- Les filtres Django ORM classiques
- Leur utilisation dans Django REST Framework (DRF)
- Comment intégrer ces filtres dans la doc Swagger/OpenAPI avec **drf-spectacular**

Cours complet : Filtres Django + DRF + drf-spectacular (Swagger/OpenAPI)

---

1. Les filtres Django ORM (rappel rapide)

Opérateur	Description	Exemple
`field=value`	Égal à	`name='Axel'`
`field__in`	Dans une liste	`id__in=[1,2,3]`
`field__icontains`	Contient (insensible casse)	`title__icontains='cake'`
`field__gt`	Supérieur à	`age__gt=18`
`field__range`	Entre 2 valeurs	`created_at__range=[start, end]`
…	…	…

---

2. Filtres dans Django REST Framework (DRF)

2.1 Ajouter `django-filter` au projet

```
pip install django-filter
```

2.2 Configurer DRF pour utiliser `django-filter`

Dans `settings.py` :

```
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS':
['django_filters.rest_framework.DjangoFilterBackend'],
}
```

2.3 Définir un FilterSet pour ton modèle

Exemple pour un modèle `Recipe` :

```
import django_filters
from .models import Recipe

class RecipeFilter(django_filters.FilterSet):
    # Filtre exact sur le champ 'user'
    user = django_filters.CharFilter(field_name='user__username',
lookup_expr='exact')

    # Recherche insensible à la casse dans le titre
    title = django_filters.CharFilter(field_name='title',
lookup_expr='icontains')

    # Filtre entre une plage de dates
    created_at = django_filters.DateFromToRangeFilter()

    class Meta:
        model = Recipe
        fields = ['user', 'title', 'created_at']
```

2.4 Utiliser ce filtre dans ton ViewSet

```
from django_filters.rest_framework import DjangoFilterBackend

class RecipeViewSet(viewsets.ModelViewSet):
    queryset = Recipe.objects.all()
    serializer_class = RecipeSerializer
    filter_backends = [DjangoFilterBackend]
    filterset_class = RecipeFilter
```

---

3. Ajouter les filtres dans ta documentation Swagger avec drf-spectacular

3.1 Installer drf-spectacular

```
pip install drf-spectacular
```

3.2 Configurer drf-spectacular dans `settings.py`

```
REST_FRAMEWORK = {
    # Tes autres configs ...
    'DEFAULT_SCHEMA_CLASS': 'drf_spectacular.openapi.AutoSchema',
}
```

```
SPECTACULAR_SETTINGS = {
    'TITLE': 'Mon API',
    'DESCRIPTION': 'Documentation API avec filtres',
    'VERSION': '1.0.0',
}
```

3.3 Intégrer dans `urls.py`

```
from drf_spectacular.views import SpectacularAPIView,
SpectacularSwaggerView

urlpatterns = [
    # ... tes autres urls
    path('api/schema/', SpectacularAPIView.as_view(), name='schema'),
    path('api/docs/', SpectacularSwaggerView.as_view(url_name='schema'),
name='swagger-ui'),
]
```

3.4 Utiliser `@extend_schema` pour documenter les filtres (optionnel mais conseillé)

```
from drf_spectacular.utils import extend_schema, OpenApiParameter,
OpenApiTypes

@extend_schema(
    parameters=[
        OpenApiParameter(name='user', description='Filtrer par username',
required=False, type=OpenApiTypes.STR),
        OpenApiParameter(name='title', description='Recherche dans le
titre', required=False, type=OpenApiTypes.STR),
        OpenApiParameter(name='created_at_after', description='Date
création après', required=False, type=OpenApiTypes.DATE),
        OpenApiParameter(name='created_at_before', description='Date
création avant', required=False, type=OpenApiTypes.DATE),
    ]
)
class RecipeViewSet(viewsets.ModelViewSet):
    # ... ta config habituelle
```

Cela ajoute des champs filtres dans ta doc Swagger.

---

4. Exemple complet de `views.py` avec filtres et doc

```
import django_filters
from rest_framework import viewsets
```

```
from django_filters.rest_framework import DjangoFilterBackend
from drf_spectacular.utils import extend_schema, OpenApiParameter,
OpenApiTypes
from .models import Recipe
from .serializers import RecipeSerializer

class RecipeFilter(django_filters.FilterSet):
    user = django_filters.CharFilter(field_name='user__username',
lookup_expr='exact')
    title = django_filters.CharFilter(field_name='title',
lookup_expr='icontains')
    created_at = django_filters.DateFromToRangeFilter()

    class Meta:
        model = Recipe
        fields = ['user', 'title', 'created_at']

@extend_schema(
    parameters=[
        OpenApiParameter(name='user', description='Filtrer par username',
required=False, type=OpenApiTypes.STR),
        OpenApiParameter(name='title', description='Recherche dans le
titre', required=False, type=OpenApiTypes.STR),
        OpenApiParameter(name='created_at_after', description='Date
création après', required=False, type=OpenApiTypes.DATE),
        OpenApiParameter(name='created_at_before', description='Date
création avant', required=False, type=OpenApiTypes.DATE),
    ]
)
class RecipeViewSet(viewsets.ModelViewSet):
    queryset = Recipe.objects.all()
    serializer_class = RecipeSerializer
    filter_backends = [DjangoFilterBackend]
    filterset_class = RecipeFilter
```

---

5. Résumé des commandes essentielles

- Installer django-filter et drf-spectacular :

```
pip install django-filter drf-spectacular
```

- Configurer `settings.py` :

```
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS':
['django_filters.rest_framework.DjangoFilterBackend'],
    'DEFAULT_SCHEMA_CLASS': 'drf_spectacular.openapi.AutoSchema',
}
```

```
SPECTACULAR_SETTINGS = {
    'TITLE': 'Mon API',
    'DESCRIPTION': 'Documentation API avec filtres',
    'VERSION': '1.0.0',
}
```

- Ajouter les URLs pour la doc Swagger :

```
from drf_spectacular.views import SpectacularAPIView,
SpectacularSwaggerView

urlpatterns = [
    path('api/schema/', SpectacularAPIView.as_view(), name='schema'),
    path('api/docs/', SpectacularSwaggerView.as_view(url_name='schema'),
name='swagger-ui'),
]
```

Conclusion

Avec cette méthode, tu as :

- Des filtres puissants et déclaratifs dans ta vue DRF via `django-filter`
- Une documentation Swagger propre, complète et interactive via `drf-spectacular`
- Des paramètres de filtres visibles et testables dans `/api/docs`

Top, je te fais un cours complet sur les filtres Django ORM avec toutes les variantes importantes — tu pourras le garder comme référence claire !

Cours complet : Les filtres en Django ORM

Introduction

Les filtres Django ORM te permettent de récupérer des objets dans ta base de données selon des conditions. La syntaxe de base est :

```
Model.objects.filter(champ__operateur=valeur)
```

`champ` est le nom du champ (attribut) du modèle, `operateur` est une sorte de "modificateur" (souvent appelé "lookup"), et `valeur` est ce que tu cherches.

Les opérateurs (lookups) les plus courants

1. Égalité simple

```
MyModel.objects.filter(name='Axel')
```

Requête SQL : `WHERE name = 'Axel'`

---

2. `__in`

- Recherche une valeur dans une liste (ou iterable).
- Ex:

```
MyModel.objects.filter(id__in=[1, 2, 5])
```

SQL : `WHERE id IN (1, 2, 5)`

---

3. `__contains` et `__icontains`

- Cherche une sous-chaîne dans un champ texte.
- `__contains` est sensible à la casse.
- `__icontains` est insensible à la casse.

Ex :

```
MyModel.objects.filter(title__contains='cake')
MyModel.objects.filter(title__icontains='cake')
```

SQL équivalent : `WHERE title LIKE '%cake%'` (sensible ou insensible selon la version)

---

4. `__startswith` et `__istartswith`

- Cherche si un champ commence par une chaîne.

```
MyModel.objects.filter(name__startswith='Ax')
MyModel.objects.filter(name__istartswith='ax')
```

---

5. `__endswith` et `__iendswith`

- Cherche si un champ finit par une chaîne.

```
MyModel.objects.filter(email__endswith='@gmail.com')
MyModel.objects.filter(email__iendswith='@gmail.com')
```

---

6. Comparaisons numériques et date

- `__gt` : greater than (>)
- `__gte` : greater than or equal (>=)
- `__lt` : less than (<)
- `__lte` : less than or equal (<=)

Ex :

```
MyModel.objects.filter(age__gt=18)
MyModel.objects.filter(price__lte=100)
MyModel.objects.filter(date__gte='2023-01-01')
```

---

7. `__exact`

- Correspondance exacte (équivalent à l'égalité simple).

```
MyModel.objects.filter(name__exact='Axel')
```

---

8. `__isnull`

- Filtrer sur la présence ou absence d'une valeur (null).

```
MyModel.objects.filter(deleted_at__isnull=True)  # Pas supprimé
MyModel.objects.filter(deleted_at__isnull=False) # Supprimé
```

---

9. `__regex` et `__iregex`

- Recherche par expression régulière (sensible ou non à la casse).

```
MyModel.objects.filter(name__regex=r'^[A-Z].*')
MyModel.objects.filter(name__iregex=r'^[a-z].*')
```

---

10. `__range`

- Filtre entre une plage de valeurs (inclus).

```
MyModel.objects.filter(created_at__range=['2023-01-01', '2023-06-30'])
```

---

11. `__date`, `__year`, `__month`, `__day`

- Permettent de filtrer sur une partie spécifique d'une date/heure.

```
MyModel.objects.filter(created_at__year=2023)
MyModel.objects.filter(created_at__month=5)
MyModel.objects.filter(created_at__date='2023-05-31')
```

---

12. `__uuid` (pour les champs UUID)

```
MyModel.objects.filter(uuid_field__exact=some_uuid_value)
```

---

13. `__bool`

- Filtrer sur un champ booléen.

```
MyModel.objects.filter(is_active=True)
```

Combinaison de filtres

Tu peux combiner plusieurs filtres avec `.filter()` (qui fait un AND) :

```
MyModel.objects.filter(is_active=True, age__gte=18)
```

Ou utiliser Q objects pour faire des OR / NOT :

```
from django.db.models import Q

MyModel.objects.filter(Q(name__icontains='ax') | Q(email__icontains='ax'))
```

Résumé tableau

Opérateur	Description	Exemple
`field=value`	Égal à	`name='Axel'`
`field__in`	Dans une liste	`id__in=[1,2,3]`
`field__contains`	Contient (sensible casse)	`title__contains='cake'`
`field__icontains`	Contient (insensible casse)	`title__icontains='cake'`
`field__startswith`	Commence par	`name__startswith='Ax'`
`field__istartswith`	Commence par (insensible casse)	`name__istartswith='ax'`
`field__endswith`	Finit par	`email__endswith='@gmail.com'`
`field__iendswith`	Finit par (insensible casse)	`email__iendswith='@gmail.com'`
`field__gt`	Supérieur à	`age__gt=18`
`field__gte`	Supérieur ou égal	`price__gte=100`
`field__lt`	Inférieur à	`age__lt=60`
`field__lte`	Inférieur ou égal	`price__lte=500`
`field__exact`	Exact	`name__exact='Axel'`
`field__isnull`	Est null ou pas	`deleted_at__isnull=True`
`field__regex`	Expression régulière	`name__regex=r'^[A-Z]'`
`field__range`	Entre 2 valeurs	`created_at__range=[start, end]`
`field__year`	Année sur un champ date	`created_at__year=2023`
`field__month`	Mois sur un champ date	`created_at__month=5`

Exemples pratiques

```
# Utilisateurs actifs âgés de 18 à 30 ans
User.objects.filter(is_active=True, age__gte=18, age__lte=30)

# Articles dont le titre contient "django" (insensible à la casse)
Article.objects.filter(title__icontains='django')

# Produits dont le prix est dans une fourchette
Product.objects.filter(price__range=[10, 50])

# Recettes avec tag dans une liste
Recipe.objects.filter(tags__name__in=['vegan', 'dessert'])

# Articles publiés cette année
Article.objects.filter(published_at__year=2025)

# Combinaison OR avec Q objects
```

```
from django.db.models import Q
User.objects.filter(Q(first_name__icontains='alex') |
Q(last_name__icontains='alex'))
```

Conclusion

Les filtres Django ORM sont très puissants et te permettent d'exprimer facilement des requêtes complexes. L'astuce est de bien comprendre la syntaxe `champ__operateur=valeur` et de savoir quand utiliser les combinaisons avec `Q()` pour les expressions logiques.

Si tu veux, je peux aussi te faire un cours pour intégrer ces filtres dans Django REST Framework via des ViewSets et des paramètres d'URL !

Veux-tu que je te fasse ça ?