Filterning in DRF

Filtering in DRF allows users to narrow down the results of a query by adding parameters to the URL.

```
GET /api/books/?author=3

# This will return only the books where author has the ID 3.
```

Why Use Filtering

- Makes your API more dynamic and useful.
- Reduces the need for multiple hard-coded endpoints.
- Enables frontend and mobile apps to fetch filtered data easily.

Let’s assume you already have a Book model with a foreign key to an Author.

Step 1 : Install django-filter

This is an external package used for filtering in DRF.

```
pip install django-filter
```

Step 2 : Add django_filters to INSTALLED_APPS

```
INSTALLED_APPS = [
    ...
    'django_filters',  # Needed for DjangoFilterBackend to work
]
```

Step 3: Configure DRF to Use Filtering Backend

```
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS': [
        'django_filters.rest_framework.DjangoFilterBackend',  # Enables filtering
support
    ]
}

# This tells DRF: "Use this backend whenever a view supports filtering."
```

Step 4: Update Your ViewSet

```
from rest_framework import viewsets
from django_filters.rest_framework import DjangoFilterBackend
from .models import Book
from .serializers import BookSerializer

class BookViewSet(viewsets.ModelViewSet):
    queryset = Book.objects.all()
    serializer_class = BookSerializer

    # This enables filtering in this ViewSet
    filter_backends = [DjangoFilterBackend]

    # These are the model fields that can be filtered via URL query params
    filterset_fields = ['author', 'title']
```

What Happens Behind the Scenes?

- DjangoFilterBackend reads filterset_fields.
- It looks for query params that match the listed fields.
- If it finds them, it applies .filter() on the queryset.

Important Notes to Remember

1. filterset_fields must match actual model field names

- You can filter by ForeignKey fields (like author) using their ID.
- Example: author=1 not author=John.

2. Only listed fields are filterable

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

# /api/books/?price=20 → will NOT work if price isn't listed.
```

3. You can filter on related fields For example, if Book has a foreign key to Author, and Author has a name field, you can do

```
filterset_fields = ['author__name']


Than call

/api/books/?author__name=William #This is very useful for nested relationships.
```

4. You can use custom lookup expressions

```
filterset_fields = {
    'title': ['exact', 'icontains', 'startswith'],
    'author': ['exact'],
}

Now you can do:

/api/books/?title__icontains=django → case-insensitive contains
/api/books/?title__startswith=Django
```

Filterning in DRF

Searching allows API users to find records based on keywords across one or more fields — such as a book's title or its author's name.

```
GET /api/books/?search=django
```

Returns all books where the title or author's name contains the word "django" (case-insensitive).

Why Use Searching?

- Flexible and user-friendly.
- Great for global keyword lookups.
- Essential for search bars, autocomplete, and admin filtering.

Step 1: Add SearchFilter to Your Project

You don’t need to install anything extra. SearchFilter is included in DRF by default.

Just add it to your REST framework config or view.

Step 1 : Global setup in settings.py

```
# settings.py

REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS': [
        'rest_framework.filters.SearchFilter',
    ]
}
```

Step 2: Use per ViewSet (Recommended)

```
```

```
# books/views.py

from rest_framework import viewsets, filters
from .models import Book
from .serializers import BookSerializer

class BookViewSet(viewsets.ModelViewSet):
    queryset = Book.objects.all()
    serializer_class = BookSerializer

    # Enables search functionality
    filter_backends = [filters.SearchFilter]

    # Fields you want to enable for search
    search_fields = ['title', 'author__name']
```

How it works?

```
GET /api/books/?search=django

DRF will automatically apply the icontains lookup on both:
* Book.title
* Author.name (via author__name)
```

Important Notes and Tips

1. Uses icontains by default Case-insensitive, Partial matches allowed

Search is not tokenized like a full-text search engine

2. Works with Foreign Key fields using __ You can search by related fields like author__name

3. search_fields must list valid fields DRF will raise an error if the field doesn't exist

Related fields (author__name) work only if related model has a **str**() method

Ordering in DRF

Ordering allows users to sort API results based on one or more fields.

```
GET /api/books/?ordering=title # This will return books ordered alphabetically by
title (A to Z).

GET /api/books/?ordering=-id # Adds a minus (-) to sort in descending order by ID.
```

Why Use Ordering?

- Useful for listing newest or oldest items.
- Helps users sort alphabetically, by price, by date, etc.
- Supports multiple field sorting.

Step 1: No extra installation needed

Ordering support is already built into DRF using OrderingFilter.

Step 2: Add OrderingFilter Globally (Optional)

```
# settings.py

REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS': [
        'rest_framework.filters.OrderingFilter',
    ]
}
```

Step 3: Use OrderingFilter in Your ViewSet

```
# views.py

from rest_framework import viewsets, filters
from .models import Book
from .serializers import BookSerializer

class BookViewSet(viewsets.ModelViewSet):
    queryset = Book.objects.all()
    serializer_class = BookSerializer

    # Enables ordering
    filter_backends = [filters.OrderingFilter]

    # Users can sort by these fields via ?ordering=field
    ordering_fields = ['title', 'id']

    # 🔧 Default ordering (if no ordering param is passed)
    ordering = ['id']
```

Example : Scenario: Job Recruitment Platform

Models:Company, JobPosting, Applicant and Application (through model between Job & Applicant)

Folder structure

```
recruitment_platform/
├── recruitment_platform/
```

```
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   ├── asgi.py
│   └── wsgi.py
├── recruitment/           ← Your main app
│   ├── models.py
│   ├── views.py
│   ├── serializers.py
│   ├── urls.py
│   ├── admin.py
│   ├── apps.py
│   └── tests.py
└── manage.py
```

Step 1: Define the Models

```
from django.db import models

class Company(models.Model):
    name = models.CharField(max_length=255)
    website = models.URLField()

    def __str__(self):
        return self.name

class JobPosting(models.Model):
    title = models.CharField(max_length=100)
    description = models.TextField()
    location = models.CharField(max_length=100)
    created_at = models.DateTimeField(auto_now_add=True)
    company = models.ForeignKey(Company, related_name='jobs',
on_delete=models.CASCADE)

    def __str__(self):
        return self.title

class Applicant(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField(unique=True)
    resume = models.TextField()

    def __str__(self):
        return self.name

class Application(models.Model):
    applicant = models.ForeignKey(Applicant, related_name='applications',
on_delete=models.CASCADE)
    job = models.ForeignKey(JobPosting, related_name='applications',
on_delete=models.CASCADE)
    applied_at = models.DateTimeField(auto_now_add=True)
    status = models.CharField(max_length=50, choices=[
        ('submitted', 'Submitted'),
        ('reviewing', 'Reviewing'),
        ('accepted', 'Accepted'),
        ('rejected', 'Rejected'),
    ], default='submitted')

    class Meta:
        unique_together = ('applicant', 'job')  # Can't apply to the same job
twice

    def __str__(self):
        return f"{self.applicant.name} -> {self.job.title}"
```

Step 2: Serializers with Nested and Custom Validations

```
from rest_framework import serializers
from .models import Company, JobPosting, Applicant, Application

class CompanySerializer(serializers.ModelSerializer):
    class Meta:
        model = Company
        fields = '__all__'

class JobPostingSerializer(serializers.ModelSerializer):
    class Meta:
        model = JobPosting
        fields = '__all__'

class ApplicantSerializer(serializers.ModelSerializer):
    class Meta:
        model = Applicant
        fields = '__all__'

class ApplicationSerializer(serializers.ModelSerializer):
    class Meta:
        model = Application
        fields = '__all__'

    def validate(self, data):
        # Prevent same applicant from applying to the same job twice (already
handled by unique_together, but just to show)
        if Application.objects.filter(applicant=data['applicant'],
job=data['job']).exists():
            raise serializers.ValidationError("You have already applied to this
job.")
        return data
```

Step 3: ViewSets

```
from rest_framework import viewsets
from .models import Company, JobPosting, Applicant, Application
from .serializers import CompanySerializer, JobPostingSerializer,
ApplicantSerializer, ApplicationSerializer

class CompanyViewSet(viewsets.ModelViewSet):
    queryset = Company.objects.all()
    serializer_class = CompanySerializer

class JobPostingViewSet(viewsets.ModelViewSet):
    queryset = JobPosting.objects.all()
    serializer_class = JobPostingSerializer

class ApplicantViewSet(viewsets.ModelViewSet):
    queryset = Applicant.objects.all()
    serializer_class = ApplicantSerializer

class ApplicationViewSet(viewsets.ModelViewSet):
    queryset = Application.objects.all()
    serializer_class = ApplicationSerializer
```

Step 4: Register URLs

```
from rest_framework.routers import DefaultRouter
from .views import CompanyViewSet, JobPostingViewSet, ApplicantViewSet,
ApplicationViewSet
from django.urls import path, include

router = DefaultRouter()
router.register('companies', CompanyViewSet)
router.register('jobs', JobPostingViewSet)
router.register('applicants', ApplicantViewSet)
router.register('applications', ApplicationViewSet)

urlpatterns = [
    path('', include(router.urls)),
]
```

Step 5: Filtering Applications

```
pip install django-filter

Than update,
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS': [
        'django_filters.rest_framework.DjangoFilterBackend',  # Enables filtering
support
        'rest_framework.filters.SearchFilter',  # Enables search
        'rest_framework.filters.OrderingFilter',  # Enables ordering
    ],
}
```

Now Add Filtering to ApplicationViewSet

```
from rest_framework import viewsets
from django_filters.rest_framework import DjangoFilterBackend  # Import filter
backend
from .models import Company, JobPosting, Applicant, Application
from .serializers import CompanySerializer, JobPostingSerializer,
ApplicantSerializer, ApplicationSerializer

class ApplicationViewSet(viewsets.ModelViewSet):
    queryset = Application.objects.all()
    serializer_class = ApplicationSerializer

    # Enable filtering by applicant, job, and status
    filter_backends = [DjangoFilterBackend]
    filterset_fields = ['applicant', 'job', 'status']  # ?
applicant=1&job=2&status=submitted


# Test :
# GET /api/applications/?status=reviewing
# GET /api/applications/?job=2
```

Step 6: Ordering Job Postings by Date

```
class JobPostingViewSet(viewsets.ModelViewSet):
    queryset = JobPosting.objects.all()
    serializer_class = JobPostingSerializer

    # Enable search and ordering
    filter_backends = [filters.SearchFilter, filters.OrderingFilter]
    search_fields = ['title', 'location']
    ordering_fields = ['created_at']  # Allows ?ordering=created_at or ?ordering=-
created_at
```

Example : E-Learning Platform We'll build an online course management API, similar to systems like Coursera or Udemy.

Key Entities (Models): Instructor, Course (created by an instructor), Student, Enrollment (student enrolls in a course), Lesson (part of a course)

Realistic Business Rules:

- An Instructor can create multiple Courses
- A Course contains multiple Lessons
- A Student can enroll in many Courses
- A Course can have many Students via the Enrollment
- A student cannot enroll in the same course twice
- Lessons are only accessible to enrolled students

```
+-------------------+
|  Instructor       |
+-------------------+
| id: int           |
| name: str         |
| email: str        |
+-------------------+
         |
         | 1
         |
         | *
+-------------------+
|  Course           |
+-------------------+
| id: int           |
| title: str        |
| description: str  |
| created_at: date  |
| instructor_id: FK|
+-------------------+
         |
         | 1
         |
         | *
+-------------------+
|   Lesson          |
+-------------------+
| id: int           |
| title: str        |
| content: text     |
| course_id: FK     |
+-------------------+

+-------------------+
|  Student          |
+-------------------+
| id: int           |
| name: str         |
| email: str        |
+-------------------+
         ^
         |
         | *
         |
         | 1
+-------------------+
|  Enrollment       |
+-------------------+
| id: int           |
| enrolled_at: dt   |
| student_id: FK    |
| course_id: FK     |
+-------------------+
```