Advanced API Concepts

Class-based Views

All the views we've written so far were **Function Based Views**. But DRF also supports Class Based Views which make our code cleaner and more concise + lot's of reuse opportunities .

```
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)
        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
            )
```

- No more `if` statements
- Cleaner Code

Routing

```
# In the urls module

urlpatterns = [
```

```
        path('products/', views.ProductList.as_view())
]
```

Mixins

A common pattern can be seen across these Class-based Views.
for instance look at the implementation of list views

- Creating a QuerySet
- Creating a Serializer and giving it the QuerySet
- Return serialized data in the Response

> ***If it's repeated, there's a better way***

QuerySet Difference

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

# Product
queryset = Product.objects.all()
```

Serializer Class Difference

```
# collection
serializer = CollectionSerializer()

# Product
serializer = ProductSerializer()
```

Same applies to creating a resource

- Deserializing Data
- Data Validation

- Saving Objects
- Returning the serializer data
  This Pattern is also repeated

Mixins

A mixin is a class that encapsulates some pattern of code like this.
Available mixins:

- `ListModelMixin`
- `CreateModelMixin`
- `RetrieveModelMixin`
- `UpdateModelMixin`
- `DestroyModelMixin`

> Tip
>
> - See the implementation of these mixins
> - Look at DRF docs for more mixins

Generic Views

Most of the time, we don't use mixins directly. instead we use concrete classes that combine these mixins together. we call these classes `Generic Views`

- `ListCreateAPIView` - Combines `CreateModelMixin` and `ListModelMixin`
- ...

Generic Views provide methods for overriding the changing part of each view like:

- `get_queryset()`
- `get_serializer_class()`

```
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}
```

If we want to implement some logic in the process of getting these objects (**QuerySet, Serializer, ...**) we should override the methods. But there's also an easier way.

```
class ProductList(ListCreateAPIView):
    queryset =
Product.objects.select_related('collection').all()
    serializer_class = ProductSerializer

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

🔥 Browsable API Changes

Now we have an HTML form for creating objects. we can also use JSON like before

Exercise

Convert Collection List View to a Generic View

Customizing Generic Views

There are situations where a generic view may not quite work for us. so let's see how we can customize it.

Product Detail has:

- GET
- PUT
- DELETE
  Equivalent Generic Class -> `RetriveUpdateDestroyAPIView`

```
class ProductDetail(RetrieveUpdateDestroyAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer

    def delete(self, request, pk, **kwargs):
        product = get_object_or_404(Product, pk=pk)
        if product.orderitems.count() > 0:
            return Response(
                {"error": "Product cannot be deleted
because it's associated with an order item."},
                status=status.HTTP_405_METHOD_NOT_ALLOWED
            )
        product.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

Overriding the Delete Method because it has custom logic

ViewSets

Currently our resources have two views

- a `ListView`
- a `DetailView`
  There are some duplications between these classes like
- `queryset`
- `serializer_class`

> ***With `ViewSets` we can combine the logic for multiple related views inside a single class. That's why it's called `ViewSet`.***

```
from rest_framework.viewsets import ModelViewSet
```

Implemetation

```
class ModelViewSet(mixins.CreateModelMixin,
                   mixins.RetrieveModelMixin,
                   mixins.UpdateModelMixin,
                   mixins.DestroyModelMixin,
                   mixins.ListModelMixin,
                   GenericViewSet):
```

```
class GenericViewSet(ViewSetMixin,
generics.GenericAPIView):
```

Usage

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

> A single class for an entire endpoint

- List

- Create
- Update
- Delete

Now our application is broken because for routing these `ViewSets` we need `routers`.

Exercise

Create Collection View Set

Read-only View Sets

if we only want to supply `GET` operations we need to use `ReadOnlyModelViewSet` instead.

- List all objects
- Retrieve a Single object

Routers

When working with `ViewSets` we don't declare URL patterns like before

- `/products`
- `/products/<int:pk>`
  This is the job of a **Router**, which generates these URL patterns for us.

```
from rest_framework.routers import SimpleRouter

router = SimpleRouter()
#                Prefix     ViewSet
router.register('products', views.ProductViewSet)
router.register('collections', views.CollectionViewSet)
```

```
# Generated Routes
router.urls
```

Router Generated URLs

```
[
        <URLPattern '^products/$' # Regex
        [name='product-list']>,

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

        <URLPattern '^collections/$'
        [name='collection-list']>,

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

Assigning `urlpatterns`

Because we don't have any explicit `urlpatterns` in this app, we can do the following

```
urlpatters = router.urls
```

However if we did have any special URLs

```
urlpatterns = [
        path('', include(router.urls)),
        path('/declared', views.special),
]
```

`DefaultRouter()`

This is another router defined in the `routers` module. Usage is the same as `SimpleRouter()` however, we get two additional features

- `/store` - Shows API Root which has hyperlinks for all available endpoints
- `/store/products.json` - Returns the data in raw JSON format

Building the Reviews API

- `/products/1/reviews`
- `/products/reviews/1`
  We have Nested Resources so we need Nested Routers
  But before that we need our review Model
- Create a model
- Create a migration
- Run the migration against the database

```
class Review(models.Model):
    product = models.ForeignKey(
        Product,
        on_delete=models.CASCADE,
        related_name='reviews'
    )
    name = models.CharField(max_length=255)
    description = models.TextField()
    date = models.DateField(auto_now_add=True)
```

Building The API

- Create a `Serializer`
- Create a `ViewSet`
- Register a `route`

Serializer

```
class ReviewSerializer(serializers.ModelSerializer):
    class Meta:
        model = Review
        fields = [
            'id',
            'product',
            'name',
            'description',
            'date'
        ]
        read_only_fields = ['product']

    def create(self, validated_data):
        review = Review(
                        **validated_data,

product_id=self.context['product_id']
                )
        review.save()
        return review
```

ViewSet

```
class ReviewViewSet(ModelViewSet):
    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']}
```

Route

```
from rest_framework_nested import routers
from . import views
```

```
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
```

Nested Routers

third-party package for managing nested routers `drf-nested-routers`

- The documentation is awesome so always read it

This is the idea

- `/products/1/reviews`
- `/products/reviews/1`

> ***Product is a*** `domain` ***and reviews is a*** `nameserver`.
> Signatures:
> `/domain/{domain_pk}/nameservers/{pk}`

```
from rest_framework_nested import routers
from . import views

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
```

Filtering

- Filtering Products by collection
- Filtering Customers by membership
- ...

> This is done with `query string parameters`
> `/products?collection_id=1`

```
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:
            return
queryset.filter(collection_id=collection_id)
        return queryset
```

Generic Filtering

- Third party library `django-filter`
- Filter any model by any field
- Add custom classes for declaring special filters

```
from django_filters.rest_framework import
DjangoFilterBackend

class ProductViewSet(ModelViewSet):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    filter_backends = [DjangoFilterBackend]
    filterset_fields = ['collection_id']
```

Filtering by `unit_price`

The Field lookup for price cannot be `exact` so

```
<<new module named `filters.py`>>
from django_filters.rest_framework import FilterSet

class ProductFilter(FilterSet):
        class Meta:
                model = Product
                fields = {
                        'collection_id': ['exact'],
                        'unit_price': ['lt', 'gt']
                }
```

```
class ProductViewSet(ModelViewSet):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    filter_backends = [DjangoFilterBackend]
    filterset_class = ProductFilter
```

Searching

- Find products via title / description
- Searching is for text based fields

```
from rest_framework.filters import SearchFilter
# This is another FilterBackend

class ProductViewSet(ModelViewSet):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    filter_backends = [DjangoFilterBackend, SearchFilter]
    filterset_class = ProductFilter
    search_fields = ['title', 'description']
# Case-Insensitive
```

Sorting

```
from rest_framework.filters import SearchFilter,
OrderingFilter

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

`PageNumberPagination`

```
from rest_framework.pagination import 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
```

settings.py

```
REST_FRAMEWORK = {
    'COERCE_DECIMAL_TO_STRING': False,
    'DEFAULT_PAGINATION_CLASS':
'rest_framework.pagination.PageNumberPagination'
    'PAGE_SIZE': 10
}
```

LimitOffsetPagination

```
REST_FRAMEWORK = {
    'COERCE_DECIMAL_TO_STRING': False,
    'DEFAULT_PAGINATION_CLASS':
'rest_framework.pagination.LimitOffsetPagination'
    'PAGE_SIZE': 10
}
```

> `PageNumber` is better.
> Don't use pagination globally

CustomPagination

```
<<New Module `pagination.py`>>

from rest_framework.pagination import PageNumberPagination

class DefaultPagination(PageNumberPagination):
        page_size = 10
# Then remove all pagination related settings in
`settings.py`
```

Then

```
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
```