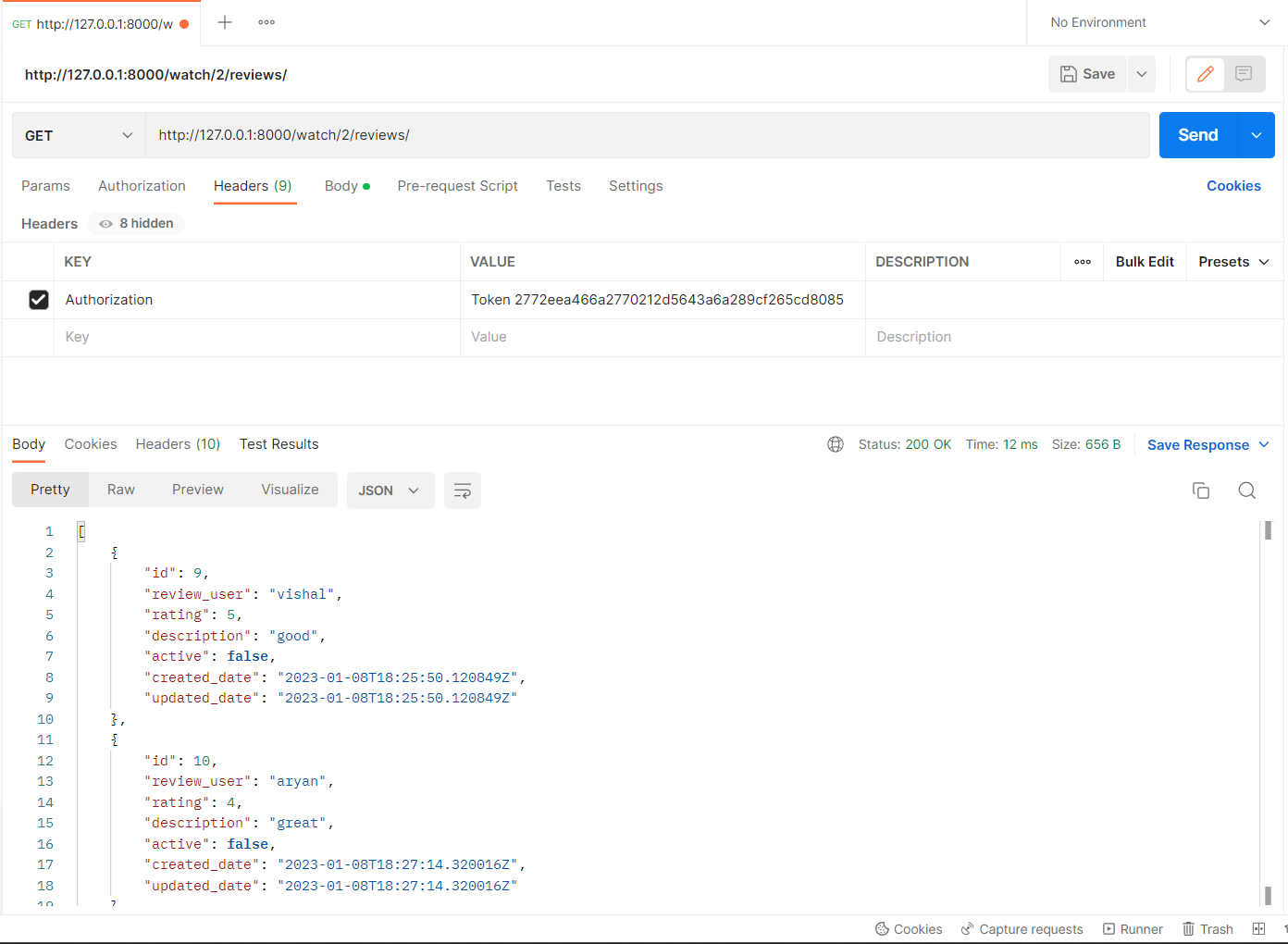
<http://127.0.0.1:8000/watch/2/reviews/>

try to get data you will get error bcz you have not authorized using token

Copy the token from admin panel of Vishal user and now

Header

Authorizerd Token 2772eea466a2770212d5643a6a289cf265cd8085



I have a review for id 10

<http://127.0.0.1:8000/watch/reviews/10>

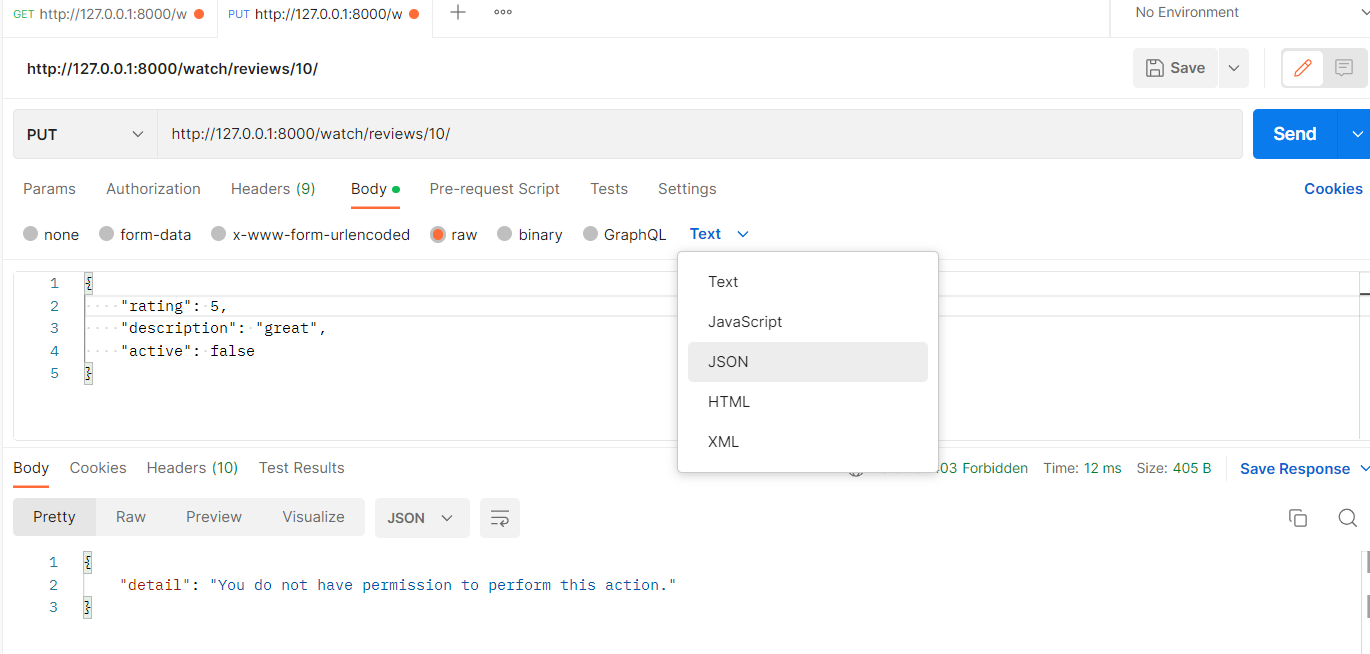
if now I want to try to update the review which is posted by Aryan thru vishal’s token key so I will get error.

{

    "detail": "You do not have permission to perform this action."

}

1. For the update we have to select post method left side of URL in postman
2. Copy the formate of jason
3. Go to the:
   1. Body
      1. Select Row
         1. Select JSON
4. Same as Image.
5. And Now authorizerd by vishal token key



Solution is copy the aryan’s token and authorized by aryan

**Now we don’t want to create Token manually from admin pannel**

**It should created automatically.**

Now Create the new app for user

Django-admin startapp user\_app

Create the api folder

Views.py

Urls.py

Serializers.py

Main urls.py

urlpatterns = [

    path('admin/', admin.site.urls),

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

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

    path('account/', include('user\_app.api.urls')),

]

User\_app -> urls.py

1. Import the ObtainAuthToken
2. Use class based view as\_view()

from django.urls import path

from rest\_framework.authtoken.views import ObtainAuthToken

urlpatterns = [

    path('login/', ObtainAuthToken.as\_view(), name='login'),

]

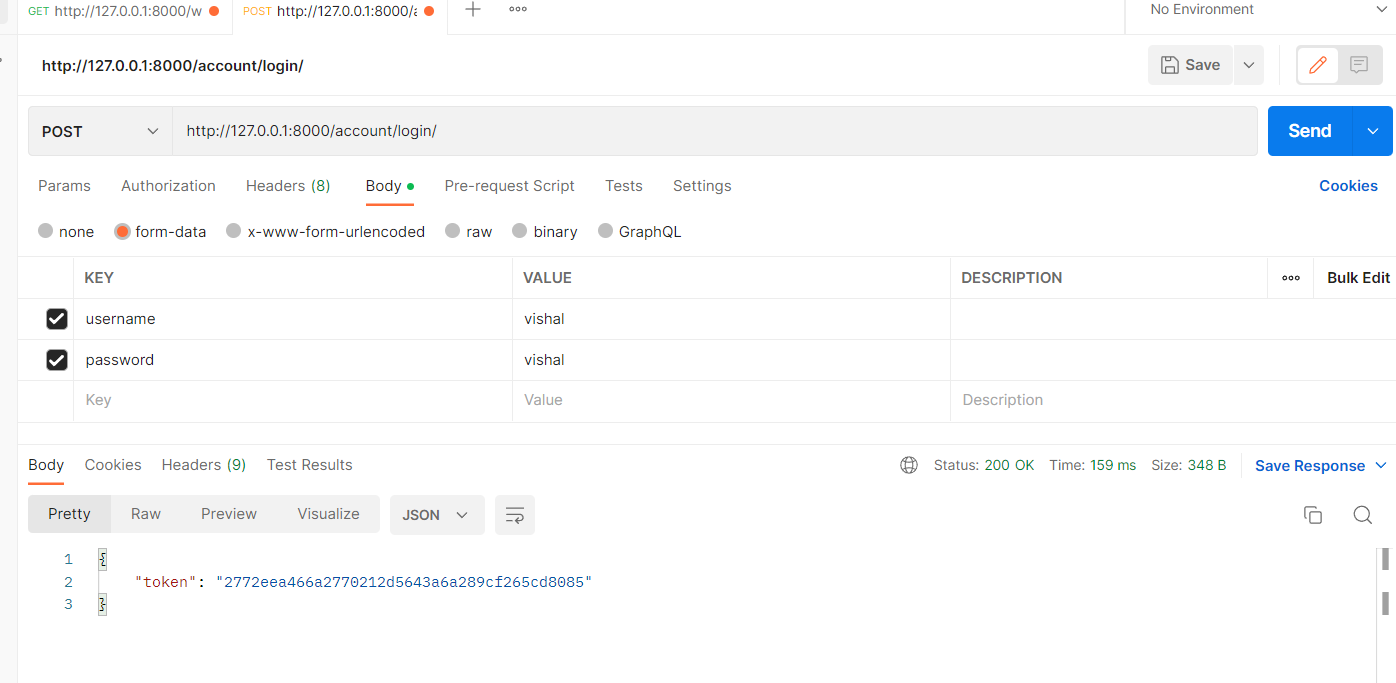
<http://127.0.0.1:8000/account/login/>

* now go to the body
* select row data

username vishal

password vishal

**You will get the Token for vishal user**



Now you can use this Token to every other link.

Now practice the postman and try to practice with other urls and data.

**Registration of New User**

1. create the register serializer

from rest\_framework import serializers

from django.contrib.auth.models import User

class RegistrationSerializer(serializers.ModelSerializer):

    password2 = serializers.CharField(write\_only=True)

    class Meta:

        model = User

        fields = ['username','email','password','password2']

        # we added new field for confirm password (password2)

        extra\_kwargs = {

            'password': {'write\_only':True},

        }

1. create the view for registration

from rest\_framework.response import Response

from rest\_framework import status

from rest\_framework.decorators import api\_view

from user\_app.api.serializers import RegistrationSerializer

@api\_view(['POST'])

def user\_registration(request):

    if request.method == 'POST':

        serializer = RegistrationSerializer(data=request.data)

        if serializer.is\_valid():

            serializer.save()

            return Response(serializer.data)

1. Create the url

from user\_app.api.views import user\_registration

urlpatterns = [

    path('login/', ObtainAuthToken.as\_view(), name='login'),

    path('register/', user\_registration, name='register'),

]

Now go to the postman

<http://127.0.0.1:8000/account/register/>

You will get the error create error. It means the problem is we are able to saving the user

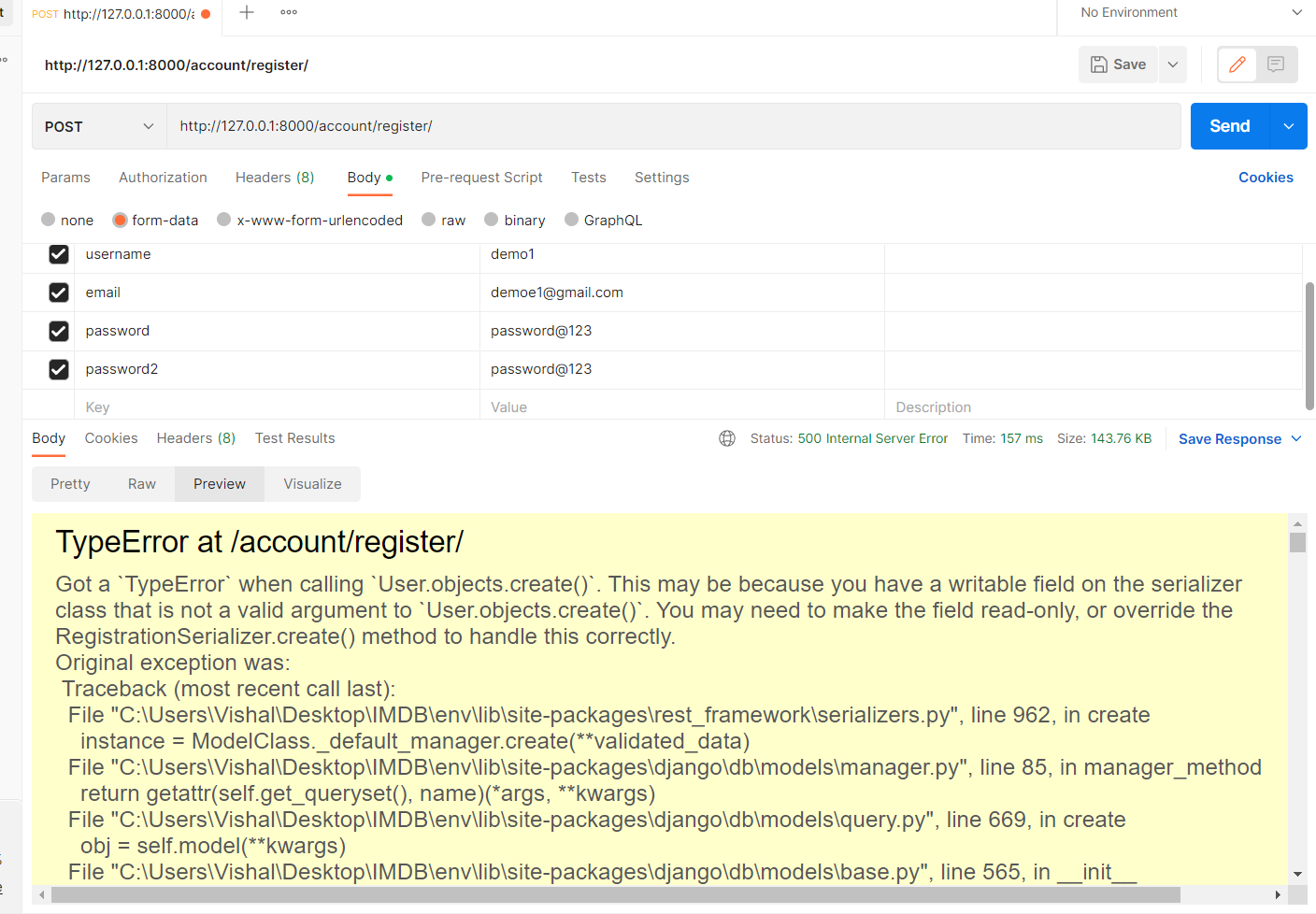
We have to override the save method.

Postman

Body

Form dada

Username, email, password, password2



**Override the save method in registerSerializer**

from rest\_framework import serializers

from django.contrib.auth.models import User

class RegistrationSerializer(serializers.ModelSerializer):

    def save(self):

        password = self.validated\_data['password']

        password2 = self.validated\_data['password2']

        if password != password2:

            raise serializers.ValidationError({'error':'password not matched'})

        if User.objects.filter(email=self.validated\_data['email']).exists():

            raise serializers.ValidationError({'error':'email already exists'})

        account = User(username=self.validated\_data['username'], email=self.validated\_data['email'])

        account.save()

        return account

Now we are able to register the new user check in admin pannel / users

But we are not able to generate the token.

**Token generating automatically when user register.**

<https://www.django-rest-framework.org/api-guide/authentication/#tokenauthentication>

first generate the token

### [**Generating Tokens**](https://www.django-rest-framework.org/api-guide/authentication/#generating-tokens)

#### [**By using signals**](https://www.django-rest-framework.org/api-guide/authentication/#by-using-signals)

If you want every user to have an automatically generated Token, you can simply catch the User's post\_save signal.

Paste this code in models.py - > user\_app

from django.conf import settings

from django.db.models.signals import post\_save

from django.dispatch import receiver

from rest\_framework.authtoken.models import Token

@receiver(post\_save, sender=settings.AUTH\_USER\_MODEL)

def create\_auth\_token(sender, instance=None, created=False, \*\*kwargs):

    if created:

        Token.objects.create(user=instance)

Now we have to take the account which is returning from RegisteratioSerializer in views

1. Create the dictionary fro showing all the information as response to user
2. Make variable for save
3. Account = serializer.save()
4. Account has all the variables/fields of user
5. Import the token class
6. Then get the token key

from rest\_framework.authtoken.models import Token

from user\_app.api.serializers import RegistrationSerializer

from user\_app import models

@api\_view(['POST'])

def user\_registration(request):

    if request.method == 'POST':

        serializer = RegistrationSerializer(data=request.data)

        data = {}

        if serializer.is\_valid():

            account = serializer.save()

            data['Response'] = 'Registration succefully!'

            data['username'] = account.username

            data['email'] = account.email

            # generate the token automatically for valid user   -> import the Token class

            token = Token.objects.get(user = account).key

            data['token'] = token

        else:

            data = serializer.errors

        return Response(data)

if you get this kind of error

# **DoesNotExist at /account/register/**

Token matching query does not exist.

|  |  |
| --- | --- |
| **Request Method:** | POST |
| **Request URL:** | http://127.0.0.1:8000/account/register/ |
| **Django Version:** | 4.1.5 |
| **Exception Type:** | DoesNotExist |
| **Exception Value:** | Token matching query does not exist. |

It means you have to import models in your view

from user\_app import models

**I skip the JWT (JSON Web Token) .**

**It will affect my furture code.**

**Vishal you should cover this in future.**

# [**Throttling**](https://www.django-rest-framework.org/api-guide/throttling/#throttling)

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

* Basically we will restrict the users (It may be Unknown user or Authenticated user).
* Like Unknow user can send 5 request on our website in a day.
* Authenticated user can send suppose 10 request in a day.
* (Depend on how we implement our Throtting rate.)
* I can be seconds, minute, hour or day. Like 100 request per second or 100/day

Throttling is similar to [permissions](https://www.django-rest-framework.org/api-guide/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.

As with permissions, multiple throttles may be used. Your API might have a restrictive throttle for unauthenticated requests, and a less restrictive throttle for authenticated requests.

The X-Forwarded-For HTTP header and REMOTE\_ADDR WSGI variable are used to uniquely identify client IP addresses for throttling. If the X-Forwarded-For header is present then it will be used, otherwise the value of the REMOTE\_ADDR variable from the WSGI environment will be used.

Global Throtting

It will apply on our whole website.

Like you have only 3 request for this website.

If you see the whole watchlist – 1 request and if you click 3 time on request button (or send request or refresh the page.) now you can able to send request again on every single page of this website.

You can not see the details of movie, review and all.

{

    "detail": "Request was throttled. Expected available in 86394 seconds."

}

You will get this kind of error.

Settings.py

REST\_FRAMEWORK = {

    # Basic Authentication

    # Token Authentication

    'DEFAULT\_AUTHENTICATION\_CLASSES': [

        'rest\_framework.authentication.TokenAuthentication',

    ],

    'DEFAULT\_THROTTLE\_CLASSES': [

        'rest\_framework.throttling.AnonRateThrottle',

        'rest\_framework.throttling.UserRateThrottle'

    ],

    'DEFAULT\_THROTTLE\_RATES': {

        'anon': '2/day',

        'user': '5/day'

    }

}

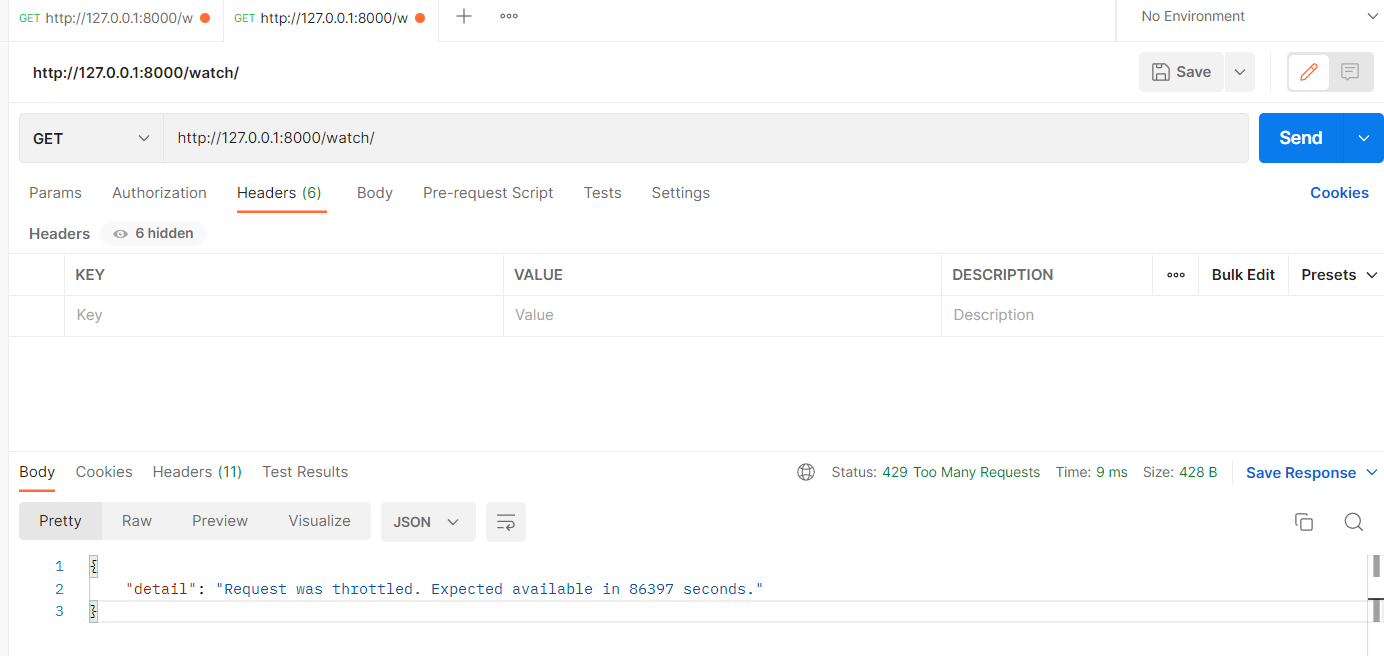
Here we have we user

Anon -> user who did not register on our website.

User -> Authenticate user -> you can use the Token for authenticate user in postman.

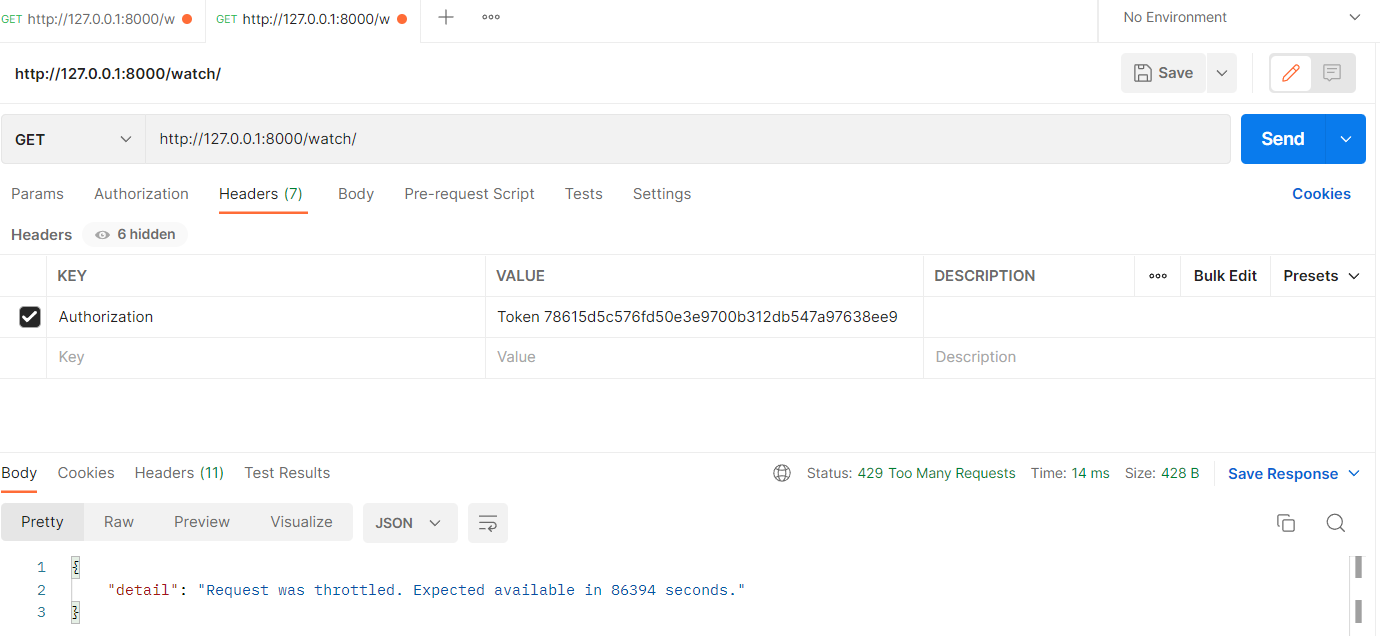
Or login in admin and see on the web browsable.

**Anon -> Unknown user**



Here if open this link for 2 time so it will work after that if you click on send button so you will get error.

**Authenticate user**



Local setting

Controlling each class.

Here we can restrict on specific URL or Page where user can send limited request.

1. Comment the first part of throttling in settings.py

# 'DEFAULT\_THROTTLE\_CLASSES': [

    #     'rest\_framework.throttling.AnonRateThrottle',

    #     'rest\_framework.throttling.UserRateThrottle'

    # ],

    'DEFAULT\_THROTTLE\_RATES': {

        'anon': '2/day',

        'user': '5/day'

    }

This commented part comment in your settings.py

1. Import the User and Anon in views.py (watchlist\_app view)

from rest\_framework.throttling import AnonRateThrottle, UserRateThrottle

class ReviewList(generics.ListAPIView):

    serializer\_class = ReviewSerializer

    # Local Throtting

    throttle\_classes = [UserRateThrottle, AnonRateThrottle]

    def get\_queryset(self):

        pk = self.kwargs['pk']

        return Review.objects.filter(watchlist=pk)

class ReviewDetail(generics.RetrieveUpdateDestroyAPIView):

    queryset = Review.objects.all()

    serializer\_class = ReviewSerializer

    permission\_classes = [IsUserReviewOrReadOnly]

    # Local Throtting

    throttle\_classes = [UserRateThrottle, AnonRateThrottle]

In above picture you can see I use the AuserRateThrttle class as well as AnonRateTrottle class for same view. Or same page.

Check in postman for this 2 urls only. Other urls have no restrictions

<http://127.0.0.1:8000/watch/2/reviews/>

**Scope rate Throttling**

1. Create throttling file and create scope classes for the pages.
2. Import directly scopeRateThrottling and use in our view itself.

1. Create file in watchlist\_app -> api -> throttling.py

**Throttling.py**

from rest\_framework.throttling import UserRateThrottle

class ReviewCreateThrottle(UserRateThrottle):

    scope = "review-create"

class ReviewListThrottle(UserRateThrottle):

    scope = "review-list"

**views.py (watchlist\_app)**

# Scop rate throttling

from watchlist\_app.api.trottling import ReviewCreateThrottle, ReviewListThrottle

class ReviewCreate(generics.CreateAPIView):

    serializer\_class = ReviewSerializer

    permission\_classes = [IsAuthenticated]

    # Scope rate trottle

    throttle\_classes = [ReviewCreateThrottle]

class ReviewList(generics.ListAPIView):

    serializer\_class = ReviewSerializer

    # Scope rate throttling

    throttle\_classes = [ReviewListThrottle, AnonRateThrottle]

**settings.py**

define the scope rate

 'DEFAULT\_THROTTLE\_RATES': {

        'anon': '2/day',

        # 'user': '15/day',

        # Scope rate throttling

        'review-create': '2/day',

        'review-list': '10/day',

    }

**2 -> Import the ScopeRateThrottle class**

Here we don’t need to create a file for throttling

We can directly define the scope inside the view only.

1. Import the ScopeRateThrottle class
2. Go to view
3. Create the throttle\_class and throttle\_scope

Like this

**Views.py**

class ReviewDetail(generics.RetrieveUpdateDestroyAPIView):

    queryset = Review.objects.all()

    serializer\_class = ReviewSerializer

    throttle\_classes = [ScopedRateThrottle]

    throttle\_scope = 'review-detail'

and now define the scope rate in settings.py

 'DEFAULT\_THROTTLE\_RATES': {

        'anon': '2/day',

        # 'user': '15/day',

        # Scope rate throttling

        'review-create': '2/day',

        'review-list': '10/day',

        "review-detail": '5/day'

    }

**Filtering, Searching, Ordering**

We can see on any side there are 3 types of filtering.

1. We search thru search box
2. We filter by different parameters like price, posted date etc.
3. We order them like new to old, low to high etc.

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

The default behavior of REST framework's generic list views is to return the entire queryset for a model manager. Often you will want your API to restrict the items that are returned by the queryset.

The simplest way to filter the queryset of any view that subclasses GenericAPIView is to override the .get\_queryset() method.

Overriding this method allows you to customize the queryset returned by the view in a number of different ways.

**We have to override the get\_queryset() method**

## [Filtering against the URL](https://www.django-rest-framework.org/api-guide/filtering/#filtering-against-the-url)

Another style of filtering might involve restricting the queryset based on some part of the URL.

Views.py

class UserReview(generics.ListAPIView):

    serializer\_class = ReviewSerializer

    def get\_queryset(self):

        username = self.kwargs['username']

        return Review.objects.filter(review\_user\_\_username = username)

Review.objects.filter(review\_user\_\_username = username)

Here you can see I user \_\_ (double underscore with review\_user) bcz we are fetching the username and it is the string field.

**And here review\_user is known as Foreign key**

URLS.py watchlist\_app

path('reviews/<str:username>/', UserReview.as\_view(), name='user-review-detail'),

here we used <str:username>

Postman

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

you will get all reviews of vishal

## [Filtering against query parameters](https://www.django-rest-framework.org/api-guide/filtering/#filtering-against-query-parameters)

A final example of filtering the initial queryset would be to determine the initial queryset based on query parameters in the url.

Views.py

class UserReview(generics.ListAPIView):

    serializer\_class = ReviewSerializer

    # Filtering against query parameters

    def get\_queryset(self):

        username = self.request.query\_params.get('username')

        return Review.objects.filter(review\_user\_\_username = username)

username = self.request.query\_params.get('username')

Here we are mapping the parameter with url

Urls.py

path('review/', UserReview.as\_view(), name='user-review-detail'),

Here you don’t need to use the <str:username> it will automatically fetch by urls.

Like this ?username=vishal

<http://127.0.0.1:8000/watch/review/?username=vishal>

Here you don’t need of end slash

# [**Generic Filtering**](https://www.django-rest-framework.org/api-guide/filtering/#generic-filtering)

<https://www.django-rest-framework.org/api-guide/filtering/#generic-filtering>

As well as being able to override the default queryset, REST framework also includes support for generic filtering backends that allow you to easily construct complex searches and filters.

Generic filters can also present themselves as HTML controls in the browsable API and admin API.

This will only work generic class based view.

If you are using the class or function based based view (APIView or api\_view) then use the above filtering techniques.

1. First install the Django-filter

Pip install Django-filter

1. Register in install app

INSTALLED\_APPS = [

'django\_filters',

]

**Here I created the WatchListTesting view only for the learning purpose of generic Filtering, searching and ordering**

**Views.py**

Import the package first

from django\_filters.rest\_framework import DjangoFilterBackend

class WatchListTesting(generics.ListCreateAPIView):

    queryset = WatchList.objects.all()

    serializer\_class = WatchListSerializer

    filter\_backends = [DjangoFilterBackend]

    filterset\_fields = ['title', 'platform\_\_name']

**Here I use the platform\_\_name**

**Why this name is came with platform**

**Bcz we plateform is a foreign key and platform has 3 field [ name, about, website]**

**And we want name i.e why we use platform\_\_name.**

urls.py

    path('watchlist-testing/', WatchListTesting.as\_view(), name='testing'),

Postman

<http://127.0.0.1:8000/watch/watchlist-testing/?title=java>

for platform filtering

<http://127.0.0.1:8000/watch/watchlist-testing/?title=&platform__name=Netflix>

go to browser and paste this link you will get filter option over ther

here It will filter exact same match like java so only if you use like title=ja then it will give us empty list or exact match of ja

Solution is to use filter

## [SearchFilter](https://www.django-rest-framework.org/api-guide/filtering/#searchfilter)

The SearchFilter class supports simple single query parameter based searching, and is based on the [Django admin's search functionality](https://docs.djangoproject.com/en/stable/ref/contrib/admin/#django.contrib.admin.ModelAdmin.search_fields).

When in use, the browsable API will include a SearchFilter control:

First import filter

Then use the fields

Views.py

from rest\_framework import filters

class WatchListTesting(generics.ListCreateAPIView):

    queryset = WatchList.objects.all()

    serializer\_class = WatchListSerializer

    # filtering

    # filter\_backends = [DjangoFilterBackend]

    # filterset\_fields = ['title', 'platform\_\_name']

    # seraching

    filter\_backends = [filters.SearchFilter]

    search\_fields = ['title', 'plateform\_\_name']

<http://127.0.0.1:8000/watch/watchlist-testing/?search=netflix>

<http://127.0.0.1:8000/watch/watchlist-testing/?search=java>

now you can get the movie java also

<http://127.0.0.1:8000/watch/watchlist-testing/?title=ja&platform__name=netflix>

you can use & for multiple search

<http://127.0.0.1:8000/watch/watchlist-testing/?search=java&hotstar>

## [OrderingFilter](https://www.django-rest-framework.org/api-guide/filtering/#orderingfilter)

The OrderingFilter class supports simple query parameter controlled ordering of results.

url

http://example.com/api/users?ordering=-username

With – (minus) descending order)

<http://127.0.0.1:8000/watch/watchlist-testing/?ordering=-average_rating>

Ascending Order

<http://127.0.0.1:8000/watch/watchlist-testing/?ordering=average_rating>

class WatchListTesting(generics.ListCreateAPIView):

    queryset = WatchList.objects.all()

    serializer\_class = WatchListSerializer

    # Ordering Filtering

    filter\_backends = [filters.OrderingFilter]

    ordering\_fields = ['average\_rating']

Project update

1. Comment the showing review on landing page
   1. Serializers.py

class WatchListSerializer(serializers.ModelSerializer):

    # reviews = ReviewSerializer(many=True, read\_only=True)

1. **Showing the name of platform instead of id**

(We just defining the field of platform with it’s name)

class WatchListSerializer(serializers.ModelSerializer):

    # reviews = ReviewSerializer(many=True, read\_only=True)

    # overiding the plateform name instead of id

    platform = serializers.CharField(source="platform.name")