The Updated Serializer Is:

```
class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = ['id', 'title', 'description', 'slug', 'inventory',
'unit_price', 'price_with_tax', 'collection']

    price_with_tax =
serializers.SerializerMethodField(method_name='calculate_tax')
    def calculate_tax(self, product: Product):
        return product.unit_price * Decimal(1.1)
```

****

Then To Save The Product, We Can Call save-Method Of Serializer:

```
@api_view(['GET', 'POST'])
def product_list(request):
    if request.method == 'GET':
        queryset = Product.objects.select_related('collection').all()
        serializer = ProductSerializer(queryset, many=True, context={ 'request':
request })
        return Response(serializer.data)
    elif request.method == 'POST':
        serializer = ProductSerializer(data = request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response('Ok')
```

****

If We Want To Override The Create Method Of Serializer:

```
def create(self, validated_data):
    product = Product(**validated_data)
    product.other = 1
    product.save()
    return product

def update(self, instance, validated_data):
    return super().update(instance, validated_data)
```

****

If We Want To Override The Update-Method:

```python
def update(self, instance, validated_data):
    instance.unit_price = validated_data.get('unit_price')
    instance.save()
    return instance
```

****

To Update The Product Using PUT, OR PATCH, We Can Use:

```python
@api_view(['GET', 'PUT', 'PATCH'])
def product_detail(request, id):
    product = get_object_or_404(Product, pk=id)

    if request.method == 'GET':
        serializer = ProductSerializer(product, context={ 'request': request })
        return Response(serializer.data)

    elif request.method == 'PUT':
        serializer = ProductSerializer(product, data = request.data);
        serializer.is_valid(raise_exception=True)
        serializer.save()

        return Response(serializer.data)
```

****

To Delete The Product, After We Check All Constraint:

```
@api_view(['GET', 'PUT', 'DELETE'])
def product_detail(request, id):
    product = get_object_or_404(Product, pk=id)

    if request.method == 'GET':
        serializer = ProductSerializer(product, context={ 'request': request })
        return Response(serializer.data)

    elif request.method == 'PUT':
        serializer = ProductSerializer(product, data = request.data);
        serializer.is_valid(raise_exception=True)
        serializer.save()

        return Response(serializer.data)

    elif request.method == 'DELETE':
        if product.orderitems.count() > 0:
            return Response({'error': 'product cannot be deleted'},
status=status.HTTP_405_METHOD_NOT_ALLOWED)

        product.delete()

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

****

Note (From Me): Always Check The Number Of Queries That Are Executed For Each Serializer OR Model, And Use: ***select_related***, ***prefetch_related***, ***annotated***.

****

```python
@api_view(['GET', 'POST'])
def collection_list(request):
    if request.method == 'GET':
        queryset =
Collection.objects.annotate(products_count=Count('product')).all();

        serializer = CollectionSerializer(queryset, many=True)

        return Response(serializer.data)
    elif request.method == 'POST':
        serializer = CollectionSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()

        return Response(serializer.data, status=status.HTTP_201_CREATED)
**************************************************************************

class CollectionSerializer(serializers.ModelSerializer):
    class Meta:
        model = Collection
        fields = ['id', 'title', 'products_count']

    products_count =
serializers.SerializerMethodField(method_name='get_products_count')

    def get_products_count(self, collection: Collection):
        return collection.products_count
**************************************************************************
```

To Create Class Based Views, We Can Use APIView-Class, That Represent The Main Class For Views-Based-Classes.

Note 1: In Class Based Views, Here We Don't Have Many If-Statements.

```
from rest_framework.views import APIView

class ProductList(APIView):
    def get(self, request):
        queryset = Product.objects.select_related('collection').all()
        serializer = ProductSerializer(queryset, many=True, context={ 'request':
request })

        return Response(serializer.data)

    def post(self, request):
        serializer = ProductSerializer(data = request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()

        return Response(serializer.data, status=status.HTTP_201_CREATED)
```

****

To Set The URL Param:

```
class CollectionDetail(APIView):
    def get(self, request, pk: int):
        collection = get_object_or_404(
            Collection.objects.annotate(products_count=Count('products')),
            pk=pk
        )

        serializer = CollectionSerializer(collection)

        return Response(serializer.data)
```

****

Note: To Encapsulate The Logic Of Creating, Listing, Updating, And Deleting We Can Use Mixins.

****

Note: The Best Way To Use Mixins, is That Using Generic Views, That Encapsulate The Works Of Business Layer.

****

To Use Generic Views For List OR Create Objects We Can Use:

Note: We Can Use This Way With Methods If We Have Logic Inside The Methods.

```
from rest_framework.generics import ListCreateAPIView


class ProductList(ListCreateAPIView):

    def get_queryset(self):
        return Product.objects.select_related('collection').all()

    def get_serializer_class(self):
        return ProductSerializer

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

In This Way We Can Use Attributes And Methods For Defining The QuerySet And SerializerClass.

```
class ProductList(ListCreateAPIView):

    queryset = Product.objects.select_related('collection').all()

    serializer_class = ProductSerializer

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

class CollectionSerializer(serializers.ModelSerializer):

    products_count = serializers.IntegerField(read_only=True)

    class Meta:
        model = Collection
        fields = ['id', 'title', 'products_count']
**************************************************************************
```

```
class CollectionDetail(RetrieveUpdateDestroyAPIView):

    queryset =
Collection.objects.annotate(products_count=Count('products')).all()

    serializer_class = CollectionSerializer

    # lookup_field = 'id'
**************************************************************************
```

For using ModelViewSet To Implement The Logic That We Want:

Note: In This Way, We Delete The Duplication In Generic Views

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

    serializer_class = ProductSerializer

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

    def delete(self, request, pk: int):
        product = get_object_or_404(Product, pk=pk)

        if product.orderitems.count() > 0:
            return Response({'error': 'product cannot be deleted'},
status=status.HTTP_405_METHOD_NOT_ALLOWED)

        product.delete()

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

And To Register View Sets Inside The URLs File Of APP:

Note: Here We Can Use Include-Function With *router.urls*.

```
from rest_framework.routers import SimpleRouter

from pprint import pprint


router = SimpleRouter()

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

pprint(router.urls)
urlpatterns = router.urls
**************************************************************************
```

```
(.venv) C:\Tests\Web\Django\Code-With-Mosh>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

[<URLPattern '^products/$' [name='product-list']>,
 <URLPattern '^products/(?P<pk>[^/.]+)/$' [name='product-detail']>,
 <URLPattern '^collections/$' [name='collection-list']>,
 <URLPattern '^collections/(?P<pk>[^/.]+)/$' [name='collection-detail']>]
**************************************************************************
```

By Using DefaultRouter We Have Two Additional Features:

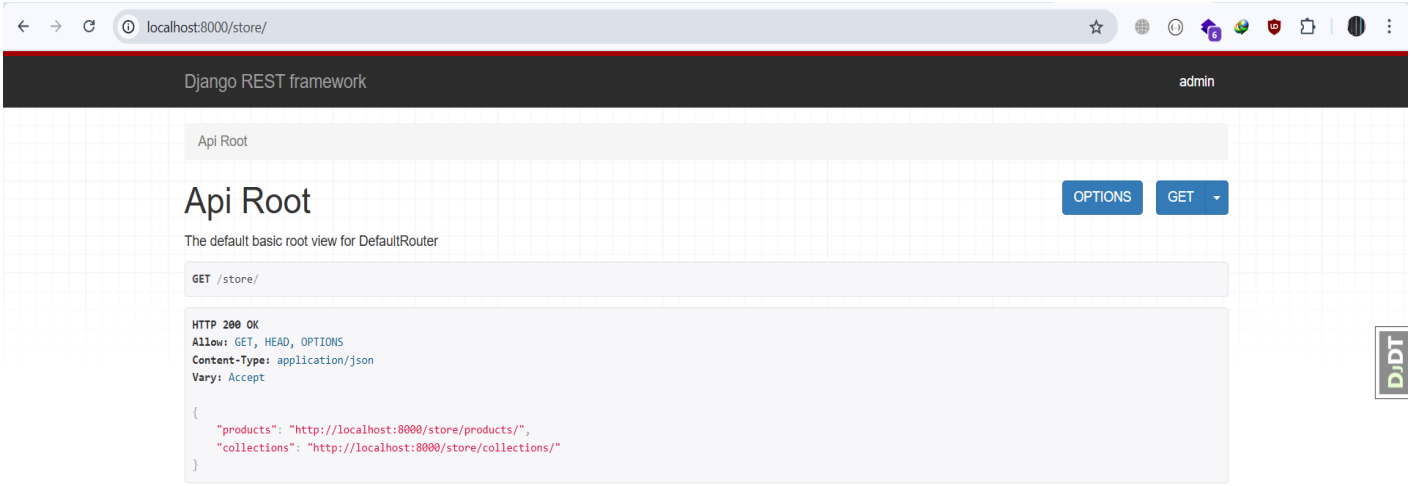
1. The Listing Of URLs Inside The Browser; Ex: http://localhost:8000/store/
2. If We Want Only The Data In JSON Format; Ex: http://localhost:8000/store/products.json

```
from rest_framework.routers import DefaultRouter
from . import views

router = DefaultRouter()

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

urlpatterns = router.urls
**************************************************************************
```

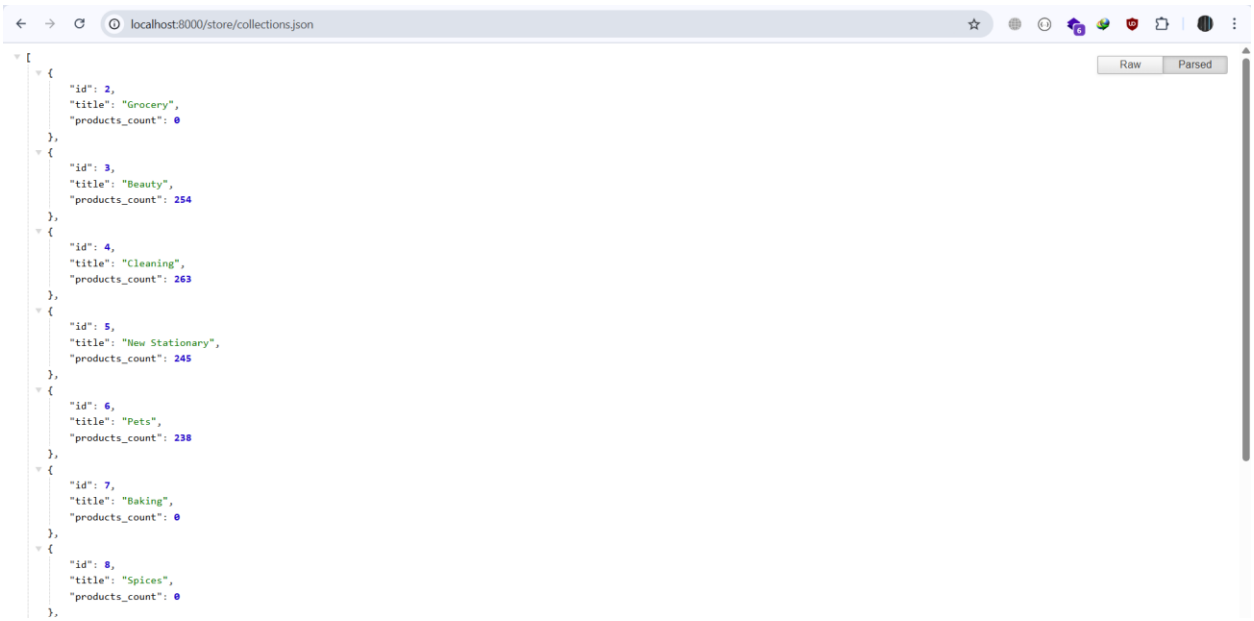
When We Use View Set, And We Want To Use Delete Method, We Must Override The destroy-Method:

```
class CollectionViewSet(ModelViewSet):
    queryset =
Collection.objects.annotate(products_count=Count('products')).all()

    serializer_class = CollectionSerializer

    def destroy(self, request, pk: int):
        collection = get_object_or_404(
            Collection.objects.annotate(products_count=Count('products')),
            pk=pk
        )

        if collection.products.count() > 0:
            return Response({'error': 'Collection Cannot Be Deleted'},
status=status.HTTP_405_METHOD_NOT_ALLOWED)

        collection.delete()

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

****



****



****

If We Want To Implement Delete Method Using View Sets, We Must Override The Destroy Method:

```
class CollectionViewSet(ModelViewSet):
    queryset =
Collection.objects.annotate(products_count=Count('products')).all()

    serializer_class = CollectionSerializer

    def destroy(self, request, *args, **kwargs):
        if Product.objects.filter(collection_id=kwargs['pk']).count() > 0:
            return Response({'error': 'Collection Cannot Be Deleted'},
status=status.HTTP_405_METHOD_NOT_ALLOWED)

        return super().destroy(request, *args, **kwargs)
```

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

    serializer_class = ProductSerializer

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

    def destroy(self, request, *args, **kwargs):
        if OrderItem.objects.filter(product_id = kwargs['pk']).count() > 0:
            return Response({'error': 'product cannot be deleted'},
status=status.HTTP_405_METHOD_NOT_ALLOWED)

        return super().destroy(request, *args, *kwargs)
**************************************************************************
```