

---

# **django-rest-framework-datatables**

## **Documentation**

*Release 0.1.0*

**David Jean Louis**

**Dec 09, 2021**



---

## Contents:

---

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Quickstart</b>	<b>5</b>
2.1	Installation . . . . .	5
2.2	Configuration . . . . .	5
2.3	And that's it ! . . . . .	6
<b>3</b>	<b>Tutorial</b>	<b>7</b>
3.1	Backend code . . . . .	7
3.2	A minimal datatable . . . . .	9
3.3	A more complex and detailed example . . . . .	11
3.4	Filtering . . . . .	13
3.5	Pagination . . . . .	14
3.6	Using DataTables via POST method . . . . .	14
3.7	Handling Duplicates in Sorting . . . . .	14
<b>4</b>	<b>The example app</b>	<b>15</b>
4.1	Postgres . . . . .	16
<b>5</b>	<b>Integration with Django-filter</b>	<b>19</b>
5.1	Django-Filter Quickstart . . . . .	19
5.2	Differences and Limitations . . . . .	20
5.3	Query Optimization . . . . .	20
5.4	Customizing (global) queries . . . . .	21
5.5	Further Reading . . . . .	21
<b>6</b>	<b>Changelog</b>	<b>23</b>
6.1	Version 0.7.0 (2021-12-09): . . . . .	23
6.2	Version 0.6.0 (2021-02-09): . . . . .	23
6.3	Version 0.5.2 (2020-04-10): . . . . .	23
6.4	Version 0.5.1 (2020-01-13): . . . . .	24
6.5	Version 0.5.0 (2019-03-31): . . . . .	24
6.6	Version 0.4.1 (2018-11-16): . . . . .	24
6.7	Version 0.4.0 (2018-06-22): . . . . .	24
6.8	Version 0.3.0 (2018-05-11): . . . . .	24
6.9	Version 0.2.1 (2018-04-11): . . . . .	24
6.10	Version 0.2.0 (2018-04-11): . . . . .	25

6.11	Version 0.1.0 (2018-04-10): . . . . .	25
<b>7</b>	<b>Useful links</b>	<b>27</b>

Seamless integration between Django REST framework and Datatables.

**Django Rest Framework + Datatables = Awesome :)**

Full example with foreign key and many to many relation

All time
50's
60's
70's
80's
90's
00's
10's

Show
10
entries

Search:

Rank	Artist	Album name	Year	Genres
11	Elvis Presley	The Sun Sessions	1976	Rock & Roll
12	Miles Davis	Kind of Blue	1959	Modal
13	The Velvet Underground	The Velvet Underground & Nico	1967	Art Rock, Experimental, Garage Rock
14	The Beatles	Abbey Road	1969	Classic Rock, Pop Rock, Psychedelic Rock
15	The Jimi Hendrix Experience	Are You Experienced	1967	Blues Rock, Psychedelic Rock
16	Bob Dylan	Blood on the Tracks	1975	Acoustic, Ballad, Folk Rock
17	Nirvana	Nevermind	1991	Alternative Rock, Grunge
18	Bruce Springsteen	Born to Run	1975	Pop Rock
19	Van Morrison	Astral Weeks	1968	Acoustic, Classic Rock, Free Improvisation
20	Michael Jackson	Thriller	1982	Disco

Showing 11 to 20 of 500 entries

Previous
1
2
3
4
5
...
50
Next



# CHAPTER 1

---

## Introduction

---

django-rest-framework-datatables provides seamless integration between [Django REST framework](#) and [Datatables](#).

Just call your API with `?format=datatables`, and you will get a JSON structure that is fully compatible with what Datatables expects.

A “normal” call to your existing API will look like this:

```
$ curl http://127.0.0.1:8000/api/albums/ | python -m "json.tool"
```

```
{
  "count": 2,
  "next": null,
  "previous": null,
  "results": [
    {
      "rank": 1,
      "name": "Sgt. Pepper's Lonely Hearts Club Band",
      "year": 1967,
      "artist_name": "The Beatles",
      "genres": "Psychedelic Rock, Rock & Roll"
    },
    {
      "rank": 2,
      "name": "Pet Sounds",
      "year": 1966,
      "artist_name": "The Beach Boys",
      "genres": "Pop Rock, Psychedelic Rock"
    }
  ]
}
```

The same call with `datatables` format will look a bit different:

```
$ curl http://127.0.0.1:8000/api/albums/?format=datatables | python -m "json.tool"
```

```
{
  "recordsFiltered": 2,
  "recordsTotal": 2,
  "draw": 1,
  "data": [
    {
      "rank": 1,
      "name": "Sgt. Pepper's Lonely Hearts Club Band",
      "year": 1967,
      "artist_name": "The Beatles",
      "genres": "Psychedelic Rock, Rock & Roll"
    },
    {
      "rank": 2,
      "name": "Pet Sounds",
      "year": 1966,
      "artist_name": "The Beach Boys",
      "genres": "Pop Rock, Psychedelic Rock"
    }
  ]
}
```

As you can see, django-rest-framework-datatables automatically adapt the JSON structure to what Datatables expects. And you don't have to create a different API, your API will still work as usual unless you specify the `datatables` format on your request.

But django-rest-framework-datatables can do much more ! As you will learn in the tutorial, it speaks the Datatables language and can handle searching, filtering, ordering, pagination, etc. Read the [quickstart guide](#) for instructions on how to install and configure django-rest-framework-datatables.



## 2.1 Installation

Just use pip:

```
$ pip install djangorestframework-datatables
```

## 2.2 Configuration

To enable Datatables support in your project, add 'rest\_framework\_datatables' to your INSTALLED\_APPS, and modify your REST\_FRAMEWORK settings like this:

```
REST_FRAMEWORK = {
    'DEFAULT_RENDERER_CLASSES': (
        'rest_framework.renderers.JSONRenderer',
        'rest_framework.renderers.BrowsableAPIRenderer',
        'rest_framework_datatables.renderers.DatatablesRenderer',
    ),
    'DEFAULT_FILTER_BACKENDS': (
        'rest_framework_datatables.filters.DatatablesFilterBackend',
    ),
    'DEFAULT_PAGINATION_CLASS': 'rest_framework_datatables.pagination.
↪DatatablesPageNumberPagination',
    'PAGE_SIZE': 50,
}
```

What have we done so far ?

- we added the `rest_framework_datatables.renderers.DatatablesRenderer` to existings renderers
- we added the `rest_framework_datatables.filters.DatatablesFilterBackend` to the filter backends

- we replaced the pagination class by `rest_framework_datatables.pagination.DatatablesPageNumberPagination`
- we added the `rest_framework_datatables.pagination.DatatablesOnlyPageNumberPagination` to paginate queries only when datatables format requested

---

**Note:** If you are using `rest_framework.pagination.LimitOffsetPagination` as pagination class, relax and don't panic ! `django-rest-framework-datatables` can handle that, just replace it with `rest_framework_datatables.pagination.DatatablesLimitOffsetPagination`.

---

## 2.3 And that's it !

Your API is now fully compatible with Datatables and will provide searching, filtering, ordering and pagination without any modification of your API code, to continue, follow the [tutorial](#).

---

**Note:** The purpose of this section is not to replace the excellent [Django REST Framework documentation](#) nor the [Datatables manual](#), it is just to give you hints and gotchas for using your datatables compatible API.

---

### 3.1 Backend code

So we have the following backend code, nothing very complicated if you are familiar with Django and Django REST Framework:

albums/models.py:

```
from django.db import models

class Genre(models.Model):
    name = models.CharField('Name', max_length=80)

    class Meta:
        ordering = ['name']

    def __str__(self):
        return self.name

class Artist(models.Model):
    name = models.CharField('Name', max_length=80)

    class Meta:
        ordering = ['name']

    def __str__(self):
```

(continues on next page)

(continued from previous page)

```

        return self.name

class Album(models.Model):
    name = models.CharField('Name', max_length=80)
    rank = models.PositiveIntegerField('Rank')
    year = models.PositiveIntegerField('Year')
    artist = models.ForeignKey(
        Artist,
        models.CASCADE,
        verbose_name='Artist',
        related_name='albums'
    )
    genres = models.ManyToManyField(
        Genre,
        verbose_name='Genres',
        related_name='albums'
    )

    class Meta:
        ordering = ['name']

    def __str__(self):
        return self.name

```

albums/serializers.py:

```

from rest_framework import serializers
from .models import Album

class ArtistSerializer(serializers.ModelSerializer):
    id = serializers.IntegerField(read_only=True)

    class Meta:
        model = Artist
        fields = (
            'id', 'name',
        )

class AlbumSerializer(serializers.ModelSerializer):
    artist = ArtistSerializer()
    genres = serializers.SerializerMethodField()

    def get_genres(self, album):
        return ', '.join([str(genre) for genre in album.genres.all()])

    class Meta:
        model = Album
        fields = (
            'rank', 'name', 'year', 'artist_name', 'genres',
        )

```

albums/views.py:

```

from django.shortcuts import render
from rest_framework import viewsets

```

(continues on next page)

(continued from previous page)

```
from .models import Album
from .serializers import AlbumSerializer

def index(request):
    return render(request, 'albums/albums.html')

class AlbumViewSet(viewsets.ModelViewSet):
    queryset = Album.objects.all().order_by('rank')
    serializer_class = AlbumSerializer
```

urls.py:

```
from django.conf.urls import url, include
from rest_framework import routers
from albums import views

router = routers.DefaultRouter()
router.register(r'albums', views.AlbumViewSet)

urlpatterns = [
    url('^api/', include(router.urls)),
    url('', views.index, name='albums')
]
```

## 3.2 A minimal datatable

In this example, we will build a simple table that will list music albums, we will display 3 columns, the album rank, name and release year. For the sake of simplicity we will also use HTML5 data attributes (which are supported by Datatables).

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Rolling Stone Top 500 albums of all time</title>
  <meta name="description" content="Rolling Stone magazine's 2012 list of 500_
↪greatest albums of all time with genres.">
  <link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.0.
↪0/css/bootstrap.css">
  <link rel="stylesheet" href="//cdn.datatables.net/1.10.16/css/dataTables.bootstrap4.
↪min.css">
</head>

<body>
  <div class="container">
    <div class="row">
      <div class="col-sm-12">
        <table id="albums" class="table table-striped table-bordered" style="width:100
↪%" data-server-side="true" data-ajax="/api/albums/?format=datatables">
          <thead>
```

(continues on next page)

(continued from previous page)

```

        <tr>
            <th data-data="rank">Rank</th>
            <th data-data="name">Album name</th>
            <th data-data="year">Year</th>
        </tr>
    </thead>
</table>
</div>
</div>
</div>
<script src="//code.jquery.com/jquery-1.12.4.js"></script>
<script src="//cdn.datatables.net/1.10.16/js/jquery.dataTables.min.js"></script>
<script src="//cdn.datatables.net/1.10.16/js/dataTables.bootstrap4.min.js"></script>
<script>
    $(document).ready(function() {
        $('#albums').DataTable();
    });
</script>
</body>
</html>

```

And that's it ! At this point, you should have a fully functional Datatable with search, ordering and pagination !

What we just did:

- included all the necessary CSS and JS files
- set the table `data-server-side` attribute to `true`, to tell Datatables to use the server-side processing mode
- set the table `data-ajax` to our API URL with `?format=datatables` as query parameter
- set a `data-data` attribute for the two columns to tell Datatables what properties must be extracted from the response
- and finally initialized the Datatable via a javascript one-liner.

Perhaps you noticed that we didn't use all fields from our serializer in the above example, that's not a problem, `django-rest-framework-datatables` will automatically filter the fields that are not necessary when processing the request from Datatables.

If you want to force serialization of fields that are not requested by Datatables you can use the `datatables_always_serialize` Meta option in your serializer, here's an example:

```

class AlbumSerializer(serializers.ModelSerializer):
    id = serializers.IntegerField(read_only=True)
    class Meta:
        model = Album
        fields = (
            'id', 'rank', 'name', 'year',
        )
        datatables_always_serialize = ('id', 'rank',)

```

In the above example, the fields 'id' and 'rank' will always be serialized in the response regardless of fields requested in the Datatables request.

**Hint:** Alternatively, if you wish to choose which fields to preserve at runtime rather than hardcoding them into your serializer models, use the `?keep=` param along with the fields you wish to maintain (comma separated). For example, if you wished to preserve `id` and `rank` as before, you would simply use the following API call:

```
data-ajax="/api/albums/?format=datatables&keep=id,rank"
```

In order to provide additional context of the data from the view, you can use the `datatables_extra_json` Meta option.

```
class AlbumViewSet(viewsets.ModelViewSet):
    queryset = Album.objects.all().order_by('rank')
    serializer_class = AlbumSerializer

    def get_options(self):
        return "options", {
            "artist": [{ 'label': obj.name, 'value': obj.pk } for obj in Artist.objects.
↪all()],
            "genre": [{ 'label': obj.name, 'value': obj.pk } for obj in Genre.objects.
↪all()]
        }

    class Meta:
        datatables_extra_json = ('get_options', )
```

In the above example, the `'get_options'` method will be called to populate the rendered JSON with the key and value from the method's return tuple.

**Important:** To sum up, the most important things to remember here are:

- don't forget to add `?format=datatables` to your API URL
- you must add a **data-data attribute** or specify the column data property via JS for each columns, the name must match one of the fields of your DRF serializers.

### 3.3 A more complex and detailed example

In this example we want to display more information about the album:

- the album artist name (`Album.artist` is a foreignkey to `Artist` model)
- the genres (`Album.genres` is a many to many relation with `Genre` model)

The HTML/JS code will look like this:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Rolling Stone Top 500 albums of all time</title>
  <meta name="description" content="Rolling Stone magazine's 2012 list of 500_
↪greatest albums of all time with genres.">
  <link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.0.
↪0/css/bootstrap.css">
  <link rel="stylesheet" href="//cdn.datatables.net/1.10.16/css/dataTables.bootstrap4.
↪min.css">
</head>

<body>
```

(continues on next page)

(continued from previous page)

```
<div class="container">
  <div class="row">
    <div class="col-sm-12">
      <table id="albums" class="table table-striped table-bordered" style="width:100
→%" data-server-side="true" data-ajax="/api/albums/?format=datatables">
        <thead>
          <tr>
            <th data-data="rank">Rank</th>
            <th data-data="artist.name" data-name="artist.name">Artist</th>
            <th data-data="name">Album name</th>
            <th data-data="year">Year</th>
            <th data-data="genres" data-name="genres.name">Genres</th>
          </tr>
        </thead>
      </table>
    </div>
  </div>
</div>
<script src="//code.jquery.com/jquery-1.12.4.js"></script>
<script src="//cdn.datatables.net/1.10.16/js/jquery.dataTables.min.js"></script>
<script src="//cdn.datatables.net/1.10.16/js/dataTables.bootstrap4.min.js"></script>
<script>
  $(document).ready(function() {
    $('#albums').DataTable();
  });
</script>
</body>
</html>
```

Notice that artist and genres columns have an extra data attribute: `data-name`, this attribute is necessary to tell to the `django-rest-framework-datatables` builtin filter backend what field part to use to filter and reorder the queryset. The builtin filter will add `__icontains` to the string to perform the filtering/ordering.

We could also have written that in a more conventional form (without data attributes), for example:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Rolling Stone Top 500 albums of all time</title>
  <meta name="description" content="Rolling Stone magazine's 2012 list of 500
→greatest albums of all time with genres.">
  <link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.0.
→0/css/bootstrap.css">
  <link rel="stylesheet" href="//cdn.datatables.net/1.10.16/css/dataTables.bootstrap4.
→min.css">
</head>

<body>
  <div class="container">
    <div class="row">
      <div class="col-sm-12">
        <table id="albums" class="table table-striped table-bordered" style="width:100
→%">
          <thead>
            <tr>
              <th>Rank</th>
```

(continues on next page)



(continued from previous page)

```

        <th>Artist</th>
        <th>Album name</th>
        <th>Year</th>
        <th>Genres</th>
    </tr>
</thead>
</table>
</div>
</div>
</div>
<script src="//code.jquery.com/jquery-1.12.4.js"></script>
<script src="//cdn.datatables.net/1.10.16/js/jquery.dataTables.min.js"></script>
<script src="//cdn.datatables.net/1.10.16/js/dataTables.bootstrap4.min.js"></script>
<script>
    $(document).ready(function() {
        $('#albums').DataTable({
            'serverSide': true,
            'ajax': '/api/albums/?format=datatables',
            'columns': [
                {'data': 'rank'},
                {'data': 'artist.name', 'name': 'artist.name'},
                {'data': 'name'},
                {'data': 'year'},
                {'data': 'genres', 'name': 'genres.name'},
            ]
        });
    });
</script>
</body>
</html>

```

**Hint:** Datatables uses the dot notation in the `data` field to populate columns with nested data. In this example, `artist.name` refers to the field name within the nested serializer `artist`.

## 3.4 Filtering

Filtering is based off of either the `data` or `name` fields. If you need to filter on multiple fields, you can always pass through multiple variables like so

```

<script>
    'columns': [
        {'data': 'artist.name', 'name': 'artist.name, artist__year'}
    ]
</script>

```

This would allow you to filter the `artist.name` column based upon `name` or `year`.

Because the `name` field is used to filter on Django queries, you can use either dot or double-underscore notation as shown in the example above.

The values within a single `name` field are tied together using a logical OR operator for filtering, while those between `name` fields are strung together with an AND operator. This means that Datatables' multicolumn search functionality

is preserved.

If you need more complex filtering and ordering, you can always implement your own filter backend by inheriting from `rest_framework_datatables.DatatablesFilterBackend`.

---

**Important:** To sum up, for **foreign keys and relations** you need to specify a **name for the column** otherwise filtering and ordering will not work.

---

You can see this code live by running the [example app](#).

## 3.5 Pagination

If you want to disable pagination for a particular table, you can pass the value `-1` in the *length* parameter to do this, just set the *pageLength* DataTable option to `-1`.

If you want to disable pagination completely just set *PAGE\_SIZE* to `None` in your *REST\_FRAMEWORK* settings dictionary and disable paging on your DataTable options (by setting *paging* to `false`).

## 3.6 Using DataTables via POST method

By default, the Ajax request that DataTables makes to obtain server-side processing data is an HTTP GET request. However, there are times when you might wish to use POST, DRF-Datatables can handle this, just configure your DataTable as explained in the [related Datatables documentation section](#).

## 3.7 Handling Duplicates in Sorting

If sorting is done on a single column with more duplicates than the page size it's possible that some rows are never retrieved as we traverse through our datatable. This is because of how order by together with limit and offset works in the database.

As a workaround for this problem we add a second column to sort by in the case of ties.

```
class AlbumViewSet(viewsets.ModelViewSet):
    queryset = Album.objects.all().order_by('year')
    serializer_class = AlbumSerializer
    datatables_additional_order_by = 'rank'
```

## CHAPTER 4

---

### The example app

---

django-rest-framework-datatables comes with an example application (the Rolling Stone top 500 albums of all time). It's a great start for understanding how things work, you can play with several options of Datatables, modify the python code (serializers, views) and test a lot of possibilities.

We encourage you to give it a try with a few commandline calls:

```
$ git clone https://github.com/izimobil/django-rest-framework-datatables.git
$ cd django-rest-framework-datatables
$ pip install -r requirements-dev.txt
$ python example/manage.py runserver
$ firefox http://127.0.0.1:8000
```

A screenshot of the example app:

## Full example with foreign key and many to many relation

All time
50's
60's
70's
80's
90's
00's
10's

Show 10 entries

Search:

Rank	Artist	Album name	Year	Genres
11	Elvis Presley	The Sun Sessions	1976	Rock & Roll
12	Miles Davis	Kind of Blue	1959	Modal
13	The Velvet Underground	The Velvet Underground & Nico	1967	Art Rock, Experimental, Garage Rock
14	The Beatles	Abbey Road	1969	Classic Rock, Pop Rock, Psychedelic Rock
15	The Jimi Hendrix Experience	Are You Experienced	1967	Blues Rock, Psychedelic Rock
16	Bob Dylan	Blood on the Tracks	1975	Acoustic, Ballad, Folk Rock
17	Nirvana	Nevermind	1991	Alternative Rock, Grunge
18	Bruce Springsteen	Born to Run	1975	Pop Rock
19	Van Morrison	Astral Weeks	1968	Acoustic, Classic Rock, Free Improvisation
20	Michael Jackson	Thriller	1982	Disco

Showing 11 to 20 of 500 entries

Previous
1
2
3
4
5
...
50
Next

## 4.1 Postgres

You can use Postgres as the source database. This example uses [Docker](#) as a container for the database.

Initialise the database as follows:

```
$ pip install psycopg2-binary

$ export DRFDT_TEST_TYPE=postgres
$ export DRFDT_POSTGRESQL_USER=pguser
$ export DRFDT_POSTGRESQL_PASSWORD=pguserpass
$ export DJANGO_SETTINGS_MODULE=example.settings

$ # start a local postgres instance
$ docker-compose -f example/pg/docker-compose.yml up -d

$ python example/manage.py migrate
$ python example/manage.py test

$ # only required if you want to login to the Admin site
$ python example/manage.py createsuperuser --username admin --email=email@example.com

$ # load test data
$ python example/manage.py loaddata test_data

$ # shutdown the db (append -v to remove the data volume and delete all data)
```

(continues on next page)

(continued from previous page)

```
$ docker-compose -f example/pg/docker-compose.yml down
```



---

## Integration with Django-filter

---

django-rest-framework-datatables will always use `icontains` or `iregex` queries on all columns, which may be costly. More fine-grained control on the generated database queries can be achieved with [Django-filter](#).

Integration with Django-filter is provided through Datatables-specific `DatatablesFilterSet` and `DatatablesFilterBackend` classes. These may be found in the `django_filters` subpackage.

### 5.1 Django-Filter Quickstart

Using the new `DatatablesFilterBackend` simply requires changing the import path. Instead of importing from `rest_framework_datatables` import from the `django_filters` subpackage.

```
from django_filters import filters
from rest_framework_datatables.django_filters.backends import DatatablesFilterBackend
from rest_framework_datatables.django_filters.filterset import DatatablesFilterSet
from rest_framework_datatables.django_filters.filters import GlobalFilter

class GlobalCharFilter(GlobalFilter, filters.CharFilter):
    pass

class AlbumGlobalFilter(DatatablesFilterSet):
    """Filter name, artist and genre by name with icontains"""

    name = GlobalCharFilter(lookup_expr='icontains')
    genres = GlobalCharFilter(field_name='genres__name', lookup_expr='icontains')
    artist = GlobalCharFilter(field_name='artist__name', lookup_expr='icontains')
    year = GlobalCharFilter()

    class Meta:
        model = Album
        fields = '__all__'
```

(continues on next page)

(continued from previous page)

```
class AlbumGlobalViewSet(viewsets.ModelViewSet):
    queryset = Album.objects.all()
    serializer_class = AlbumSerializer
    filter_backends = (DatatablesFilterBackend,)
    filterset_class = AlbumGlobalFilter
```

## 5.2 Differences and Limitations

The `data-data` attribute or the `columns[<n>][data]` query parameter for the column must contain the name of the filter attribute on the filterset.

Although the `DjangoFilterBackend` uses the same parser for the queries generated by `DataTables`, the `data-name` attribute (or `columns[<n>][name]` query parameter) is completely ignored by the `django-filter` backend.

You can specify the lookup in the `lookup_expr` kwarg on the `Filter`. The ordering is implemented by looking at the `field_name` and `lookup_expr` attributes of the filters.

(Because of that, the `keep` parameter will not have the same semantics. It will cause the renderer to return the column, but it won't be respected for global search by the backend and filterset. That means that only columns defined in the datatable can be filtered.)

The Django-Rest-Framework browsable api will not support global filters and the `DataTables` javascript frontend can't take advantage of the automatic widgets generated by `Django-filter`.

## 5.3 Query Optimization

The above example will act as a drop-in replacement for the standard behaviour of `django-rest-framework-datatables`, which uses `icontains` and `iregex` for local and global queries.

With large tables this might generate very inefficient queries especially for non-string datatypes.

By simply not using the `GlobalFilter` mixin, you can switch off global search per column to gain efficiency.

```
class AlbumGlobalFilter(AlbumFilter):
    """Filter name, artist and genre by name with icontains"""

    name = GlobalCharFilter(lookup_expr='icontains')
    genres = GlobalCharFilter(field_name='genres__name', lookup_expr='icontains')
    artist = GlobalCharFilter(field_name='artist__name', lookup_expr='icontains')

    class Meta:
        model = Album
        fields = '__all__'
```

This will revert the `year` field to the `NumberFilter` automatically generated by `Django-filter` which will require an exact number match.

Also you can use the capability of `Django-filter` to automatically generate the `FilterSet` for you:



```
class AlbumFilterViewSet(viewsets.ModelViewSet):
    queryset = Album.objects.all()
    serializer_class = AlbumSerializer
    filter_backends = [DatatablesFilterBackend]
    filterset_fields = '__all__'
```

In this case there will be no support for regular expressions, `icontains` or global searches, as `Django-filter` will use automatic lookups (e.g `exact` for strings), and you'll need to add appropriate widgets to the datatable, because `genres` will need a multiple selection. It's possible to use a javascript library such as `yadcf` to ease that task.

See the *example app* for an example of multiple selection using `yadcf`.

## 5.4 Customizing (global) queries

The defined filters will be used to filter the column search queries. Global queries are implemented with the optional `global_q` method on the `GlobalFilter` mixin. This will generate `icontains` or `iregex` lookups by default.

If you want more fine-grained control over queries, you can simply define your own filters.

Only filters that provide a `global_q` method will support global search queries.

The `global_q` method (as for example in the `GlobalFilter` mixin), should return a [Q-object](#) for the global field query. All these Q-objects will be combined with `|` (OR) and the resulting Q-object will be used to filter the queryset that was returned by the applying the column filters.

This logic is identical to the one implemented by plain `django-rest-framework-datatables`.

## 5.5 Further Reading

It's highly recommended to read the [documentation of Django-filter](#).



### 6.1 Version 0.7.0 (2021-12-09):

- Django 4.0 compatibility
- Added global search support to YADCFModelMultipleChoiceFilter
- Various fixes on filters
- Various fixes on pagination
- Fixed / improved documentation and examples

Many thanks to all the contributors on this release !

### 6.2 Version 0.6.0 (2021-02-09):

- Integration with django-filter
- Example of using yadcf and django-filter to create a multi-select column
- Fixed support for POST requests from datatables
- Some fixes on pagination

Many thanks to all the contributors on this release !

### 6.3 Version 0.5.2 (2020-04-10):

- Added support for POST requests from datatables
- Avoid extra count queries
- Handle dummy columns gracefully

## 6.4 Version 0.5.1 (2020-01-13):

- Added support for Django 3.0
- Added support for disabling pagination when the client requests it with `length=-1` parameter
- Added optional column sorting to handle ties
- Minor code fixes

## 6.5 Version 0.5.0 (2019-03-31):

- Fixed total number of rows when view is using multiple filter back-ends
- New meta option `datatables_extra_json` on view for adding key/value pairs to rendered JSON
- Minor docs fixes

## 6.6 Version 0.4.1 (2018-11-16):

- Added support for Django 2.1 and DRF 3.9
- Updated README

## 6.7 Version 0.4.0 (2018-06-22):

- Added top level filtering for nested serializers
- Added multiple field filtering
- Added a `?keep=` parameter that allows to bypass the filtering of unused fields
- Better detection of the requested format
- Fixed typo in `Queryset.count()` method name

## 6.8 Version 0.3.0 (2018-05-11):

- Added a serializer Meta option `datatables_always_serialize` that allows to specify a tuple of fields that should always be serialized in the response, regardless of what fields are requested in the Datatables request
- Optimize filters
- Use AND operator for column filtering instead of OR, to be consistent with the client-side behavior of Datatables

## 6.9 Version 0.2.1 (2018-04-11):

- This version replaces the 0.2.0 who was broken (bad `setup.py`)

## 6.10 Version 0.2.0 (2018-04-11):

- Added full documentation
- Removed serializers, they are no longer necessary, filtering of columns is made by the renderer

## 6.11 Version 0.1.0 (2018-04-10):

Initial release.



## CHAPTER 7

---

### Useful links

---

- [Github project page](#)
- [Bugtracker](#)
- [Documentation](#)
- [Pypi page](#)