
django-rest-framework-datatables- editor

Release 0.3.0

Vyacheslav VV

Jun 10, 2022

CONTENTS:

1	Introduction	3
1.1	View tables	3
1.2	Editing tables	4
2	Quickstart	5
2.1	Installation	5
2.2	Configuration	5
2.3	And that's it !	6
3	Tutorial	7
3.1	Backend code	7
3.2	A minimal datatable	9
3.3	A more complex and detailed example with the ability to edit data	11
3.4	Authorization	13
3.5	Filtering	14
4	The example app	15
5	Changelog	17
5.1	Version 0.3.3 (2020-05-17):	17
5.2	Version 0.3.2 (2019-05-23):	17
5.3	Version 0.3.1 (2019-05-22):	17
5.4	Version 0.3.0 (2019-05-06):	17
5.5	Version 0.2.1 (2019-04-29):	17
5.6	Version 0.2.0 (2019-04-20):	17
5.7	Version 0.1.0 (2019-04-15):	18
5.8	Version 0.5.0 (2019-03-31):	18
5.9	Version 0.4.1 (2018-11-16):	18
5.10	Version 0.4.0 (2018-06-22):	18
5.11	Version 0.3.0 (2018-05-11):	18
5.12	Version 0.2.1 (2018-04-11):	19
5.13	Version 0.2.0 (2018-04-11):	19
5.14	Version 0.1.0 (2018-04-10):	19
6	Useful links	21
7	Indices and tables	23

Seamless integration between Django REST framework and Datatables with supporting Datatables Editor.

Django Rest Framework + Datatables + Editor = Awesome :)

The project is based on the project [django-rest-framework-datatables](#) by David Jean Louis

Full example with foreign key and many to many relation

New

Edit

Delete

All time

50's

60's

70's

80's

90's

00's

10's

Search:

Rank	Artist	Album name	Year	Genres
1	The Beatles	Sgt. Pepper's Lonely Hearts Club Band	1967	Psychedelic Rock, Rock & Roll
2	The Beach Boys	Pet Sounds	1966	Pop Rock, Psychedelic Rock
3	The Beatles	Revolver	1966	Pop Rock, Psychedelic Rock
4	Bob Dylan	Highway 61 Revisited	1965	Blues Rock, Folk Rock
5	The Beatles	Rubber Soul	1965	Pop Rock
6	Marvin Gaye	What's Going On	1971	Soul
7	The Rolling Stones	Exile on Main St.	1972	Blues Rock, Classic Rock, Rock & Roll
8	The Clash	London Calling	1979	New Wave, Punk
9	Bob Dylan	Blonde on Blonde	1966	Folk Rock, Rhythm & Blues
10	The Beatles	The Beatles ("The White Album")	1968	Experimental, Pop Rock, Psychedelic Rock, Rock & Roll

Showing 1 to 10 of 500 entries

Previous

1

2

3

4

5

...

50

Next

INTRODUCTION

1.1 View tables

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": [
```

(continues on next page)

(continued from previous page)

```
{
  {
    "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.

1.2 Editing tables

The URL for interaction with the Datatables Editor: <http://127.0.0.1:8000/api/albums/editor> for this view.

You must set the parameter `ajax`: `"/api/albums/editor/` and that's it!

QUICKSTART

2.1 Installation

Just use pip:

```
$ pip install djangoestframework-datatables-editor
```

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_editor.renderers.DatatablesRenderer',
    ),
    'DEFAULT_FILTER_BACKENDS': (
        'rest_framework_datatables_editor.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`

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 Datatables Editor and will provide searching, filtering, ordering and pagination and editing, to continue, follow the [tutorial](#).

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):
        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,
```

(continues on next page)

(continued from previous page)

```

        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)

    # if we need to edit a field that is a nested serializer,
    # we must override to_internal_value method
    def to_internal_value(self, data):
        return get_object_or_404(Artist, pk=data['id'])

    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
from .models import Album
from .serializers import AlbumSerializer

```

(continues on next page)

(continued from previous page)

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

class AlbumViewSet(EditorModelMixin, 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>
            <tr>
              <th data-data="rank">Rank</th>
              <th data-data="name">Album name</th>
              <th data-data="year">Year</th>
```

(continues on next page)

(continued from previous page)

```

        </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 with the ability to edit data

In this example we want to display more informations 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>
    <link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.
↪1.3/css/bootstrap.css">
    <link rel="stylesheet" type="text/css" href="https://cdn.datatables.net/1.10.19/
↪css/dataTables.bootstrap4.min.css">
    <link rel="stylesheet" href="https://cdn.datatables.net/buttons/1.5.6/css/buttons.
↪bootstrap4.min.css">
    <link rel="stylesheet" href="https://cdn.datatables.net/select/1.3.0/css/select.
↪bootstrap4.min.css">
    <link rel="stylesheet" href="/static/css/editor.bootstrap4.min.css">
</head>
<body>
    <div class="container" style="font-size: .9em;">
```

(continues on next page)

(continued from previous page)

```

<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>
          <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-3.3.1.js"></script>
<script src="//cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.1.3/js/
↪ bootstrap.min.js"></script>
<script src="https://cdn.datatables.net/1.10.19/js/jquery.dataTables.min.js"></
↪ script>
<script src="https://cdn.datatables.net/1.10.19/js/dataTables.bootstrap4.min.js">
↪ </script>
<script src="https://cdn.datatables.net/buttons/1.5.6/js/dataTables.buttons.min.js
↪ "></script>
<script src="https://cdn.datatables.net/buttons/1.5.6/js/buttons.bootstrap4.min.js
↪ "></script>
<script src="https://cdn.datatables.net/select/1.3.0/js/dataTables.select.min.js">
↪ </script>
<script src="/static/js/dataTables.editor.js"></script>
<script src="/static/js/editor.bootstrap4.min.js"></script>
<script>
  $(document).ready(function () {
    editor = new $.fn.dataTable.Editor({
      ajax: "/api/albums/editor/?format=datatables",
      table: "#albums",
      fields: [{
        label: "rank",
        name: "rank",
      }, {
        label: "artist:",
        name: "artist.id",
        type: "select"
      }, {
        label: "name:",
        name: "name",
      }, {
        label: "year:",
        name: "year",
      }
    ]
  });
  var table = $('#albums').DataTable({
    "serverSide": true,
    dom: "Bfrtip",
    ajax: "/api/albums/?format=datatables",
    "columns": [

```

(continues on next page)

(continued from previous page)

```

        {"data": "rank", "searchable": false},
        {"data": "artist.name", "name": "artist.name"},
        {"data": "name"},
        {"data": "year"},
        {"data": "genres", "name": "genres.name", "sortable": false},
    ],
    select: true,
    buttons: [
        {extend: "create", editor: editor},
        {extend: "edit", editor: editor},
        {extend: "remove", editor: editor}
    ]
    });
    table.buttons().container()
        .appendTo($('.col-md-6:eq(0)', table.table().container()));
    });
</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.

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 Authorization

If you use user authorization you must sent a CSRF token with each POST request. To do this, you can use the following script:

```

<script>

function getCookie(name) {
    var cookieValue = null;
    if (document.cookie && document.cookie != '') {
        var cookies = document.cookie.split(';');
        for (var i = 0; i < cookies.length; i++) {
            var cookie = jQuery.trim(cookies[i]);
            // Does this cookie string begin with the name we want?
            if (cookie.substring(0, name.length + 1) == (name + '=')) {
                cookieValue = decodeURIComponent(cookie.substring(name.length + 1));
                break;
            }
        }
    }
    return cookieValue;
}

var csrftoken = getCookie('csrftoken');

```

(continues on next page)

(continued from previous page)

```
function csrfSafeMethod(method) {  
    // these HTTP methods do not require CSRF protection  
    return (/^(GET|HEAD|OPTIONS|TRACE)$/.test(method));  
}  
  
$.ajaxSetup({  
    beforeSend: function (xhr, settings) {  
        if (!csrfSafeMethod(settings.type) && !this.crossDomain) {  
            xhr.setRequestHeader("X-CSRFToken", csrftoken);  
        }  
    }  
});  
</script>
```

3.5 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](#).

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/VVyacheslav/django-rest-framework-datatables-editor.git
$ cd django-rest-framework-datatables-editor
$ pip install -r requirements-dev.txt
```

You need to download [Datatables Editor](#), the JS+CSS version, and unpack the downloaded archive in `django-rest-framework-datatables-editor/static`

```
$ 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

New Edit Delete

Search:

Rank	Artist	Album name	Year	Genres
1	The Beatles	Sgt. Pepper's Lonely Hearts Club Band	1967	Psychedelic Rock, Rock & Roll
2	The Beach Boys	Pet Sounds	1966	Pop Rock, Psychedelic Rock
3	The Beatles	Revolver	1966	Pop Rock, Psychedelic Rock
4	Bob Dylan	Highway 61 Revisited	1965	Blues Rock, Folk Rock
5	The Beatles	Rubber Soul	1965	Pop Rock
6	Marvin Gaye	What's Going On	1971	Soul
7	The Rolling Stones	Exile on Main St.	1972	Blues Rock, Classic Rock, Rock & Roll
8	The Clash	London Calling	1979	New Wave, Punk
9	Bob Dylan	Blonde on Blonde	1966	Folk Rock, Rhythm & Blues
10	The Beatles	The Beatles ("The White Album")	1968	Experimental, Pop Rock, Psychedelic Rock, Rock & Roll

Showing 1 to 10 of 500 entries

Previous 1 2 3 4 5 ... 50 Next

CHANGELOG

5.1 Version 0.3.3 (2020-05-17):

- Added support for Django 3.0

5.2 Version 0.3.2 (2019-05-23):

- Fixed checking fields when deleting

5.3 Version 0.3.1 (2019-05-22):

- Fixed requirements

5.4 Version 0.3.0 (2019-05-06):

- Added checking of the writable fields of Datatables Editor passed to Django
- Added information about CSRF authorization to the documentation

5.5 Version 0.2.1 (2019-04-29):

- Added documentation

5.6 Version 0.2.0 (2019-04-20):

- Added tests for editor functionality

5.7 Version 0.1.0 (2019-04-15):

- Initial release.
 - New project released with supporting [Datatables](#) editor.
 - The project is based on [django-rest-framework-datatables](#) by [David Jean Louis](#))
-

5.8 Version 0.5.0 (2019-03-31):

The changelog bellow is the changelog of [django-rest-framework-datatables](#) by [David Jean Louis](#))

- 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

5.9 Version 0.4.1 (2018-11-16):

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

5.10 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

5.11 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 consistant with the client-side behavior of Datatables

5.12 Version 0.2.1 (2018-04-11):

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

5.13 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

5.14 Version 0.1.0 (2018-04-10):

Initial release.

USEFUL LINKS

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

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`