

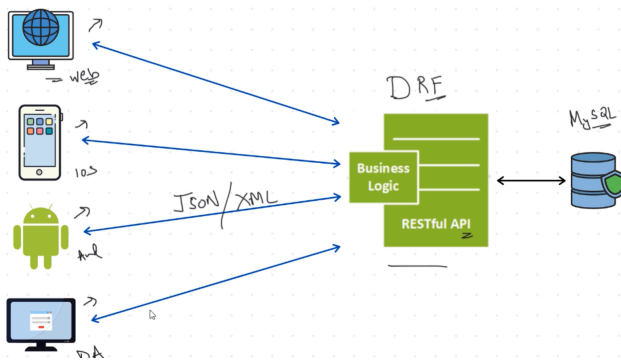
LEC 1:

Api: Application programming interface : use to connect two application/project, works as a middleman between two application/project

Types:

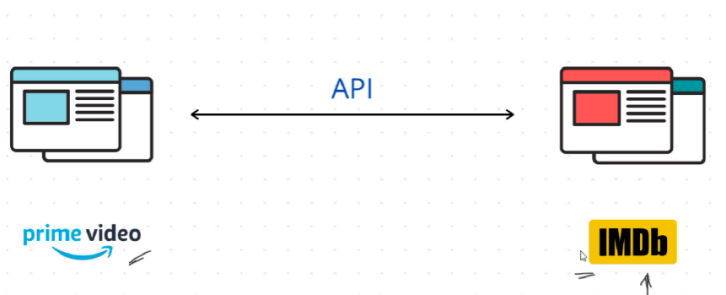
1)Private (with organization) 2) Partner(with business) 3)public Api(3rd party app)

1) Private API



All left side can have different tech.. Like for IOS flutter, any website can have angular. We connect with Api single backend that done in django.everything with organization can access the API no other 3rd party application can access this API.

2)Partner API



In future if netflix comes to access IMDB api so after some process they can use, only selective user can access this API.ex:insta api,uber

3)Public API

Anyone can access this API , just register and use it.ex:crypto api

Rest Api: uses the REST architectural style to connect applications and components



API + REST Architecture → REST API

- | | |
|-------------------|--------------------------|
| 1. End Points | 3. Headers (Status Code) |
| 2. Methods (CRUD) | 4. The Data (JSON) |

```
C:\Users\Devansh shrimali\Desktop\drf-project>python -m venv env
```

```
C:\Users\Devansh shrimali\Desktop\drf-project>
```

```
C:\Users\Devansh shrimali\Desktop\drf-project>env\Scripts\activate
```

```
(env) C:\Users\Devansh shrimali\Desktop\drf-project> freeze
'freeze' is not recognized as an internal or external command,
operable program or batch file.
```

```
(env) C:\Users\Devansh shrimali\Desktop\drf-project>pip freeze
```

```
(env) C:\Users\Devansh shrimali\Desktop\drf-project>
```

LEC 2:

DRF: Django rest framework

Back to Front => serialization and front to back => Deserialization

serialization: Complex data => python native data => json (Back to Front)

Deserialization: json => Python native data => complex data (front to back)

Two types of serialization :

1)serializers.Serializer and 2)serializers.ModelSerializer

Two types of views: 1) class base and 2)function base

Working with API:

DRF browsable API, postman, HTTPie

Get objects using serializers (GET)

Map all things in serializer.py

Call this in views.py

```
#with serializer we map all things in another file
#The core of this functionality is the api_view decorator, which takes a
list of HTTP methods that your view should respond to
#@api_view(['GET', 'POST']) we are accepting GET and POST
#default there is get request and get means back to front
#@api_view()

@api_view(['GET', 'POST'])
def movie_list(request):
    if request.method == 'GET': #back to front
        movies = Movie.objects.all()
        serializer = MovieSerializer(movies,many=True)#our serializer will
visit multiple objects and serialize(map) them
        return Response(serializer.data)
```

Get =send objects back to front

Post =send objects front to back

Put =update object

delete= delete object

Create objects using serializers (post)

Use Create method in serializers.py

```
def create(self, validated_data): #this method is called when we want to
create a new object
    return Movie.objects.create(**validated_data)
```

Now make call this method in views.py

```
#we get data from user now we create object using serializer using create
method
    if request.method == 'POST': #front to back
        serializer=MovieSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()#save() will save data in database
            return Response(serializer.data)
        else:
            return Response(serializer.errors)
```

Update objects using serializers (PUT)

Use update method in serializers.py

```
def update(self, instance, validated_data): #this method is called when we
want to update an existing object
    instance.name = validated_data.get('name', instance.name)
    instance.description = validated_data.get('description',
instance.description)
    instance.active = validated_data.get('active', instance.active)
    instance.save()
    return instance
```

Now call this method in views.py

```
if request.method == 'PUT': #we are updating object
    movie = Movie.objects.get(pk=pk)
    serializer = MovieSerializer(movie, data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data)
    else:
        return Response(serializer.errors)
```

Delete objects using serializers (DELETE)

For this we don't need to anything in serializer.py just get item and delete it in views.py

```
if request.method == 'DELETE': #we are deleting object
    movie=Movie.objects.get(pk=pk)
    movie.delete()
    return Response(status=204)
```

Status code uses :

```
return Response(status=status.HTTP_204_NO_CONTENT)
```

```
return Response({'Error':Not Found'}, status=status.HTTP_404_NOT_FOUND)
```

Validator

Validators can be useful for re-using validation logic between different types of fields.

Field level validator

```
#this is field level validation
def validate_name(self, value):
    if len(value)<2:
        raise serializers.ValidationError("Name is too sort") # we
can add some validation here like check if name length
    else:
        return value
```

Object level

```
#object level validation
def validate(self, data):
    if data['name'] == data['description']:
        raise serializers.ValidationError("Name and Description should
not be same")
    else:
        return data
```

```
#use of validators for validation
def name_length(value):
    if len(value)<2:
        raise serializers.ValidationError("Name is too sort") # we can
add some validation here like check if name length
```

```
name = serializers.CharField(validators=[name_length])
```

Serializer Field

Read-only,Write-only

Model Serializer

```
#Model serializer
#here set of default fields are automatically included,a set of validaties
are automatically included
#we have .create() and.update() methods for creating and updating objects
respectively
```

```
class MovieSerializer(serializers.ModelSerializer):
    class Meta: # We are mapping all fields in our model to serializer
        model = Movie
        #fields = ('id', 'name', 'description', 'active') # we only want
to serialize these fields
        fields = '__all__' # we want to serialize all fields
automatically
        #exclude = ['name'] # we exclude name field from serialization
```

Custom Serializer Field

```
len_name= serializers.SerializerMethodField() # we calculate length of
name field in our serializer
```

```
def get_len_name(self, obj): # we define a method in serializer to
calculate length of name
    return len(obj.name)
```

Class based views

Here we iterate over the whole list and do operation like get ,post

```
class WatchListAV(APIView):

    def get(self, request):
        movies = Watchlist.objects.all()
        serializer = WatchlistSerializer(movies, 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,
status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```

Here we iterate over particular object and do operation like get,put,delete

```
class WatchDetailAV(APIView):

    def get(self,request, pk):
        try:
            movie = Watchlist.objects.get(pk=pk)

        except Movie.DoesNotExist:
            return Response({'Error': 'Not
Found'}, status=status.HTTP_404_NOT_FOUND)
```

```

        serializer = WatchlistSerializer(movie)
        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,status=status.HTTP_400_BAD_REQUEST)

    def delete(self, request, pk):
        movie=Watchlist.objects.get(pk=pk)
        movie.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)

```

Django Relationship

one to one , one to many, many to one , many to many

```

platform = models.ForeignKey(StreamPlatform, on_delete=models.CASCADE,
related_name='watchlist', null=True)

```

here one movie can have many platforms

Now we want to show the movie connected to which platform

So we have to update our serializers.py(mapping)

```

class StreamPlatformSerializer(serializers.ModelSerializer):
    #here we use watchlist coz we define related name as watchlist in
models.py
    watchlist= watchListSerializer(many=True, read_only=True) # many=True
means this field can have many watchlists associated with it
    class Meta:
        model = StreamPlatform
        fields = '__all__'

```


Serializer Relationship

```
#watchlist= WatchListSerializer(many=True, read_only=True) # many=True
means this field can have many watchlist associated with it
watchlist=serializers.StringRelatedField(many=True)
```

It just show the name of title

```
def __str__(self):
    return self.title
```

Insted of this we can show only primary key, link

```
#watchlist= WatchListSerializer(many=True, read_only=True) # many=True
means this field can have many watchlist associated with it
#watchlist=serializers.StringRelatedField(many=True)
watchlist=serializers.HyperlinkedRelatedField(
    many=True,
    read_only=True,
    view_name='movie_detail'
)
```

Hyperlink model serializer: instead of id we can accces the url

Class Base view 1st method is APIview

In Django REST Framework (DRF), **class-based views (CBVs)** and the **APIView** method provide a structured way to implement RESTful APIs

APIView is the base class for creating class-based views in DRF. It extends Django's **View** class to add RESTful behavior.

Workflow of Class-Based Views in Your Code

1. Request Handling:

- The client sends an HTTP request (e.g., GET or POST).
- The URL resolver passes the request to the appropriate view class.

2. Class Methods:

- The `APIView` dispatches the request to the appropriate method (e.g., `get`, `post`, `put`, `delete`) based on the HTTP method.

3. Data Serialization:

- Data is serialized or deserialized using the appropriate serializer (e.g., `StreamPlatformSerializer`, `WatchListSerializer`).

4. Response:

- The view returns a structured HTTP response (e.g., JSON) to the client

Class Base view 2nd method is Mixins

In Django REST Framework (DRF), **mixins** are reusable building blocks that provide common functionality for class-based views. They simplify CRUD (Create, Retrieve, Update, Delete) operations without the need to reimplement common logic.

`mixins.RetrieveModelMixin:`

- Provides the functionality to retrieve a single model instance by its primary key (`pk`).

`generics.GenericAPIView:`

- A base class that connects the mixin with the `queryset` and `serializer_class`

`mixins.ListModelMixin:`

- Provides functionality to list all objects in the `queryset`.

`mixins.CreateModelMixin:`

- Provides functionality to create a new object from the request data.

generics.GenericAPIView:

- Acts as the foundation that connects the mixins with the required attributes (`queryset`, `serializer_class`).

Class Base view 3rd method is Generic class base view

=>concrete view classes

In this we don't have write

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

Here all this are inbuilt

Before we have to define all this methods get,put ,delete , methos

```
def get(self, request):  
  
    movies = Watchlist.objects.all()  
  
    serializer = WatchListSerializer(movies, many=True)  
  
    return Response(serializer.data)
```

Then we reduce this mixins methos

```
class ReviewDetails(mixins.RetrieveModelMixin, generics.GenericAPIView):

    queryset = Review.objects.all()

    serializer_class = ReviewSerializer

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

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

Then we reduce all this by generic class base view

```
class ReviewDetails(generics.RetrieveUpdateDestroyAPIView): #get put and
delete review

    queryset=Review.objects.all()

    serializer_class = ReviewSerializer
```

URL configuration

```
#we get all reviews of particular stream platform
```

```
path('stream/review', ReviewList.as_view(), name='review_list'),
```

#now we want to get specific review for specific movie id

```
class ReviewList(generics.ListCreateAPIView):

    queryset=Review.objects.all() # here we are getting all reviews

    #but we need specific stream platform reviews so we create get
queryset method

    serializer_class = ReviewSerializer
```

```

def get_queryset(self):

    pk=self.kwargs.get('pk')

    return Review.objects.filter(watchlist=pk) #coz se set related name
as watchlist in serializer

```

Now if need to add review so need to enter the id number of movie

Content:

```

{
  "rating": 3,
  "description": "good 3",
  "active": true,|
  "watchlist": 1
}

```

But we don't want this we want just put review without write id

```

class ReviewList(generics.ListCreateAPIView):

```

Allow: GET, POST, HEAD, OPTIONS

```

class ReviewList(generics.ListAPIView):

```

Allow: GET, HEAD, OPTIONS

We remove post (create) method

Now we create new method for creating new review for specific movie with it's PK

```

class ReviewCreate(generics.CreateAPIView):

```

```

    serializer_class = ReviewSerializer

```

```

    def perform_create(self, serializer):

```

```

        pk=self.kwargs.get('pk')

```

```
movie=Watchlist.objects.get(pk=pk)

serializer.save(watchlist=movie)
```

```
path('stream/<int:pk>/review-create',ReviewCreate.as_view(),name='review_create'),

#we create a review for particular stream platform
```

```
path('stream/<int:pk>/review-create',ReviewCreate.as_view(),
name='review_create'),#we create a review for particular stream platform

    path('stream/<int:pk>/review',ReviewList.as_view(),
name='review_list'),#we get all reviews of particular stream platform

    path('stream/review/<int:pk>',ReviewDetails.as_view(),
name='review_detail'),#we get particular review of particular stream
platform
```

View set and router

When ever we want to get complete list and particular details We create different class but with the view set we can combined two classes.

urls.py

```
router=DefaultRouter()

router.register('stream', StreamPlatformVS, basename='streamplatform')

urlpatterns = [

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

views.py

We merge `StreamPlatformAV` and `StreamPlatformDetailAV`

```
class StreamPlatformVS(viewsets.ViewSet):

    def list(self, request):

        queryset=StreamPlatform.objects.all()

        serializer=StreamPlatformSerializer(queryset, many=True)

        return Response(serializer.data)

    def retrieve(self, request, pk=None):

        queryset=StreamPlatform.objects.all()

        watchlist=get_object_or_404(queryset,pk=pk)

        serializer=StreamPlatformSerializer(watchlist)

        return Response(serializer.data)

    def create(self, request):

        serializer=StreamPlatformSerializer(data=request.data)

        if serializer.is_valid():

            serializer.save()

            return Response(serializer.data)

        else:

            return Response(serializer.errors)
```

```

def destroy(self, request, pk):

    platform=StreamPlatform.objects.get(pk=pk)

    platform.delete()

    return Response(status=status.HTTP_204_NO_CONTENT)


def update(self, request, pk):

    platform=StreamPlatform.objects.get(pk=pk)

    serializer=StreamPlatformSerializer(platform,data=request.data)

    if serializer.is_valid():

        serializer.save()

        return Response(serializer.data)

    else:

        return
Response(serializer.errors,status=status.HTTP_400_BAD_REQUEST)

```

By the model view-set we reduce this code into too small

```

class StreamPlatformVS(viewsets.ModelViewSet):

    queryset=StreamPlatform.objects.all()

    serializer_class = StreamPlatformSerializer

```

It have all get,post,put,delete inbuilt

Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS

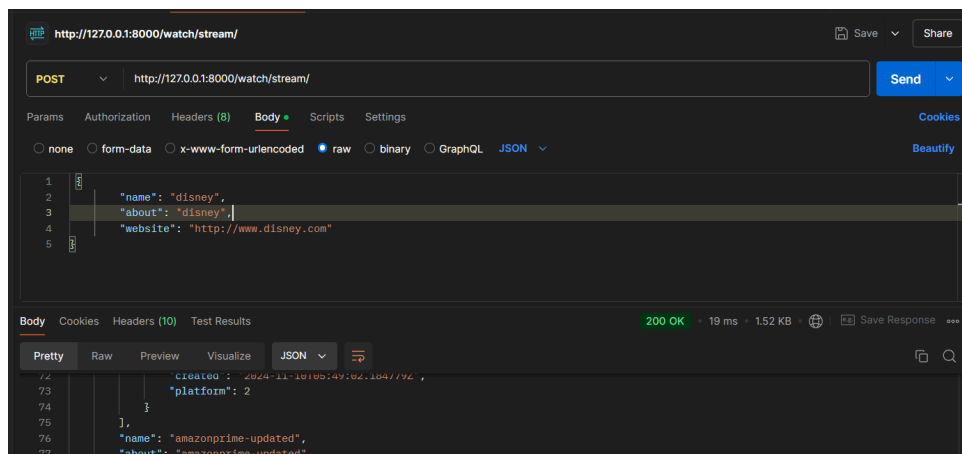

```
class StreamPlatformVS(viewsets.ReadOnlyModelViewSet):
```

Allow: GET, HEAD, OPTIONS

POSTMAN:

Currently we are using browsable API but in future we will work with authentication or may be javascript so we may use Postman.

Postman is the third party software that help us to use API.



It is simple just type url and there is header in which we can see what types of operation we can do , now if we need to CRUD the data we go to body and select body and JSON response and do stuff on data.

->Now we add functionality that one user can submit only one review for particular movie

For this we have to create relation between our django user model and review class

```
review_user=models.ForeignKey(User,on_delete=models.CASCADE,related_name='review_user')#one user can have many reviews
```

Now when we create review we have to check that curr user reviewed the this movie

```
user=self.request.user

review_queryset=Review.objects.filter(watchlist=movie,review_user=user)

    if review_queryset.exists():

        raise ValidationError("You have already reviewed this movie.")
```

Now we want to show the user name that had review the movie so we have to add user name in our serializer.py

```
review_user=serializers.StringRelatedField(read_only=True) # here we are
using StringRelatedField which is used for showing related fields in our
serializer
```

Now we are building our project FR.

Permission and Authentication

Is_staff user is true => he can access admin panel but can't change until the permission given by super user

to make this we have to do is_superuser=True,
is_active false means He can't login

Permissions help us to add restrictions for different access actions

We can add permissions(restrictions) to all using settings.py or we can add object level restrictions in views.py for particular action

In setting.py

After write this we can access any page if and only we are logged in .

This will apply to all urls

```
REST_FRAMEWORK = {  
  
    'DEFAULT_PERMISSION_CLASSES': [  
  
        'rest_framework.permissions.IsAuthenticated',  
  
    ]  
}
```

Object level permission :

Here we give permission in views.py

Class base view:

```
permission_classes = [IsAuthenticated] #only authenticated users can see  
reviews
```

For function base view

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

```
class ReviewDetails(generics.RetrieveUpdateDestroyAPIView): #get put and  
delete review  
  
    queryset=Review.objects.all()  
  
    serializer_class = ReviewSerializer  
  
    permission_classes = [IsAuthenticatedOrReadOnly] #only authenticated  
users can see reviews
```

Here if logged in so we can do all things like `allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS`

But if log out so we can only read content

```
allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
```

Custom Permission :

If user is admin so he can edit it and other users can only read.

Now we create one file name permissions.py in this we write all permissions

```
class AdminOrReadOnly(permissions.IsAdminUser):

    def has_permission(self, request, view):

        if request.method in permissions.SAFE_METHODS: #here SAFE_METHODS
are GET, HEAD, OPTIONS, TRACE

            return True

        else:

            return bool(request.user and request.user.is_staff) #check if
user is admin or staff

class ReviewUserOrReadOnly(permissions.BasePermission):

    def has_object_permission(self, request, view, obj):

        if request.method in permissions.SAFE_METHODS: #check permission
for read only requests

            return True

        else:

            return obj.review_user == request.user #check permission for
writr request and if user is the same as review user so he can update or
delete the review
```

```
permission_classes = [ReviewUserOrReadOnly] #only authenticated users  
can see reviews
```

Now we add 2 new fields , 1)total number of reviews for movie and 2)avg rating

In Watchlist class

```
avg_rating=models.FloatField(default=0.0)  
  
total_ratings=models.PositiveIntegerField(default=0)
```

When ever we create new review so we need to update this two field

So in views.py we have to write some logic

In ReviewCreate class

```
if movie.total_ratings==0:  
  
    movie.average_rating=serializer.validated_data['rating']  
else:  
  
    movie.average_rating=(movie.average_rating+serializer.validated_data  
['rating'])  
  
    movie.total_ratings=movie.total_ratings+1  
  
    movie.save()
```

Authentication:

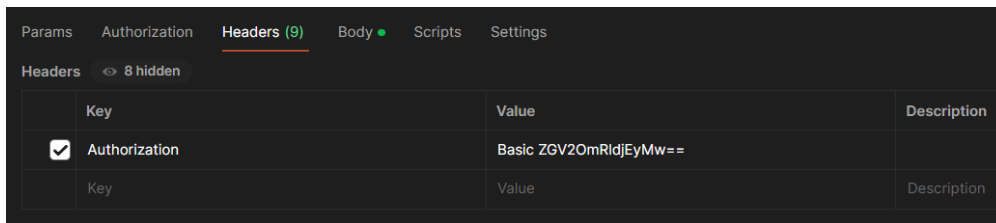
BasicAuthentication, TokenAuthentication, session authentication,
RemoteUserAuthentication

Imp once is JWT authentication and token authentication

Basic Authentication :

settings.py

```
REST_FRAMEWORK = {  
  
    'DEFAULT_AUTHENTICATION_CLASSES': [  
  
        'rest_framework.authentication.BasicAuthentication',  
  
    ]  
  
}  
  
#it apply to all endpoints/classes
```



Params	Authorization	Headers (9)	Body	Scripts	Settings
Headers 8 hidden					
	Key	Value	Description		
<input checked="" type="checkbox"/>	Authorization	Basic ZGV2OmRldjEyMw==			
	Key	Value	Description		

We have to encode the user name and password in base64 formate

Dev:dev123 =>ZGV2OmRldjEyMw==

Then enter into value field

Token Authentication :

Here We create a token for every user whenever he register/login we carry this token

For generating this token we have write some in setting.py

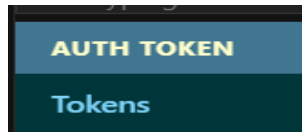
```
REST_FRAMEWORK = {  
  
    'DEFAULT_AUTHENTICATION_CLASSES': [  
  
        'rest_framework.authentication.TokenAuthentication',  
  
    ]  
  
}
```

```
],
}

In INSTALLED_APPS write

'rest_framework.authtoken',
```

Now we can see the Token module in admin site



Now we have to create a token for every user ,currently we are doing this manually but in future we automate this process

<input checked="" type="checkbox"/>	Authorization	Token fb34d6fc1c3bb8a177d069927fcd2...
-------------------------------------	---------------	--

Currently we are manually copy token from admin site

Now we will send a request through from postmen and get a token

Three case : 1)login,2)register,3)logout

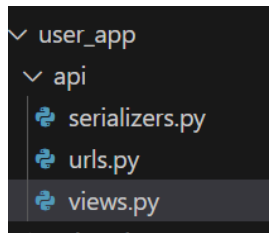
We login using password and username and send this to a login link and we get token from this

When we logout out token get destroy.

Now For login we create another app

We make different app bcoz we can keep all thing separate regarding account system like registration ,logout , login.Their serializer model,urls,views

So we create new app name user_app in this we create three files



We need to include url in our main app (watchmate)

```
urlpatterns = [

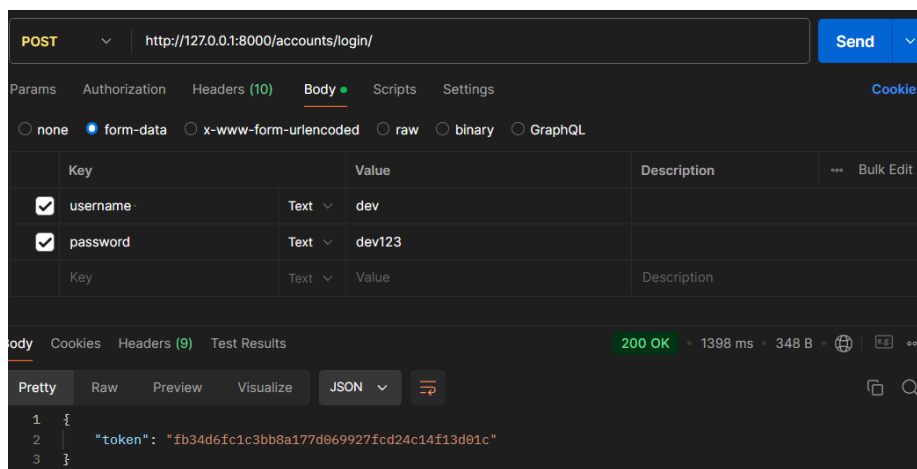
    path('admin/', admin.site.urls),

    path('watch/', include('watchlist_app.api.urls')),

    path('accounts/', include('user_app.api.urls')),

]
```

Now we send post request through from postman to get token



Now we have to carry this token to access reviews

For login we don't need to create model here coz we have user model in our waitlist_app. So we don't have to write anything in our models.py of user_app and we don't have to create login function coz we have inbuilt function for login name : obtain_auth_token so we just direct the process from urls.py


```
urlpatterns = [path('login/', obtain_auth_token, name='login'),]
```

Now we have to work for the registration process :

For this we have to modify our serializer.py we have map all things
username,email,password

```
class RegistrationSerializers(serializers.ModelSerializer):
    password2=serializers.CharField(style={'input_type':'password'},write_only=True)#here write only means user can send information about password but can not read.

    class Meta:

        model = User

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

        extra_kwargs = {'password': {'write_only': True}}#here password is inbuilt field so we have to manually set it readonly

    def save(self):

        password=self.validated_data['password']

        password2=self.validated_data['password2']

        if password != password2:

            raise serializers.ValidationError({'error':'password and password2 should be the same'})

        if
User.objects.filter(email=self.validated_data['email']).exists():

            raise serializers.ValidationError({'error':'email already exists'})
account=User(email=self.validated_data['email'],username=self.validated_data['username'],)
```

```
account.set_password(password)

account.save()

return account
```

Here we have to check two conditions 1)pass1 and pass2 are same and email is unique

Then we have to write function in views.py

```
@api_view(['POST'])

def registration_view(request):

    if request.method == 'POST':

        serializer=RegistrationSerialzers(data=request.POST)

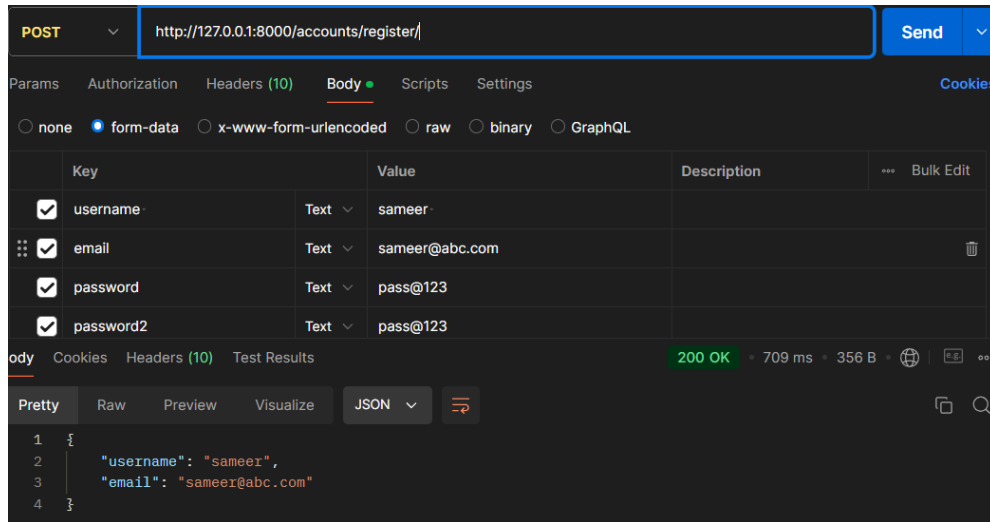
        if serializer.is_valid():

            serializer.save()

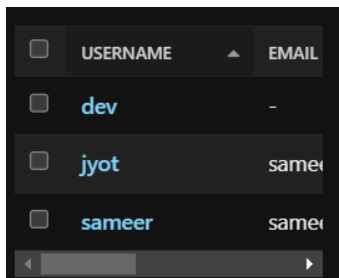
            return Response(serializer.data) #user name and email is send
```

Now we just make URL

```
path('register/', registration_view, name='register'),
```



We created our user



But for token we have to go to login and url and get token, but we don't want this we want that after registration we get our token (don't need to login for token)

So we return created token in form of response

So for this we update our registration view

```
@api_view(['POST'])

def registration_view(request):

    if request.method == 'POST':

        serializer=RegistrationSerialzers(data=request.POST)

        data={}

        if serializer.is_valid():
```

```

        account=serializer.save() #when we call serializer twice it
gives error

        data['response']="Registration successfull"

        data['username']=account.username

        data['email']=account.email

        # Generate token

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

        data['token']=token

        #serializer.save() #when we call serializer twice it gives
error

    else:

        data=serializer.errors

    return Response(data) #user name and email is send

```

We generate token and send it in form of data={}

Now the user and its token both generate on the admin site

Now someone logout so we need to delete this token ,

For logout

```

@api_view(['POST'])

def logout_view(request):

    if request.method == 'POST':

        request.user.auth_token.delete()

    return Response(status=status.HTTP_200_OK)

```

And the token will delete

Here we are creating IBMD clone so there should be only admin can add movie and user can see and review the movies and also admin can edit any type of review

And we also add some filtering , pagination and throttling.

Flow:

Login as Admin (can add data to watchlist(movies) and streaming platform)

Register as User then login and add review

Now we test project with postman

JWT Authentication

Json web token: it is use when we are using python-django rest framework as backend

And javascript along with any framework in frontend (angular,react) and we want authentication with JWT

```
pip install djangorestframework-simplejwt
```

settings.py

```
'DEFAULT_AUTHENTICATION_CLASSES': [  
    'rest_framework.authentication.JWTAuthentication',  
],
```

In jwt token is store in database

In Jwt when we login there are pair of token generate

1)access token(short term token - 5 min)

2)refresh token(long term token - 24 hours)

This token is not stored in our database. When our access token is expire so we use refresh token to regenerate token.

```
#jwt authentication

    path('api/token/', TokenObtainPairView.as_view(),
name='token_obtain_pair'),

    path('api/token/refresh/', TokenRefreshView.as_view(),
name='token_refresh'),
```

Algorithm

HS256

Encoded

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwiaWibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```

Decoded

HEADER:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

Header,payload,Signature

Now we send post request from postman to api/token/ url it return us access and refresh token

```
{
  "refresh":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2t1b190eXB1IjoicmVmcVzaCIsImV4cCI6MTczMjEyMjUxNCwiaWF0IjoxNzMyMDM2MTE0LCJqdGkiOiJiNTlmMGViMTM1YWY0NWRL0WE0OWUxNmFmNDE2NzMyYyIsInVzZXJfaWQoIjF9.xopza3q5XO9OkizmhrFJQSJf9ukn6uwbVmmNKqvX_Ac",

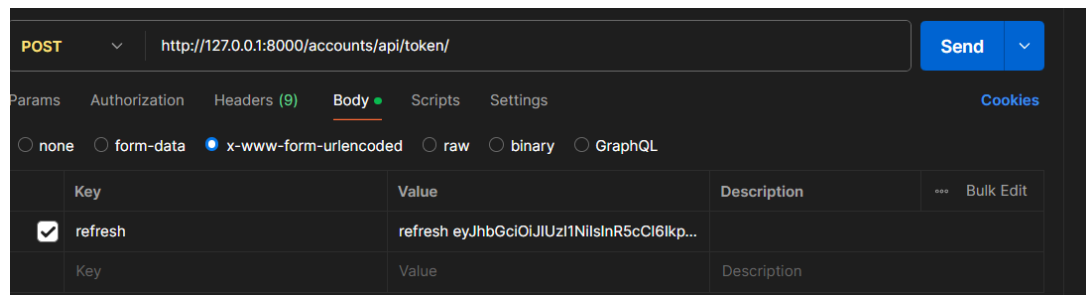
  "access":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2t1b190eXB1IjoicmVmcVzaCIsImV4cCI6MTczMjEyMjUxNCwiaWF0IjoxNzMyMDM2MTE0LCJqdGkiOiJiNTlmMGViMTM1YWY0NWRL0WE0OWUxNmFmNDE2NzMyYyIsInVzZXJfaWQoIjF9.xopza3q5XO9OkizmhrFJQSJf9ukn6uwbVmmNKqvX_Ac"
```

```
}
```

Now we send get request from postman to access the review

Bearer <access_token> : 'api/token/'

refresh <refresh_token> 'api/token/refresh/'



Now we get our new access token now we can do all works with this access token

Please watch video again to understand.

Throttling:

Throttling is like permissions. We can make restrictions to user base on the number of user access.

There are two types of throttling

- 1) `AnonRateThrottle`: not registered on the website
- 2) `UserRateThrottle`: who have registered on the website

Now we can set the number of users base on our requirement

In `setting.py`

```
'DEFAULT_THROTTLE_CLASSES': [  
    'rest_framework.throttling.AnonRateThrottle',  
    'rest_framework.throttling.UserRateThrottle'  
],
```

```
'DEFAULT_THROTTLE_RATES': {  
  
    'anon': '1/day',  
  
    'user': '3/day'  
  
}
```

Here we can send total 3 requests per day this is default.

But we want to custom this for each class

For this we add

```
throttle_classes=[UserRateThrottle , AnonRateThrottle]
```

In our classes now the restrictions will only apply to all class

Like here the restriction is on review details but if we want to access watch list so there should be no restrictions.

Now we don't want to give same number that can access per day. Like in this we define anon and user numbers and whenever we use this any class this number of request apply to that class. But if we want to modify user's request base on the different class.

So we create custom class and scope for this

So we create new file name throttling.py and

```
class ReviewCreateThrottle(UserRateThrottle):  
  
    scope='review_create'  
  
class ReviewListThrottle(UserRateThrottle):  
  
    scope='review_list'
```


Now we add this in settings.py

```
'DEFAULT_THROTTLE_RATES': {  
  
    'anon': '1/day',  
  
    'user': '3/day',  
  
    'review_create': '1/day',  
  
    'review_list': '10/day',  
  
}
```

Now we just add restriction to classes in views.py

```
class ReviewList(generics.ListAPIView:  
  
    #if we don;t want to make throttling.py file  
  
    serializer_class = ReviewSerializer  
  
    throttle_classes=[ReviewListThrottle,AnonRateThrottle]
```

```
class ReviewCreate(generics.CreateAPIView):  
  
    serializer_class = ReviewSerializer  
  
    permission_classes = [IsAuthenticated] #only authenticated users can  
create review  
  
    throttle_classes=[ReviewCreateThrottle]
```

Now we will test this by sending request from postman

`http://127.0.0.1:8000/watch/5/review/` : we can access 10/day

`http://127.0.0.1:8000/watch/5/review-create/` we can create 2 review/day

If we don't want to make another file so we use ScopeRateThrottle

In this we direct define the scope in class

```
class ReviewDetails(generics.RetrieveUpdateDestroyAPIView):#get put and delete review

    queryset=Review.objects.all()

    serializer_class = ReviewSerializer

    permission_classes = [IsReviewUserOrReadOnly] #only authenticated users can see reviews

    throttle_classes=[ScopedRateThrottle]

    throttle_scope='review_detail'
```

Like this

Filtering:

Three things we can do inside filtering : filter,search,order content

Filtering review base on user name

We make UserReview class in views.py

Filtering against the URL

```
class UserReview(generics.ListAPIView):

    serializer_class = ReviewSerializer

    #this will give current user

    def get_queryset(self):
```

```
username=self.kwargs.get('username')

return Review.objects.filter(review_user=username)#this will
filter review by current username
```

```
path('review/<int:pk>/',ReviewDetails.as_view(),
name='review_detail'),#filtering base on id

path('review/<str:username>/',UserReview.as_view(),
name='user_review_detail'),#filtering base on username
```

Here we got error :Field 'id' expected a number but got 'dev'.

Coz review_user is foreign key to User model so we have to modify method

```
return Review.objects.filter(review_user__username=username)
```

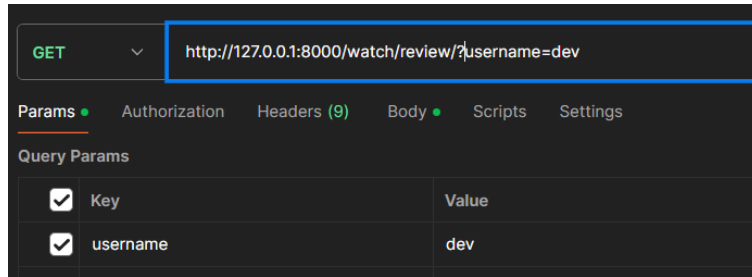
This will go to first foreign key review_user then it goes to username to match it

Whenever we want to search base on some value that is foreign key so we have to do __ . other wise not

Filtering against the Query Parameter:

<http://example.com/api/purchases?username=denvercoder9>, and filter the queryset only if the `username` parameter is included in the URL:

```
username=self.request.query_params.get('username',None)#insted of direcly
mapping the value we map the paramaters
```



We also have to change url

```
path('review/', UserReview.as_view(),  
name='user_review_detail'), #filtering base on username parameters
```

Generic Filters:

We have to install django filter

pip install django-filter

Add in settings.py in installed_apps

Add this in your class

```
filter_backends = [DjangoFilterBackend]  
  
filterset_fields = ['review_user__username', 'active']
```

Now we can filter base on review user and active parameter

http://127.0.0.1:8000/watch/4/review/?review_user__username=rutu

<http://127.0.0.1:8000/watch/4/review/?&active=true>

http://127.0.1:8000/watch/4/review/?review_user__username=rutu&active=true

We can also create same for watchlist

```
class WatchList(generics.ListAPIView):

    queryset = Watchlist.objects.all()

    serializer_class = WatchListSerializer

    filter_backends = [DjangoFilterBackend]

    filterset_fields = ['title', 'platform__name']
```

http://127.0.0.1:8000/watch/list2/?title=blind spot&platform__name=Netflix

When we use filter we should have perfect matching value
So it is good when we are searching for rating or some number wise filter

Search Filter

When we don't have perfect matching value

```
filter_backends = [SearchFilter]

    search_fields = ['title', 'platform__name']

#for exact match user = before fields

search_fields = ['title', 'platform__name']

#for ordering

    filter_backends = [OrderingFilter]

ordering_fields = ['average_rating']
```

<http://127.0.0.1:8000/watch/list2/?search=still>

We have to show the name of stream platform while displaying list

```
platform = serializers.CharField(source='platform.name') #we have to
display the platform name platform --> name
```

Pagination:

When we try to extract too many information so we divide them into multiple pages.

the process of breaking large amounts of data into multiple web pages.

Ex: on page 1 > 10 elements, page 2 > 10 elements.

Pagination can improve performance by reducing the amount of content loaded on a single page, which can decrease page load time.

We can do global declaration of pagination that make impact on all pages by define it into settings.py

```
'DEFAULT_PAGINATION_CLASS':  
'rest_framework.pagination.LimitOffsetPagination',  
  
    'PAGE_SIZE': 100
```

Now we make one file name by pagination.py in that we define our pagination and import this in class in which we have to implement pagination

Pagenuumber pagination:

In pagination.py

```
class WatchListPagination(PagenuumberPagination):  
  
    page_size = 3
```

In watchlistGV class add

```
pagination_class = WatchListPagination
```

```
{
  "count": 8,
  "next": "http://127.0.0.1:8000/watch/list2/?page=2",
  "previous": null,
  "results": [
    {
      "id": 4,
      "platform": "primevideo",
      "title": "blind spot",
      "storyline": "good",
      "active": true,
      "average_rating": 4.5,
      "total_ratings": 2,
      "created": "2024-11-19T13:00:32.711576Z"
    }
  ]
}
```

<http://127.0.0.1:8000/watch/list2/?page=2>

```
{
  "count": 8,
  "next": "http://127.0.0.1:8000/watch/list2/?page=3",
  "previous": "http://127.0.0.1:8000/watch/list2/",
  "results": [
    {
      "id": 7,
      "platform": "stream2",
      "title": "pyaar",
      "storyline": "dec1",
      "active": true,
      "average_rating": 0.0,
      "total_ratings": 0,
      "created": "2024-11-08T06:24:30.650947Z"
    }
  ]
}
```

In all this pages 3 elements will display

We can modify this page_size (parameter name) read documentation

When we want to modify our pagesize at runtime so we can

Redefine size

```
class WatchListPagination(PageNumberPagination):

    page_size = 3

    #customize pagination settings

    #page_size_query_param='size'

    #max_page_size =10
```

<http://127.0.0.1:8000/watch/list2/?size=5>

LimitOffset pagination:

Here are two main keyword

Limit: decide how many items we want

Offset: decide from where you want

Limit is basically page size.

Offset simply means the number of records you want to skip before selecting records.

Ex: we have 50 elements, limit=5, offset=12 this means i want 5 items after 12.

In pagination.py

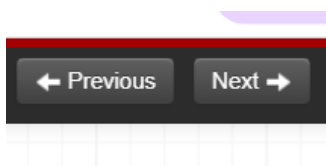
```
class WatchListLOpagination(LimitOffsetPagination):  
  
    default_limit = 3
```

Use this in views.py

```
pagination_class = WatchListLOpagination
```

<http://127.0.0.1:8000/watch/list2/?limit=3&offset=1>

Cursor pagination:



We have option like this.

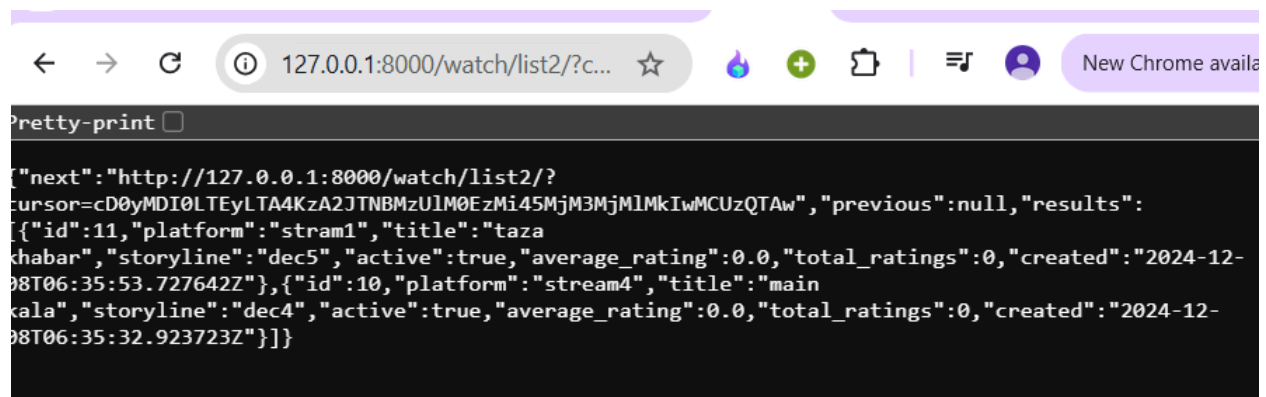


All this pagination works with viewset or generic view classes.

All this we are done it is on the browsable API.

If we any website we get response in json formate , so for this we need to add below setting.

```
'DEFAULT_RENDERER_CLASSES': (
    'rest_framework.renderers.JSONRenderer',
)
```



Now our page should look like this.

Our project is completed Now we are doing testing.

Automated API testing.

Testing is important because in future we will modify our code so because of this code our current/previous code should not affected.

For testing we write test case in test.py. Every app module has this test.py file.

We can also create test folder in that we can create multiple tests file according to our requirements.here we don't create this folder. And name of file should start with test like test_anyword.

For the sending post,put,get requests django use client class. This have all these properties.

When we creates methos in class this should start with test.

Registration test:

```
class RegisterTestCase(APITestCase):

    def test_register(self):

        data={

            "username": "testcase",

            "email": "testcase@example.com",

            "password": "NewPassword@123",

            "password2": "NewPassword@123"

        }

        #response =self.client.post('register/',data)

        response = self.client.post(reverse('register'), data) #when we
use reverse we just need to pass name

        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
```

Now we have to run: python manage.py test on terminal

(menv) C:\Users\Devansh shrimali\Desktop\drf-project\watchmate>python manage

.py test

Found 1 test(s).

Creating test database for alias 'default'...

System check identified no issues (0 silenced).

.

Ran 1 test in 0.598s

OK

Destroying test database for alias 'default'...

For the login/logout test we should have user so for this we have to create a fake user.

For create any fake data we use the setUp method.

For login, the same is registered.

```
class LoginLogoutTestCase(APITestCase):

    def setUp(self):

        self.user = User.objects.create_user(username='testuser',
password='NewPassword@123')

    def test_login(self):

        data={

            "username": "testuser",

            "password": "NewPassword@123"

        }

        response=self.client.post(reverse('login'), data)

        self.assertEqual(response.status_code, status.HTTP_200_OK)
```

Now we create test case for our watchlist app.

For add movie we first have stream platform and for add review we first have movie.

TDD: test driven development

Normally we are first develop models then views, urls then we are doing testing. This call function then test(development first case)

In TDD we are creating our models then we test it like we write test case then we write our views.