Note 1: With ReadOnlyModelViewSet We Can Only Use list, And Retrieve By Field.

****

For creating Nested Routers, We Use: ***pip install drf-nested-routers***

Then We Import It, And Use Its Routers-Classes:

```
from rest_framework_nested import routers

router = routers.DefaultRouter()

router.register('products', views.ProductViewSet)
router.register('collections', views.CollectionViewSet)

products_router = routers.NestedDefaultRouter(router,
'products', lookup='product')

products_router.register('reviews', views.ReviewViewSet,
basename='product-reviews')
urlpatterns = router.urls + products_router.urls
```
****

To Avoid Problems of Parent Pk Not Used For Child **OR** Related Objects:

Note: Here We Use Context to Avoid Read the Parent Data from User Input, Then We Override The ***create-Method*** Of ReviewSerializer

```
class ReviewViewSet(ModelViewSet):

    # queryset = Review.objects.all()

    serializer_class = ReviewSerializer

    def get_queryset(self):
        return Review.objects.filter(product_id =
self.kwargs['product_pk'])

    def get_serializer_context(self):
        return { 'product_id': self.kwargs['product_pk'] }
```

****

To Implement Filtering Using Query Params, We Must:

- Override The *get_queryset-Method*
- Use *get-Method of Dict*
- Set ***basename in urls.py*** For Target Class

```
class ProductViewSet(ModelViewSet):

    serializer_class = ProductSerializer

    def get_queryset(self):
        queryset = Product.objects.all()

        collection_id =
self.request.query_params.get('collection_id')

        if collection_id is not None:
            queryset =
queryset.filter(collection_id=collection_id)

        return queryset

router = routers.DefaultRouter()

router.register('products', views.ProductViewSet,
basename='products')
```
****

To Use Generic Filtering In Django: We Use: *pip install django-filter*

Then We Register It in installed APPs, AS: ***django_filters*** And Before The ***rest_framework***

****

Then To Implement The Filters For Products:

Note: In This Way We Can Return ***queryset-Attribute***

```
from django_filters.rest_framework import
DjangoFilterBackend

class ProductViewSet(ModelViewSet):
    queryset = Product.objects.all()

    serializer_class = ProductSerializer

    filter_backends = [DjangoFilterBackend]

    filterset_fields = ['collection_id']
```
****

Note: We Must Be Careful When Using django-filter, Because We May Have Duplicate Query

****

To Implement Custom Filter Using django-filter:

```
from django_filters import FilterSet

from .models import Product

class ProductFilter(FilterSet):
    class Meta:
        model = Product
        fields = {
            'collection_id': ['exact'],
            'unit_price': ['lt', 'gt'],
        }
Then In views.py:

class ProductViewSet(ModelViewSet):
    queryset = Product.objects.all()

    serializer_class = ProductSerializer

    filter_backends = [DjangoFilterBackend]

    filterset_class = ProductFilter
```
****

To Implement Search Filters, We Use Filters of *rest_framework*:

```
from rest_framework.filters import SearchFilter
```

Then We Add it to Filters Array:

```
filter_backends = [DjangoFilterBackend, SearchFilter]
```

Then We Define The Search Array Fields:

```
search_fields = ['title', 'description']
```

Note 1: The Search Is *Case Insensitive*

Note 2: This Will Use search query Param: *http://localhost:8000/store/products/?search=coffee*

The Complete Implementation:

```
class ProductViewSet(ModelViewSet):
    queryset = Product.objects.all()

    serializer_class = ProductSerializer
    filter_backends = [DjangoFilterBackend, SearchFilter]

    filterset_class = ProductFilter

    search_fields = ['title', 'description']

    def get_serializer_context(self):
        return { 'request': self.request }
```
****

To implement ordering using specific fields:

```
from rest_framework.filters import SearchFilter,
OrderingFilter
```

Then We Add It To Filters Backend Array:

```
filter_backends = [DjangoFilterBackend, SearchFilter,
OrderingFilter]
```

Then We Define The Ordering Fields: `ordering_fields = ['unit_price',`
```
'last_update']
```

The Complete Implementation:

```
class ProductViewSet(ModelViewSet):
    queryset = Product.objects.all()

    serializer_class = ProductSerializer

    filter_backends = [DjangoFilterBackend, SearchFilter,
OrderingFilter]
```

```
    filterset_class = ProductFilter

    search_fields = ['title', 'description']

    ordering_fields = ['unit_price', 'last_update']

    def get_serializer_context(self):
        return { 'request': self.request }
```
****

Note: To Order Using Multiple Fields: *http://localhost:8000/store/products/?ordering=-unit_price,last_update&search=*

****

To implement the pagination in rest_framework:

```
from rest_framework.pagination import PageNumberPagination
```

Then We Define The Pagination Class Inside The ViewSet:

```
pagination_class = PageNumberPagination
```

Then to define the Page Size, We Go To settings.py Of Main Project:

```
REST_FRAMEWORK = {
    'COERCE_DECIMAL_TO_STRING': False,
    'PAGE_SIZE': 10,
    'DEFAULT_PAGINATION_CLASS':
'rest_framework.pagination.PageNumberPagination'
}
class ProductViewSet(ModelViewSet):
    queryset = Product.objects.all()

    serializer_class = ProductSerializer

    filter_backends = [DjangoFilterBackend, SearchFilter,
OrderingFilter]

    filterset_class = ProductFilter

    search_fields = ['title', 'description']

    ordering_fields = ['unit_price', 'last_update']

    pagination_class = PageNumberPagination # This Can Be
Deleted
```

****

By Using The Default Pagination Class, We Can Delete The ***pagination_class*** From Product ViewSet

****

Note: If We Want to Use Limit and Offset for Pagination, We Have *LimitOffsetPagination-Class*.

****

The Best Way, If We Don't Want To use Pagination for all ViewSets, Is By Implementing Our Paginator Class

And Define Its Settings:

```
from rest_framework.pagination import PageNumberPagination

class DefaultPagination(PageNumberPagination):
    page_size = 10
```

Then We Set The ***pagination_class*** Only For ViewSets, That Needed It:

```
from .pagination import DefaultPagination

class ProductViewSet(ModelViewSet):
    queryset = Product.objects.all()

    serializer_class = ProductSerializer

    filter_backends = [DjangoFilterBackend, SearchFilter,
OrderingFilter]

    filterset_class = ProductFilter

    search_fields = ['title', 'description']

    ordering_fields = ['unit_price', 'last_update']

    pagination_class = DefaultPagination
```
****

```
from uuid import uuid4
```

To use UUID AS ***Id Field***: `id = models.UUIDField(primary_key=True, default=uuid4)`

Note: Here We Don't Call ***uuid4-Function***

****

If Any Problems Happened When Changing Id Field From Bigint-8 To UUID: (From Stackoverflow)

So, to handle this in django, do the following:

1) revert migrations to a working graph

2) add **temp_id = models.UUIDField(default=uuid.uuid4)** to your model, then run **makemigrations**

3) * add **primary_key=True** to the **temp_id** field, then run **makemigrations** again

4) rename the field to **id** (or to whatever you want), then run **makemigrations** a third time

5) push the migrations to the database via **python3 manage.py migrate**

****

Note (To Remember):

```
from rest_framework.mixins import CreateModelMixin,
RetrieveModelMixin

from rest_framework.viewsets import ModelViewSet,
GenericViewSet
```

****

Note: To create simple serializer that can be used instead of full one:

```
class SimpleProductSerialzier(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = [ 'id', 'title', 'unit_price' ]
```

****

```
class CartItemSerializer(serializers.ModelSerializer):

    product = SimpleProductSerialzier()

    total_price =
serializers.SerializerMethodField(method_name='get_total_pri
ce')

    def get_total_price(self, cart_item: CartItem):
        return cart_item.quantity *
cart_item.product.unit_price

    class Meta:
        model = CartItem
        fields= ['id', 'product', 'quantity', 'total_price']
```

****

```
class CartSerializer(serializers.ModelSerializer):

    id = serializers.UUIDField(read_only=True)
    items = CartItemSerializer(many=True, read_only = True)

    total_price =
serializers.SerializerMethodField(method_name='get_total_pri
ce')

    def get_total_price(self, cart: Cart):
        return sum([item.quantity * item.product.unit_price
for item in cart.items.all()])

    class Meta:
        model = Cart
        # Here We Define items Inside The CartItem-Model AS
The Related Name
        fields = [ 'id', 'items', 'total_price']
```

****

Note (To remember): serializer objects has method for validation: **is_valid(...)**

****

```
class AddCartItemSerialzier(serializers.ModelSerializer):
    id = serializers.UUIDField(read_only=True)

    product_id = serializers.IntegerField() # Here we set product_id
                                       # because it is populated only in runtime

    # here will call self.is_valid
    def save(self, **kwargs):
        cart_id = self.context['cart_id']
        product_id = self.validated_data['product_id']
        quantity = self.validated_data['quantity']

        try:
            cart_item = CartItem.objects.get(cart_id = cart_id, product_id =
product_id)
            cart_item.quantity += quantity
            cart_item.save()

            self.instance = cart_item

        except CartItem.DoesNotExist:
            self.instance = CartItem.objects.create(cart_id=cart_id,
**self.validated_data)

        return self.instance

    class Meta:
        model = CartItem
        fields = ['id', 'product_id', 'quantity']
```
****