rest framework

rest architecture

api = application programming interface -> we can run multiple app using one database. Example facebook whatsapp insta server down if the organisation use one database but they are interconnected each other through api. if server down then 3 application will not work properly

convert database in api

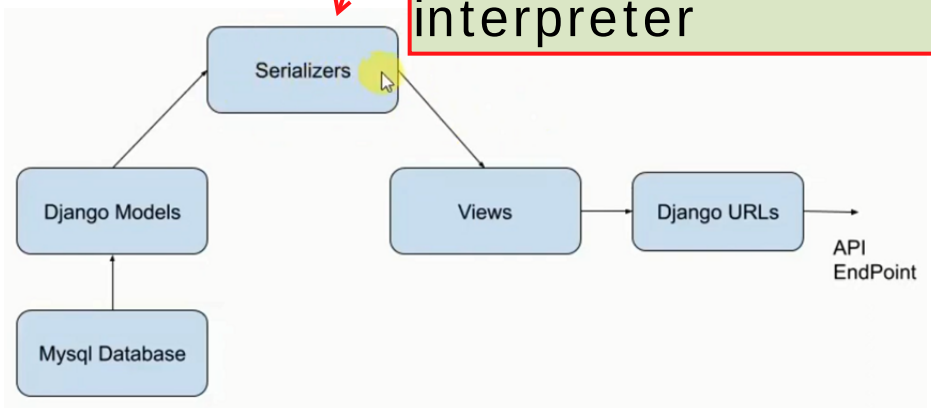
Rest = Representational state transfer

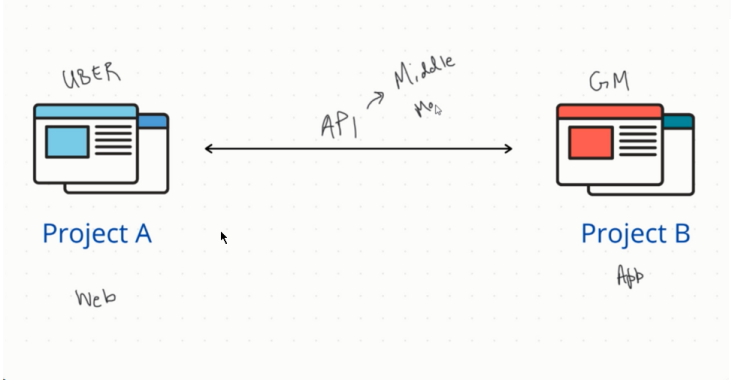
is a type of architecture (make of such kinds of algo) thats makes the data in one kind of standard data form JSON

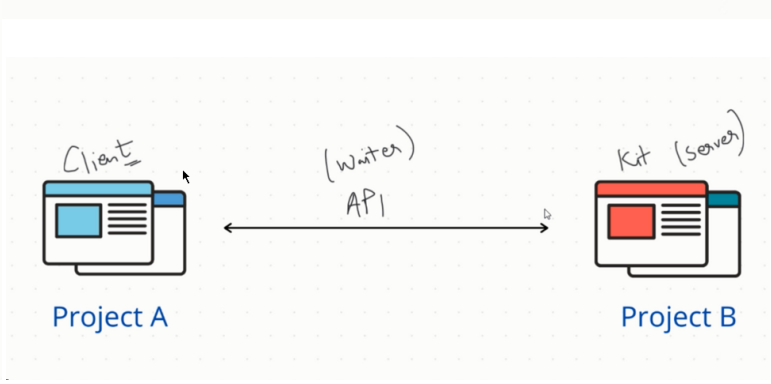
take data from model and convert in json.example chineese interpreter

json=Java Script Object Notation

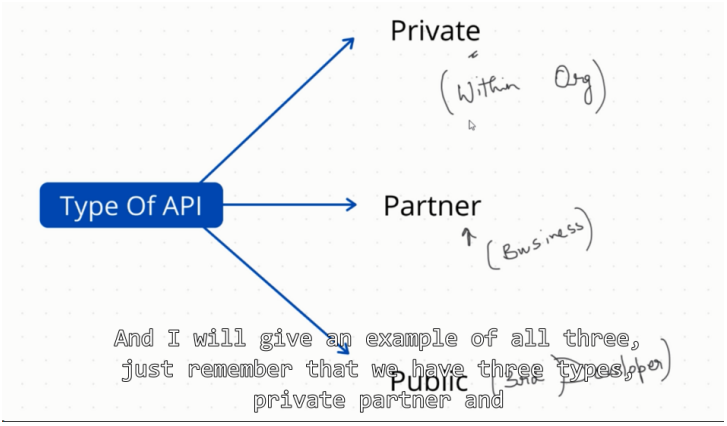
user request-> view
->model->database
->view

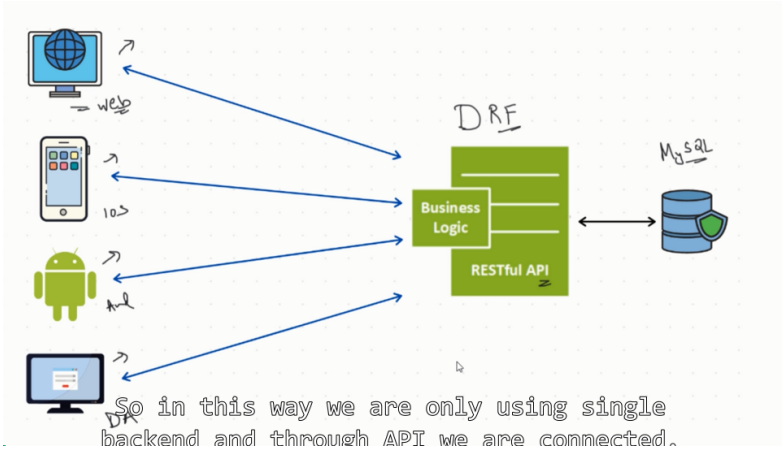


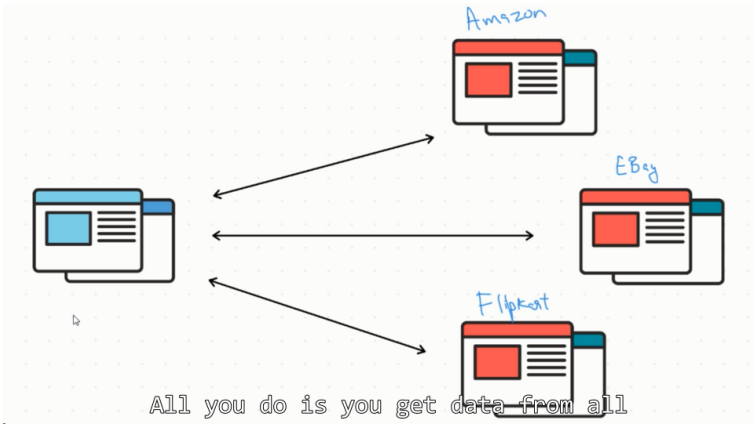


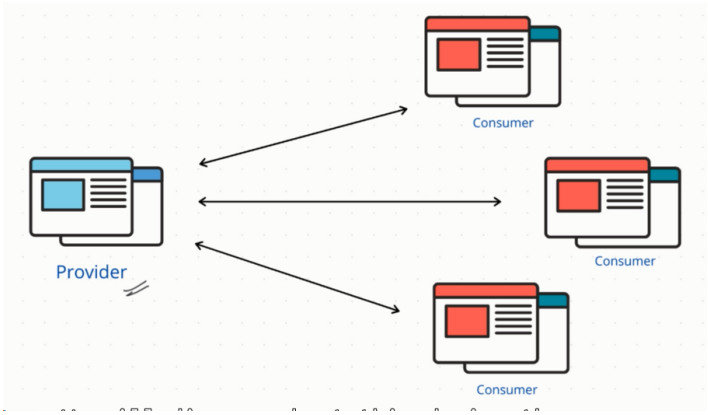


api act as a middle man between client and server









Understanding URL

https://www.api.movielist.com/movies/
https://www.api.movielist.com/movies/list/

https://www.api.movielist.com/movies/127/
https://www.api.movielist.com/movies/127/reviews/
https://www.api.movielist.com/movies/127/reviews/?limit=20

https://www.api.movielist.com/account/login/
https://www.api.movielist.com/account/register/

It can be Spider-Man, Superman or any other movie.

Understanding URL

https://www.api.movielist.com/movies/ List
https://www.api.movielist.com/movies/list/

Ind
https://www.api.movielist.com/movies/127/
https://www.api.movielist.com/movies/127/reviews/
https://www.api.movielist.com/movies/127/reviews/?limit=20

Base URL
End Point

https://www.api.movielist.com/account/login/
https://www.api.movielist.com/account/register/

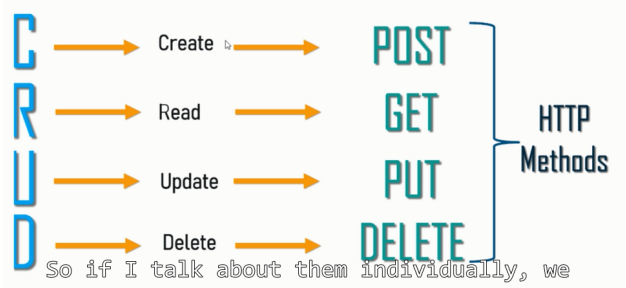
And then this remaining part, this part is known as End Point.

API REST → Rest API

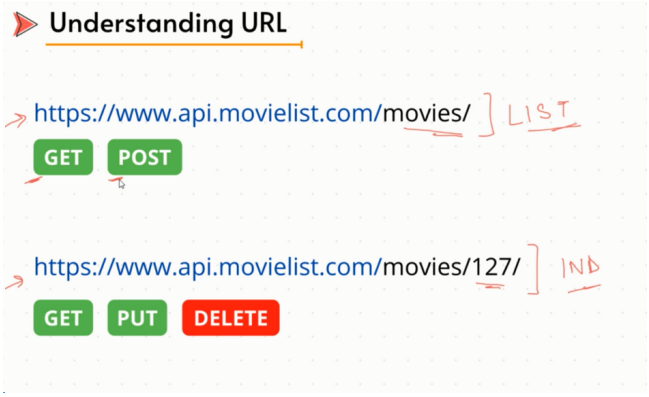
(status code)

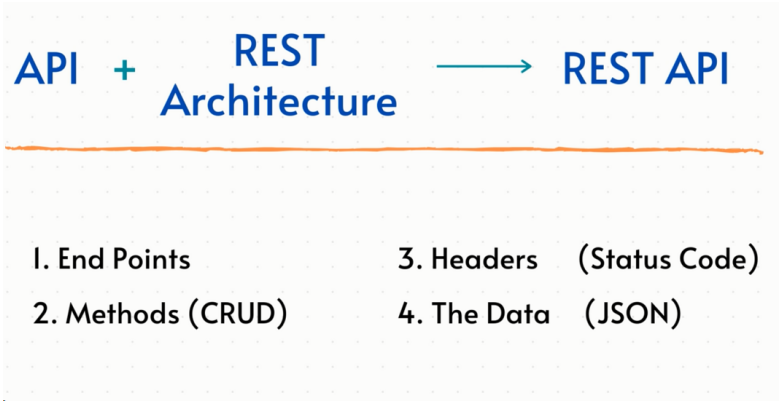
① endpoint
② method
③ headers
④ the data

json data



So if I talk about them individually, we
:l them Krod. create. retrieve. update and delet



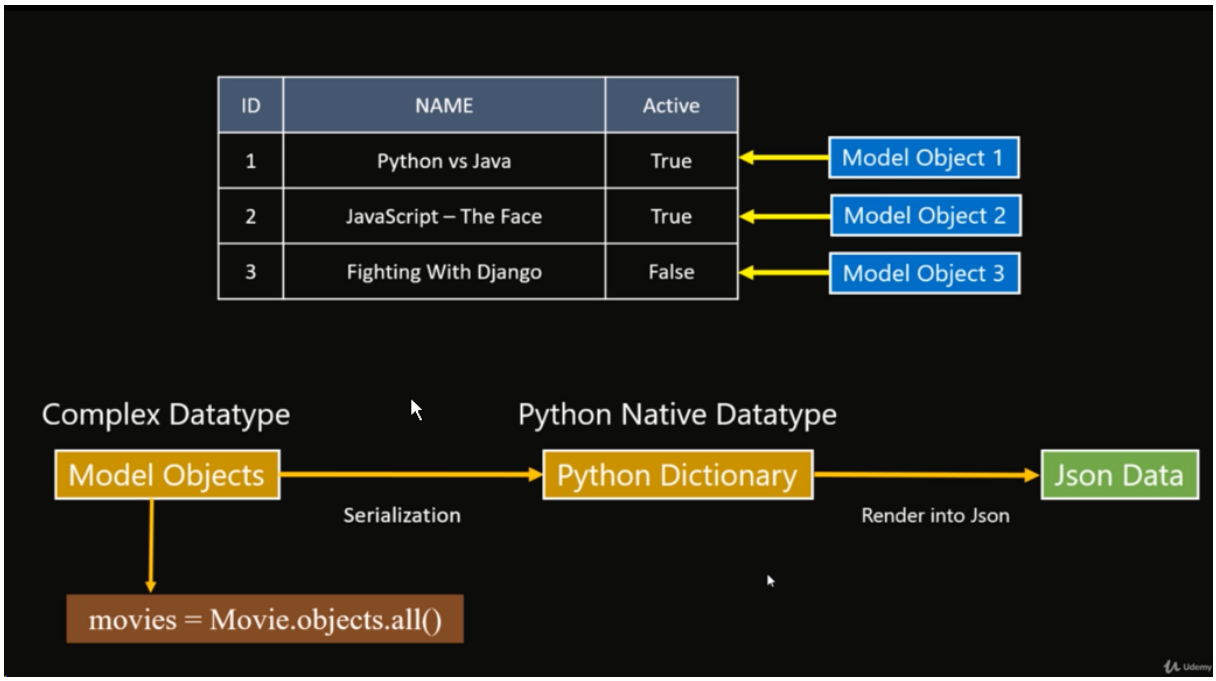


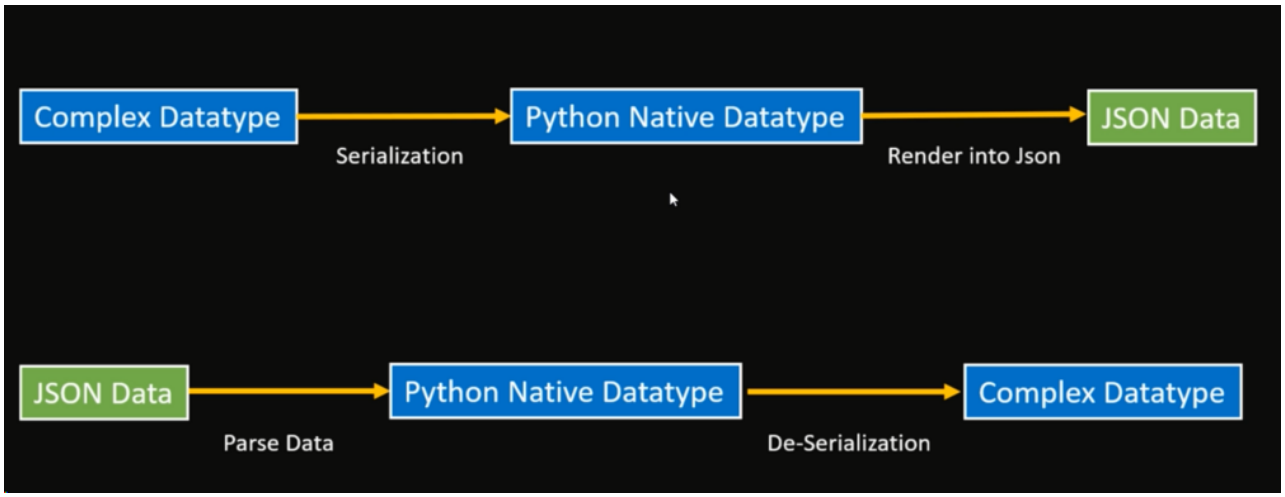

```
def movie(request):
    movies = Movie.objects.all()
    data = {
        'movies': list(movies.values())
    }
    return JsonResponse(data)


    def movie_details(requst,pk):
        movie=Movie.objects.get(id=pk)
        data={
            'name'=movie.name,
            'description'=movie.description
    }
    return JsonResponse(data)
```

queryset -> python dictionary

python dictionary ->json response

serializations in DRF





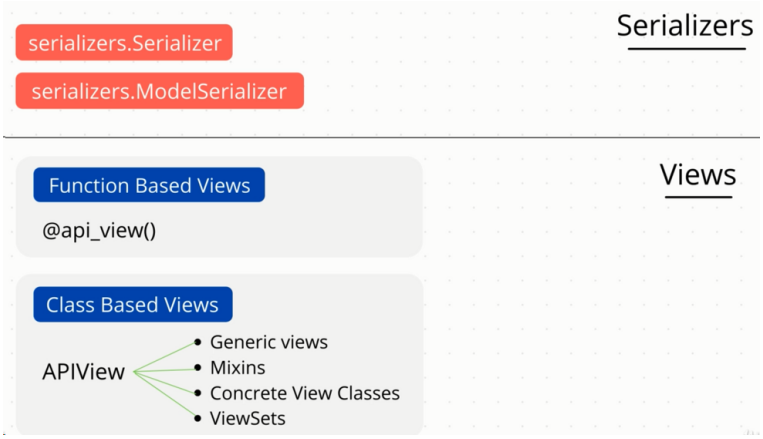
Type Of Serializers

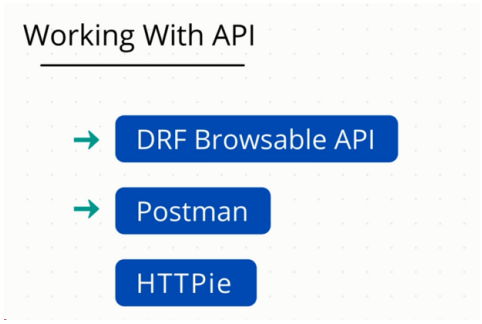
Type Of Views

Working With API

serializers.Serializer

serializers.ModelSerializer





serializers:

```
class MovieSerializer(serializers.Serializer):
    id=serializers.IntergerField(read_only=True)
    name=serializers.CharField()
```

views:

```
@api_view(['GET','POST'])
def movie_list(request):
    movies=Movie.objects.all()
    serializer=MovieSerializer(movies, many=True)
    return Response(serializer.data)

@api_view(['GET','POST'])
def movie_details(request,pk):
    movie=Movie.objects.get(id=pk)
    serializer=MovieSerializer(movie)
    return Response(serializer.data)
```

views:

```
@api_view(['GET','POST'])
def movie_list(request):
    if request.method == 'GET':
        movies=Movie.objects.all()
        serializer=MovieSerializer(movies)
        return Response(serializer.data)

    if request.method =='POST':
        serializer=MovieSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        else:
            return Response(serializer.errors)
```

```
@api_view(['GET','PUT','DELETE'])
def movie_details(request,pk):
   if request.method =="GET":
      movie=Movie.objects.get(id=pk)
      serializer=MovieSerializer(movie)
      return Response(serializer.data)

   if request.method=='PUT'
     movie=Movie.objects.get(id=pk)
     serializer=MovieSerializer(movie,data=request.data)
     if serializer.is_valid():
         serializer.save()
         return Response(serializer.data)
     else:
        w  return Response(serializer.errors)

    if request.method=="DELETE":
       movie=Movie.objects.get(id=pk)
       movie.delete()
       return Response(status=status.HTTP__204__No
Content)
```

status Code

Informational - 1xx

This class of status code indicates a provisional response. There are no 1xx status codes used in REST framework by default.

```
HTTP_100_CONTINUE
HTTP_101_SWITCHING_PROTOCOLS
```

Successful - 2xx

This class of status code indicates that the client's request was successfully received, understood, and accepted.

```
HTTP_200_OK
HTTP_201_CREATED
HTTP_202_ACCEPTED
HTTP_203_NON_AUTHORITATIVE_INFORMATION
HTTP_204_NO_CONTENT
HTTP_205_RESET_CONTENT
HTTP_206_PARTIAL_CONTENT
HTTP_207_MULTI_STATUS
HTTP_208_ALREADY_REPORTED
HTTP_226_IM_USED
```

Redirection - 3xx

This class of status code indicates that further action needs to be taken by the user agent in order to fulfill the request.

```
HTTP_300_MULTIPLE_CHOICES
HTTP_301_MOVED_PERMANENTLY
HTTP_302_FOUND
HTTP_303_SEE_OTHER
HTTP_304_NOT_MODIFIED
HTTP_305_USE_PROXY
HTTP_306_RESERVED
HTTP_307_TEMPORARY_REDIRECT
HTTP_308_PERMANENT_REDIRECT
```

Client Error - 4xx

The 4xx class of status code is intended for cases in which the client seems to have erred. Except when responding to a HEAD request, the server SHOULD include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition.

HTTP_400_BAD_REQUEST
HTTP_401_UNAUTHORIZED
HTTP_402_PAYMENT_REQUIRED
HTTP_403_FORBIDDEN
HTTP_404_NOT_FOUND
HTTP_405_METHOD_NOT_ALLOWED
HTTP_406_NOT_ACCEPTABLE
HTTP_407_PROXY_AUTHENTICATION_REQUIRED
HTTP_408_REQUEST_TIMEOUT
HTTP_409_CONFLICT
HTTP_410_GONE
HTTP_411_LENGTH_REQUIRED
HTTP_412_PRECONDITION_FAILED
HTTP_413_REQUEST_ENTITY_TOO_LARGE
HTTP_414_REQUEST_URI_TOO_LONG
HTTP_415_UNSUPPORTED_MEDIA_TYPE
HTTP_416_REQUESTED_RANGE_NOT_SATISFIABLE
HTTP_417_EXPECTATION_FAILED
HTTP_422_UNPROCESSABLE_ENTITY
HTTP_423_LOCKED
HTTP_424_FAILED_DEPENDENCY
HTTP_426_UPGRADE_REQUIRED
HTTP_428_PRECONDITION_REQUIRED
HTTP_429_TOO_MANY_REQUESTS
HTTP_431_REQUEST_HEADER_FIELDS_TOO_LARGE
HTTP_451_UNAVAILABLE_FOR_LEGAL_REASONS

Server Error - 5xx

Response status codes beginning with the digit "5" indicate cases in which the server is aware that it has erred or is incapable of performing the request. Except when responding to a HEAD request, the server SHOULD include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition.

HTTP_500_INTERNAL_SERVER_ERROR
HTTP_501_NOT_IMPLEMENTED
HTTP_502_BAD_GATEWAY
HTTP_503_SERVICE_UNAVAILABLE
HTTP_504_GATEWAY_TIMEOUT
HTTP_505_HTTP_VERSION_NOT_SUPPORTED
HTTP_506_VARIANT_ALSO_NEGOTIATES
HTTP_507_INSUFFICIENT_STORAGE
HTTP_508_LOOP_DETECTED
HTTP_509_BANDWIDTH_LIMIT_EXCEEDED
HTTP_510_NOT_EXTENDED
HTTP_511_NETWORK_AUTHENTICATION_REQUIRED

Helper functions

The following helper functions are available for identifying the category of the response code.

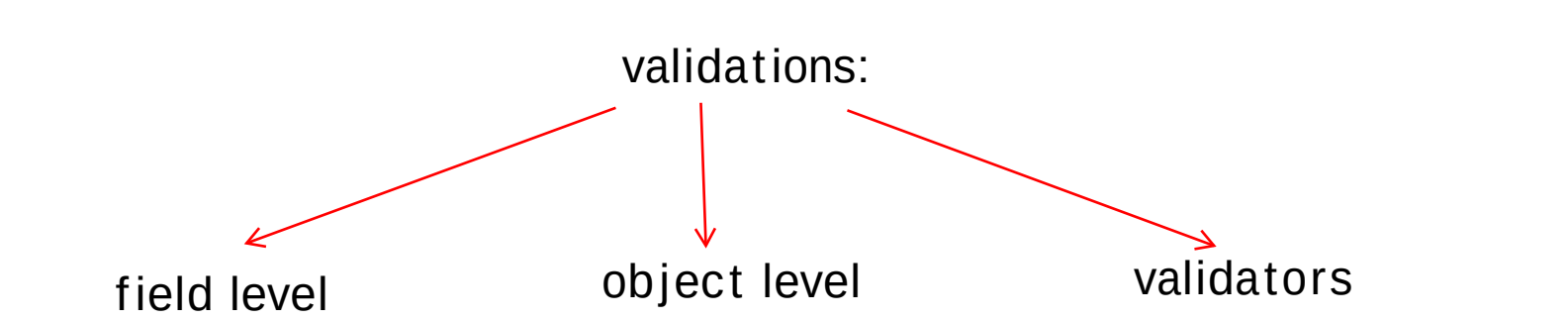
```
is_informational()  # 1xx
is_success()        # 2xx
is_redirect()       # 3xx
is_client_error()   # 4xx
is_server_error()   # 5xx
```

```
@api_view(['GET','POST'])
def movie_list(request):
    if request.method=="get":
        movies=Movie.objects.all()
        serilizer=MovieSerializer(movies,many=True)
        return Response(serializer.data)

     if request.method=="POST":
        serializer=MovieSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        else:
             return Response(status=status.HTTP__404)
```

```
@api_view('GET','PUT','DELETE')
def movie_details(request,pk):
    if request.method=='GET':
        try:
            movie=Movie.objects.get(id=pk)
        except Movie.DoesNotExit:
            content={'errors':'Movie not found'}
            return Response(content,status=status.)

        serializer=MovieSerializer(movie)
        return response(serializer.data)

    if request.method=="PUT":
        movie=Movie.objects.get(id=pk)
        serializer=MovieSerializer(movie,data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        else:
            return Response(status=status.HTTP __204__)

       if request.method=="DELETE":
           movie=Movie.objects.get(id=pk)
           movie.Delete()
           return Response(status=status.HTTP__data not fo)
```

```
class MovieListAV(APIView):
    def get(self, request):
        movies = Movie.objects.all()
        serializer = MovieSerializer(movies, many=True)
        return Response(serializer.data)

    def post(self, request):
        serializer = MovieSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
status=status.HTTP_201_CREATED)
        else:
            return Response(serializer.errors)
```

```
class MovieDetailsAV(APIView):
    def get(self, request, pk):
        try:
            movie = Movie.objects.get(id=pk)
        except Movie.DoesNotExist:
            content = {
                'error': 'Movie not found'
            }
            return Response(content,
status=status.HTTP_400_BAD_REQUEST)
        serializer = MovieSerializer(movie)
        print(serializer)
        return Response(serializer.data)

    def put(self, request, pk):
        movie = Movie.objects.get(id=pk)
        serializer = MovieSerializer(movie, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
status=status.HTTP_202_ACCEPTED)
        else:
            return Response(status=status.HTTP_400_BAD_REQUEST)

    def delete(self, request, pk):
        Movie.objects.get(id=pk).delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```




```
def validate_name(self, value):
    if len(value) < 2:
        raise serializers.ValidationError
('Name can not be less than 2')
    else:
        return value
```

```
def validate(self, data):
    if data['name'] == data['description']:
        raise serializers.ValidationError
('Name and Description can not same')
    else:
        return data
```

```
def length_name(value):
    if len(value) < 2:
        raise serializers.ValidationError('name can not
less than 2')
    else:
        return value


class MovieSerializer(serializers.Serializer):
    id = serializers.IntegerField(read_only=True)
    name = serializers.CharField(validators=[length_name])
```

Serializer Fields and Core Arguments:

```
class MovieSerializer(serializers.Serializer):
    id = serializers.IntegerField(read_only=True)
    name = serializers.CharField(validators=[length_name])
```

Model Serializer:

```
class MovieSerializer(serializers.ModelSerializer):
    class Meta:
        model = Movie
        fields = "__all__"
        #  fields = ['id', 'name', 'description']
        # exclude = ['name']
```

```
def validate_name(self, value):
    if len(value) < 2:
        raise serializers.ValidationError('Name can not be less than 2')
    else:
        return value

def validate(self, data):
    if data['name'] == data['description']:
        raise serializers.ValidationError('Name and Description can not same')
    else:
        return data
```

custom Serializer:

```
class MovieSerializer(serializers.ModelSerializer):
    length_name=serializers.SerializerMethodField()
    class Meta:
        model=Movie
        fields="__all__"
    def get_length_name(self,object):
        return len(object.name)
```

Three kinds of connections:
one to one =
one to many =
many to many=

One To One Relations:

```
class Place(models.Model):
    location=models.CharFields(max_length=70)

class Restaurant(models.Model):
    place=models.OneToOne(Place,on_delete=models.CASCADE)
```

One To Many:

```
class Reporter(models.Model):
    name=models.CharField(max_length=80)

class Article(models.Model):
    reporter=models.Foreignkey(Reporter)
```

Many To Many

```
class Publication(models.Model):
    name=models.CharField(max_length=90)

class Article(models.Model):
    publication=models.ManyToMany(Publication)
```

Nested Serializer->

```
class StreamingPlatformSerializer
(serializers.ModelSerializer):
    watchlist = WatchListSerializer(many=True,
read_only=True)

    class Meta:
        model = StreamingPlatform
        fields = "__all__"
```

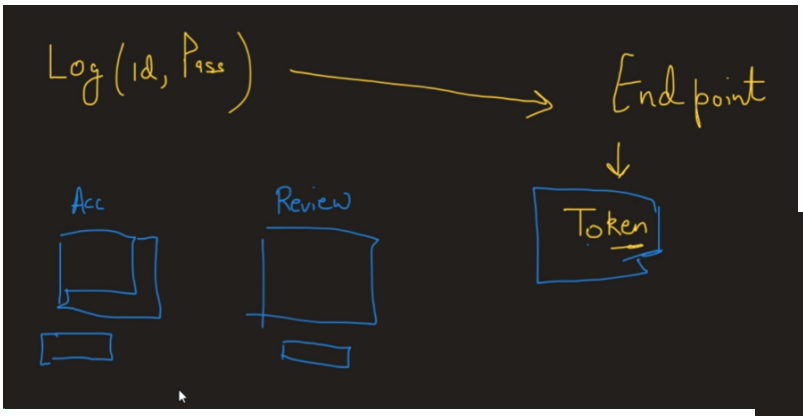
```
class StreamPlatformSerializer(serializers.ModelSerializer):
    # watchlist = WatchListSerializer(many=True, read_only=True)
    watchlist = serializers.HyperlinkedRelatedField(
        many=True,
        read_only=True,
        view_name='movie-detail'
    )
```

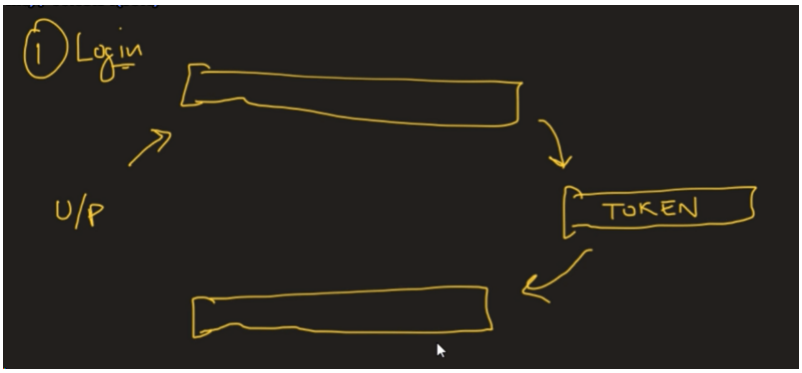
```
class StreamPlatFormDetailsAV(APIView):
    def get(self, request, pk):
        try:
            stream_platform = StreamingPlatform.objects.get(id=pk)
        except StreamingPlatform.DoesNotExist:
            return Response(status=status.HTTP_204_NO_CONTENT)
        serializer = StreamingPlatformSerializer(stream_platform)
        return Response(serializer.data)

    def put(self, request, pk):
        stream_platform = StreamingPlatform.objects.get(id=pk)
        serializer = StreamingPlatformSerializer(stream_platform,data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        else:
            return Response(status=status.HTTP_400_BAD_REQUEST)

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


class WatchListListAV(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)
        else:
            return Response(serializer.errors)


class WatchListDetailsAV(APIView):
    def get(self, request, pk):
        try:
            movie = WatchList.objects.get(id=pk)
        except WatchList.DoesNotExist:
            content = {
                'error': 'not found'
            }
            return Response(content, status=status.HTTP_400_BAD_REQUEST)
        serializer = WatchListSerializer(movie)
        print(serializer)
        return Response(serializer.data)

    def put(self, request, pk):
        movie = WatchList.objects.get(id=pk)
        serializer = WatchListSerializer(movie, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_202_ACCEPTED)
        else:
            return Response(status=status.HTTP_400_BAD_REQUEST)

    def delete(self, request, pk):
        WatchList.objects.get(id=pk).delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

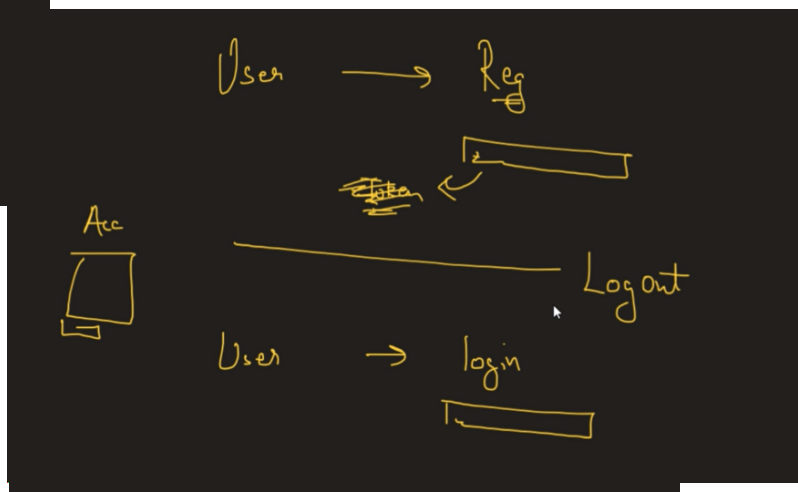
```
class ReviewList(mixins.ListModelMixin,
                 mixins.CreateModelMixin,
                 generics.GenericAPIView):
    queryset = Review.objects.all()
    serializer_class = ReviewSerializer

    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,
                   mixins.UpdateModelMixin,
                   mixins.DestroyModelMixin,
                   generics.GenericAPIView):
    queryset = Review.objects.all()
    serializer_class = ReviewSerializer

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

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

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

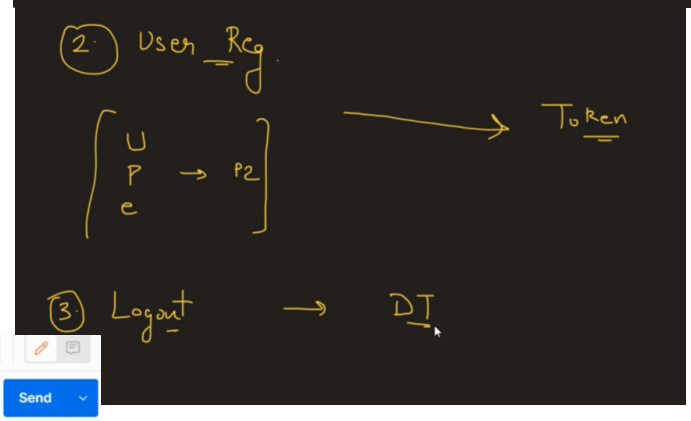
Authentication manages user is logged in or not or valid user

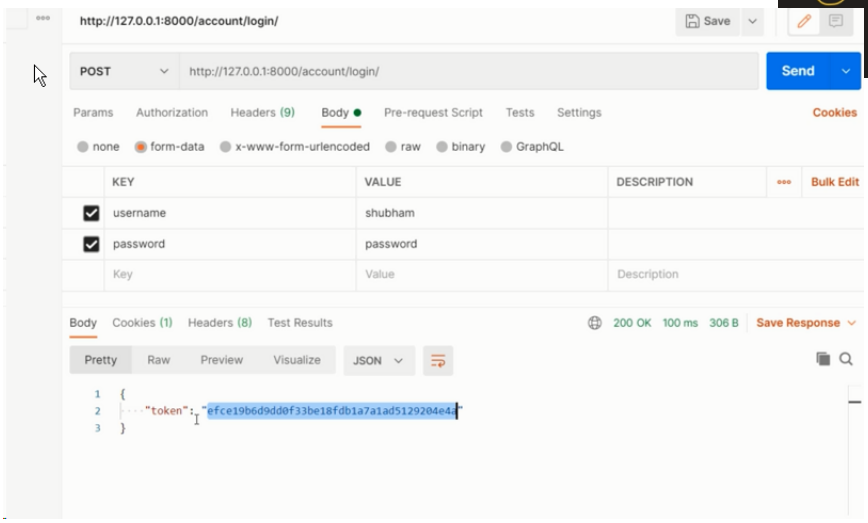
Permission : user can access this url or not (any kinds of restriction)

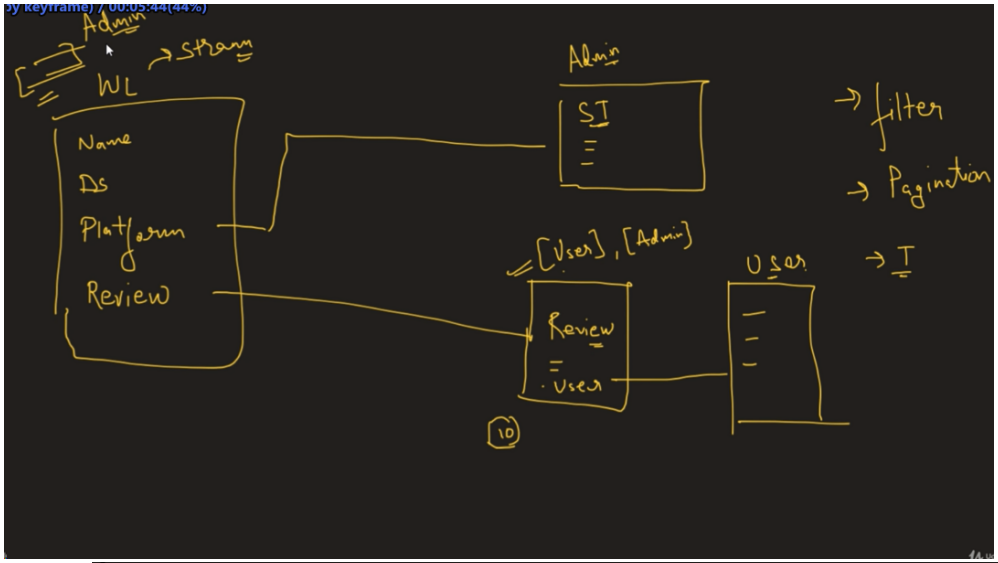


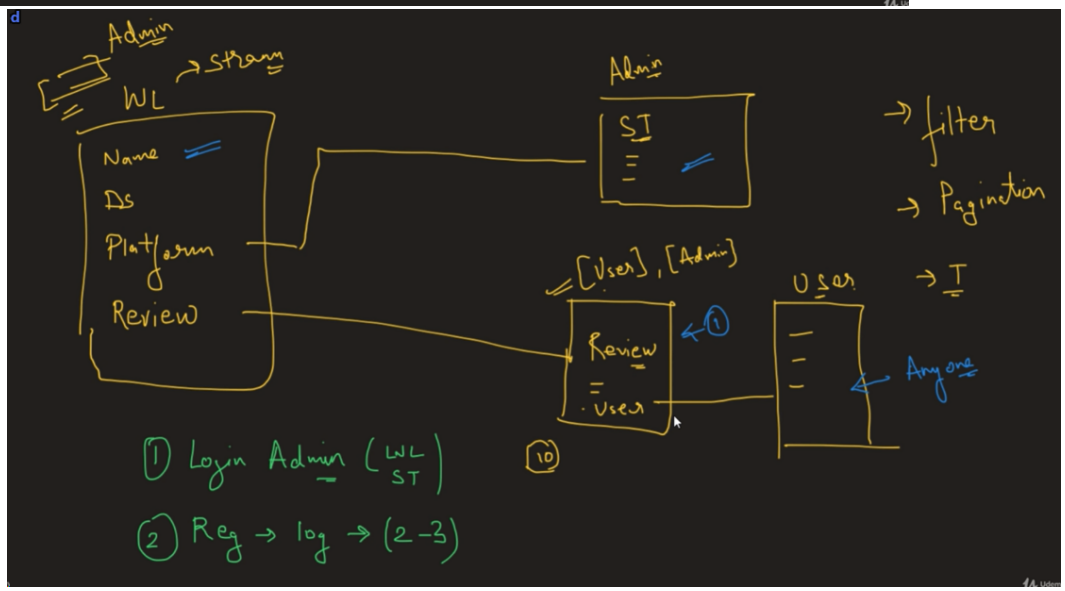




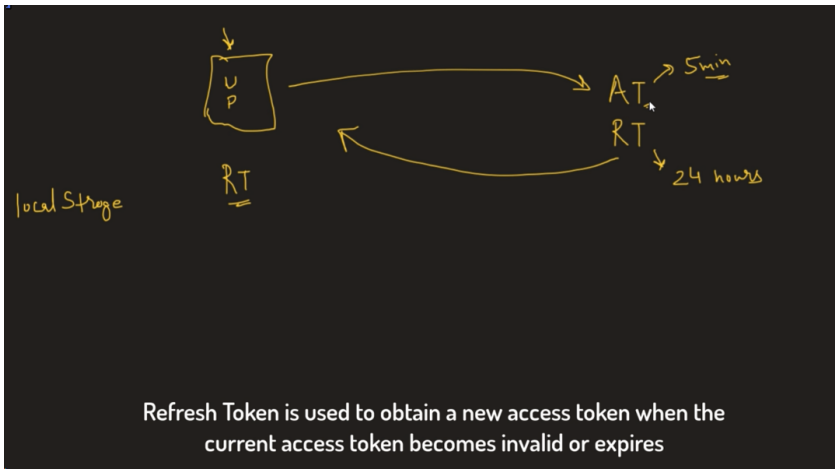








JWT : JOSON web Token



advantage: not rdepending on database

duration:
active token :5-15min
refresh token : up to 14 days

disadvantage:
caching information storing it for 5-15 min

we can only revoke access by deleting user

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

Filtering:
filter
oredr
search

Pagination:

server need not to load all the elements

PageNumberPagination

LimitOffsetPagination

CursorPagination

Tests.py:

```
class Stream(models.Model):
    name = models.CharField(max_length=150)
    about = models.CharField(max_length=150)
    website_link=models.UrlField(max_length=150)

    def __str__(self):
        return self.name

class WatchList(models.Model):
    title = models.CharField(max_length=150)
    story_line = models.CharField(max_length=150)
    platform= models.ForeignKey(Stream,on_delete=models.CASCADE,related_name="
watchlist")
    active=models.BooleanField(default=False)
    avg_rating=models.FloatField(default=0)
    num_rating=models.IntegerField(default=0)
    created=models.DateTimeField(auto_now_add=True)

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

This is the behaviour to adopt when the referenced object is deleted. It is not specific to Django; this is an SQL standard. Although Django has its own implementation on top of SQL. (1)

There are seven possible actions to take when such event occurs:

CASCADE: When the referenced object is deleted, also delete the objects that have references to it (when you remove a blog post for instance, you might want to delete comments as well). SQL equivalent: CASCADE.
PROTECT: Forbid the deletion of the referenced object. To delete it you will have to delete all objects that reference it manually. SQL equivalent: RESTRICT.
RESTRICT: (introduced in Django 3.1) Similar behavior as PROTECT that matches SQL's RESTRICT more accurately. (See django documentation example)
SET_NULL: Set the reference to NULL (requires the field to be nullable). For instance, when you delete a User, you might want to keep the comments he posted on blog posts, but say it was posted by an anonymous (or deleted) user. SQL equivalent: SET NULL.
SET_DEFAULT: Set the default value. SQL equivalent: SET DEFAULT.
SET(...): Set a given value. This one is not part of the SQL standard and is entirely handled by Django.
DO_NOTHING: Probably a very bad idea since this would create integrity issues in your database (referencing an object that actually doesn't exist). SQL equivalent: NO ACTION. (2)
Source: Django documentation

See also the documentation of PostgreSQL for instance.

In most cases, CASCADE is the expected behaviour, but for every ForeignKey, you should always ask yourself what is the expected behaviour in this situation. PROTECT and SET_NULL are often useful. Setting CASCADE where it should not, can potentially delete all of your database in cascade, by simply deleting a single user.

Additional note to clarify cascade direction

It's funny to notice that the direction of the CASCADE action is not clear to many people. Actually, it's funny to notice that only the CASCADE action is not clear. I understand the cascade behavior might be confusing, however you must think that it is the same direction as any other action. Thus, if you feel that CASCADE direction is not clear to you, it actually means that on_delete behavior is not clear to you.

In your database, a foreign key is basically represented by an integer field which value is the primary key of the foreign object. Let's say you have an entry comment_A, which has a foreign key to an entry article_B. If you delete the entry comment_A, everything is fine. article_B used to live without comment_A and don't bother if it's deleted. However, if you delete article_B, then comment_A panics! It never lived without article_B and needs it, and it's part of its attributes (article=article_B, but what is article_B???). This is where on_delete steps in, to determine how to resolve this integrity error, either by saying:

"No! Please! Don't! I can't live without you!" (which is said PROTECT or RESTRICT in Django/SQL)
"All right, if I'm not yours, then I'm nobody's" (which is said SET_NULL)
"Good bye world, I can't live without article_B" and commit suicide (this is the CASCADE behavior).
"It's OK, I've got spare lover, and I'll reference article_C from now" (SET_DEFAULT, or even SET(...)).
"I can't face reality, and I'll keep calling your name even if that's the only thing left to me!" (DO_NOTHING)
I hope it makes cascade direction clearer. :)

```
class Review(models.Model):
    review_user=models.ForeignKey(User,on_delete=models.CASCADE)
    rating=models.PositiveIntegerField(validators=[MinValueValidator(1),MaxValueValidator
(5)])
    description=models.CharField(max_length=200,null=True)
    watchlist=models.ForeignKey(WatchList,on_delete=models.CASCADE)
    active=models.BooleanField(default=True)
    created=models.DateTimeField(auto_now_add=True)
    updated=models.DateTimeField(auto_now=True)
```

Problems are encountered if auto_now and auto_now_add are confused. How do auto_now or auto_now_add work?

auto_now : Time will be created every time when use models.save() or models.create() but it doesn't work if you use query.update(), it only updates some data but it does not update date automatically

auto_now_add : Time will be created only the first time when using models.save() or models.create()

How should they be used?

auto_now_add should be used with created_date and auto_now should used with updated_date

```
created_date = models.DateTimeField(auto_now_add = True)
updated_date = models.DateTimeField(auto_now = True)
```

Serializers :

```
class ReviewSerializer(serializer.ModelSerializer):
        class Meta:
            model=Review
            fields="__all__"



class WatchListSerializers(serializers.ModelSerializer):
        reviews=ReviewSerializer(many=True,read_only=True)
        class Meta:
            model=WatchList
            fields="__all__"


class StreamSerializer(serializers.ModelSerializer):
      watchlist=WatchListSerializer(many=True,read_only=True)
      class Meta:
          model=Stream
          fields="__all__"
```

view

```
def stream_list(request):
      if request.method=="get":
          stream=Stream.objects.all()
          serializer=StreamSerializer(stream,many=True)
          return Response(serializer.data)

     if request.method=="POST":
        serializer=StreamSerializer(data=request.data)
        if serializer.is_valid():
        serializer.save()
        return Response(serializer.data)
```

```
def stream_details(requset,pk):
    if request.method=="GET":
        try:
            stream=Stream.objects.get(id=pk)
        except Stream.DoesNotExit:
            return Response(status=status.Http bad request)

        serializer=StreamSerializer(stream)
        return Response(serializer.data)

    if request.method=="PUT"
       stream=Stream.objects.get(id=pk)
       serializer=StreamSerializer(stream,data=request.data)
       if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)

     if request.method=="DELETE":
          Stream.objects.get(id=pk).delete()
          return Response(status=status. HTTP no content)
```