DjangoRestMultipleModels Documentation

Release 2.1.3

Matt Nishi-Broach

Dec 01, 2018

Contents

Django Rest Framework provides some incredible tools for serializing data, but sometimes you need to combine many serializers and/or models into a single API call. **drf-multiple-model** is an app designed to do just that.

CHAPTER 1

Installation

Install the package from pip:

```
pip install django-rest-multiple-models
```

Make sure to add 'drf_multiple_model' to your INSTALLED_APPS:

```
INSTALLED_APPS = (
    ....
    'drf_multiple_model',
)
```

Then simply import the view into any views.py in which you'd want to use it:

```
from drf_multiple_model.views import ObjectMultipleModelAPIView
```

Note: This package is built on top of Django Rest Framework's generic views and serializers, so it presupposes that Django Rest Framework is installed and added to your project as well.

Contents:

1.1 Usage

1.1.1 Basic Usage

drf-multiple-model comes with two generic class-based-view for serializing multiple models: the `ObjectMultipleModelAPIView` and the `FlatMultipleModelAPIView`. Both views require a `querylist` attribute, which is a list or tuple of dicts containing (at minimum) a `queryset` key and a `serializer_class` key; the main difference between the views is the format of the response data. For example, let's say you have the following models and serializers:

```
# Models
class Play(models.Model):
    genre = models.CharField(max_length=100)
    title = models.CharField(max_length=200)
    pages = models.IntegerField()

class Poem(models.Model):
    title = models.CharField(max_length=200)
    style = models.CharField(max_length=100)
    lines = models.IntegerField()
    stanzas = models.IntegerField()

# Serializers
class PlaySerializer(serializers.ModelSerializer):
    class Meta:
        model = Play
        fields = ('genre','title','pages')

class PoemSerializer(serializers.ModelSerializer):
    class Meta:
        model = Poem
        fields = ('title','stanzas')
```

Then you might use the `ObjectMultipleModelAPIView` as follows:

```
from drf_multiple_model.views import ObjectMultipleModelAPIView

class TextAPIView(ObjectMultipleModelAPIView):
    querylist = [
        {'queryset': Play.objects.all(), 'serializer_class': PlaySerializer},
        {'queryset': Poem.objects.filter(style='Sonnet'), 'serializer_class':␣
↪PoemSerializer},
        ....
    ]
```

which would return:

```
{
    'Play' : [
        {'genre': 'Comedy', 'title': "A Midsummer Night's Dream", 'pages': 350},
        {'genre': 'Tragedy', 'title': "Romeo and Juliet", 'pages': 300},
        ....
    ],
    'Poem' : [
        {'title': 'Shall I compare thee to a summer's day?', 'stanzas': 1},
        {'title': 'As a decrepit father takes delight', 'stanzas': 1},
        ....
    ],
}
```

Or you coulde use the `FlatMultipleModelAPIView` as follows:

```
from drf_multiple_model.views import FlatMultipleModelAPIView

class TextAPIView(FlatMultipleModelAPIView):
    querylist = [
        {'queryset': Play.objects.all(), 'serializer_class': PlaySerializer},
```

(continues on next page)

(continued from previous page)

```
        {'queryset': Poem.objects.filter(style='Sonnet'), 'serializer_class':␣
↪PoemSerializer},
        ....
    ]
```

which would return:

```
[
    {'genre': 'Comedy', 'title': "A Midsummer Night's Dream", 'pages': 350, 'type':
↪'Play'},
    {'genre': 'Tragedy', 'title': "Romeo and Juliet", 'pages': 300, 'type': 'Play'},
    ....
    {'title': 'Shall I compare thee to a summer's day?', 'stanzas': 1, 'type': 'Poem'}
↪,
    {'title': 'As a decrepit father takes delight', 'stanzas': 1, 'type': 'Poem'},
    ....
]
```

1.1.2 Mixins

If you want to combine `ObjectMultipleModelAPIView` or `FlatMultipleModelAPIViews`'s `list()` function with other views, you can use their base mixins from `mixins.py` instead.

1.2 Installation

Install the package from pip:

```
pip install django-rest-multiple-models
```

Make sure to add 'drf_multiple_model' to your INSTALLED_APPS:

```
INSTALLED_APPS = (
    ....
    'drf_multiple_model',
)
```

Then simply import the view into any views.py in which you'd want to use it:

```
from drf_multiple_model.views import ObjectMultipleModelAPIView
```

Note: This package is built on top of Django Rest Framework's generic views and serializers, so it presupposes that Django Rest Framework is installed and added to your project as well.

1.3 ObjectMultipleModelAPIView Options

1.3.1 Labels

By default, `ObjectMultipleModelAPIView` uses the model name as a label. If you want to use a custom label, you can add a `label` key to your `querylist` dicts, like so:

```
from drf_multiple_model.views import ObjectMultipleModelAPIView

class TextAPIView(ObjectMultipleModelAPIView):
    querylist = [
        {
            'querylist': Play.objects.all(),
            'serializer_class': PlaySerializer,
            'label': 'drama',
        },
        {
            'querylist': Poem.objects.filter(style='Sonnet'),
            'serializer_class': PoemSerializer,
            'label': 'sonnets'
        },
        ....
    ]
```

which would return:

```
{
    'drama': [
        {'genre': 'Comedy', 'title': "A Midsummer Night's Dream", 'pages': 350},
        {'genre': 'Tragedy', 'title': "Romeo and Juliet", 'pages': 300},
        ....
    ],
    'sonnets':[
        {'title': 'Shall I compare thee to a summer's day?', 'stanzas': 1},
        {'title': 'As a decrepit father takes delight', 'stanzas': 1},
        ....
    ],
}
```

1.4 FlatMultipleModelAPIView Options

1.4.1 Labels

By default, `FlatMultipleModelAPIView` adds a `type` property to returned items with the model name. If you want to use a custom value for the `type` property other than the model name, you can add a `label` key to your `querylist` dicts, like so:

```
from drf_multiple_model.views import FlatMultipleModelAPIView

class TextAPIView(FlatMultipleModelAPIView):
    querylist = [
        {
            'queryset': Play.objects.all(),
            'serializer_class': PlaySerializer,
            'label': 'drama',
        },
        {
            'queryset': Poem.objects.filter(style='Sonnet'),
            'serializer_class': PoemSerializer,
            'label': 'sonnet'
        },
```

(continues on next page)

(continued from previous page)

```
        ....
    ]
```

which would return:

```
[
    {'genre': 'Comedy', 'title': "A Midsummer Night's Dream", 'pages': 350, 'type':
↪'drama'},
    {'genre': 'Tragedy', 'title': "Romeo and Juliet", 'pages': 300, 'type': 'drama'},
    ....
    {'title': 'Shall I compare thee to a summer's day?', 'stanzas': 1, 'type': 'sonnet
↪'},
    {'title': 'As a decrepit father takes delight', 'stanzas': 1, 'type': 'sonnet'},
    ....
]
```

If you'd prefer not to add the `type` property to returned items, you can set the class-level field of `add_model_type` to `False`:

```
class TextAPIView(FlatMultipleModelAPIView):
    add_model_type = False

    querylist = [
        {'queryset': Play.objects.all(), 'serializer_class': PlaySerializer},
        {'queryset': Poem.objects.filter(style='Sonnet'), 'serializer_class':␣
↪PoemSerializer},
        ....
    ]
```

which would return:

```
[
    {'genre': 'Comedy', 'title': "A Midsummer Night's Dream", 'pages': 350},
    {'genre': 'Tragedy', 'title': "Romeo and Juliet", 'pages': 300},
    ....
    {'title': 'Shall I compare thee to a summer's day?', 'stanzas': 1},
    {'title': 'As a decrepit father takes delight', 'stanzas': 1},
    ....
]
```

Note: adding a custom label to your querylist elements will **always** override `add_model_type`. However, labels are taken on an element-by-element basis, so you can add labels for some of your models/querysets, but not others.

1.4.2 sorting_field

By default the objects will be arranged by the order in which the querysets were listed in your `querylist` attribute. However, you can specify a different ordering by adding the `sorting_fields` to your view, which works similar to Django's `ordering`:

```
class TextAPIView(FlatMultipleModelAPIView):
    sorting_fields = ['title']

    querylist = [
        {'queryset': Play.objects.all(), 'serializer_class': PlaySerializer},
        {'queryset': Poem.objects.filter(style='Sonnet'), 'serializer_class':␣
↪PoemSerializer},
```

(continues on next page)

(continued from previous page)

```
        ....
    ]
```

would return:

```
[
    {'genre': 'Comedy', 'title': "A Midsummer Night's Dream", 'pages': 350, 'type':
↪'Play'},
    {'title': 'As a decrepit father takes delight', 'stanzas': 1, 'type': 'Poem'},
    {'genre': 'Tragedy', 'title': "Romeo and Juliet", 'pages': 300, 'type': 'Play'},
    {'title': 'Shall I compare thee to a summer's day?', 'stanzas': 1, 'type': 'Poem'}
↪,
    ....
]
```

As with django field ordering, add '-' to the beginning of the field to enable reverse sorting. Setting `sorting_fields=['-title', 'name']` would sort the title fields in __descending__ order and name in __ascending__

Also, a DRF-style sorting is supported. By default it uses `o` parameter from request query string. `sorting_parameter_name` property controls what parameter to use for sorting. Lookups are working in the django-filters style, like `property_1__property_2` (which will use object's `property_1` and, in turn, its `property_2` as key argument to `sorted()`) Sorting is also possible by several fields. Sorting field have to be split with commas for that. Could be passed either via `sorting_parameter_name` in request parameters, or via view property.

WARNING: the field chosen for ordering must be shared by all models/serializers in your `querylist`. Any attempt to sort objects along non_shared fields will throw a `KeyError`.

1.5 Filtering

1.5.1 Django Rest Framework Filters

Django Rest Frameworks default Filter Backends work out of the box. These filters will be applied to **every** queryset in your queryList. For example, using the *SearchFilter* Backend in a view:

```
class SearchFilterView(ObjectMultipleModelAPIView):
    querylist = (
        {'queryset': Play.objects.all(), 'serializer_class': PlaySerializer},
        {'queryset': Poem.objects.filter(style="Sonnet"), 'serializer_class':␣
↪PoemSerializer},
    )
    filter_backends = (filters.SearchFilter,)
    search_fields = ('title',)
```

accessed with a url like `http://www.example.com/texts?search=as` would return only the `Plays` and `Poems` with "as" in the title:

```
{
    'Play': [
        {'title':'As You Like It','genre':'Comedy','year':1623},
    ],
    'Poem': [
        {'title':"As a decrepit father takes delight",'style':'Sonnet'},
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

1.5.2 Per Queryset Filtering

Using the built in Filter Backends is a nice DRY solution, but it doesn't work well if you want to apply the filter to some items in your queryList, but not others. In order to apply more targeted queryset filtering, DRF Multiple Models provides two technique:

Override get_querylist()

drf-multiple-model now supports the creation of dynamic queryLists, by overwriting the get_queryList() function rather than simply specifying the queryList variable. This allows you to do things like construct queries using url kwargs, etc:

```
class DynamicQueryView(ObjectMultipleModelAPIView):
    def get_querylist(self):
        title = self.request.query_params['play'].replace('-',' ')

        querylist = (
            {'queryset': Play.objects.filter(title=title), 'serializer_class':␣
↪PlaySerializer},
            {'queryset': Poem.objects.filter(style="Sonnet"), 'serializer_class':␣
↪PoemSerializer},
        )

        return querylist
```

That view, if accessed via a url like `http://www.example.com/texts?play=Julius-Caesar` would return only plays that match the provided title, but the poems would be untouched:

```
{
    'play': [
        {'title':'Julius Caesar','genre':'Tragedy','year':1623},
    ],
    'poem': [
        {'title':"Shall I compare thee to a summer's day?",'style':'Sonnet'},
        {'title':"As a decrepit father takes delight",'style':'Sonnet'}
    ],
}
```

Custom Filter Functions

If you want to create a more complicated filter or use a custom filtering function, you can pass a custom filter function as an element in your querylist using the `filter_fn` key:

```
from drf_multiple_model.views import MultipleModelAPIView

def title_without_letter(queryset, request, *args, **kwargs):
    letter_to_exclude = request.query_params['letter']
    return queryset.exclude(title__icontains=letter_to_exclude)
```

(continues on next page)

(continued from previous page)

```
class FilterFnView(MultipleModelAPIView):
    querylist = (
        {'queryset': Play.objects.all(), 'serializer_class': PlaySerializer, 'filter_
↪fn': title_without_letter},
        {'queryset': Poem.objects.all(), 'serializer_class':PoemSerializer},
    )
```

The above view will use the `title_without_letter()` function to filter the queryset and remove and title that contains the provided letter. Accessed from the url `http://www.example.com/texts?letter=o` would return all plays without the letter 'o', but the poems would be untouched:

```
{
    'play': [
        {'title':"A Midsummer Night's Dream",'genre':'Comedy','year':1600},
        {'title':'Julius Caesar','genre':'Tragedy','year':1623},
    ],
    'poem': [
        {'title':"Shall I compare thee to a summer's day?",'style':'Sonnet'},
        {'title':"As a decrepit father takes delight",'style':'Sonnet'},
        {'title':"A Lover's Complaint",'style':'Narrative'}
    ],
}
```

1.6 Pagination

Because Django and Rest Framework's paginators are designed to work with a single model/queryset, they cannot simply be dropped into a `MultipleModelAPIView` and function properly. Currently, only **Limit/Offset** pagination has been ported to **drf_mutliple_model**, although other `rest_framework` paginators may be ported in the future.

1.6.1 Limit/Offset Pagination

Limit/Offset functions very similarly to (and with the same query parameters as) Rest Framework's LimitOffsetPagination, but formatted to handle multiple models:

```
from drf_multiple_model.views import ObjectMultipleModelAPIView
from drf_multiple_model.pagination import MultipleModelLimitOffsetPagination

class LimitPagination(MultipleModelLimitOffsetPagination):
    default_limit = 2


class ObjectLimitPaginationView(ObjectMultipleModelAPIView):
    pagination_class = LimitPagination
    querylist = (
        {'queryset': Play.objects.all(), 'serializer_class': PlaySerializer},
        {'queryset': Poem.objects.all(), 'serializer_class': PoemSerializer},
    )
```

which would return:

```
{
    'highest_count': 4,   # Play model has four objects in the database
    'overall_total': 7,   # 4 Plays + 3 Poems
    'next': 'http://yourserver/yourUrl/?page=2',
    'previous': None,
    'results':
        {
            'Play': [
                {'genre': 'Comedy', 'title': "A Midsummer Night's Dream", 'pages':␣
↪350},
                {'genre': 'Tragedy', 'title': "Romeo and Juliet", 'pages': 300},
            ],
            'Poem': [
                {'title': 'Shall I compare thee to a summer's day?', 'stanzas': 1},
                {'title': 'As a decrepit father takes delight', 'stanzas': 1},
            ],
        }
}
```

This would also work with the `FlatMultipleModelAPIView` (with caveats, see below):

```
class FlatLimitPaginationView(FlatMultipleModelAPIView):
    pagination_class = LimitPagination
    querylist = (
        {'queryset': Play.objects.all(), 'serializer_class': PlaySerializer},
        {'queryset': Poem.objects.all(), 'serializer_class': PoemSerializer},
    )
```

which would return:

```
{
    'highest_count': 4,   # Play model has four objects in the database
    'overall_total': 7,   # 4 Plays + 3 Poems
    'next': 'http://yourserver/yourUrl/?page=2',
    'previous': None,
    'results':
        [
            {'genre': 'Comedy', 'title': "A Midsummer Night's Dream", 'pages': 350},
            {'genre': 'Tragedy', 'title': "Romeo and Juliet", 'pages': 300},
            {'title': 'Shall I compare thee to a summer's day?', 'stanzas': 1},
            {'title': 'As a decrepit father takes delight', 'stanzas': 1}
        ]
}
```

Warning: Important `FlatMultipleModel` caveats below!

The `limit` in LimitOffsetPagination is applied **per queryset**. This means that the number of results returned is actually *number_of_querylist_items * limit*. This is intuitive for the `ObjectMultipleModelAPIView`, but the `FlatMultipleModelAPIView` may confuse some developers at first when a view with a limit of 50 and three different model/serializer combinations in the `querylist` returns a list of 150 items.

The other thing to note about `MultipleModelLimitOffsetPagination` and `FlatMultipleModelAPIView` is that sorting is done **after** the querylists have been filter by the limit/offset pair. To understand why this may return some internal results, imagine a project `ModalA`, which has 50 rows whose `name` field all start with 'A', and ModelB, which has 50 rows whose `name` field all start with 'B'. If limit/offset pagination with a limit of 10 is used in a view that sorts by `name`, the first page will return 10 results with names that start

with 'A' followed by 10 results that start with 'B'. The second page with then **also** contain 10 results that start with 'A' followed by 10 results that start with 'B', which certainly won't map onto a users expectation of alphabetical sorting. Unfortunately, sorting before fetching the data would likely require bypassing Django's querysets entirely and writing raw SQL with a join on the `sorting_field` field, which would be difficult to integrate cleanly into the current system. It is therefore recommended that when using `MultipleModelLimitOffsetPagination` that `sorting_field` values by hidden fields like `id` that won't be visible to the end user.

1.7 ViewSets

For user with ViewSets and Routers, **drf-multiple-model** provides the `ObjectMultipleModelAPIViewSet` and `FlatMultipleModelAPIViewSet`. A simple configuration for using the provided ViewSets might look like:

```
from rest_framework import routers

from drf_multiple_model.viewsets import ObjectMultipleModelAPIViewSet

class TextAPIView(ObjectMultipleModelAPIViewSet):
    querylist = [
        {'queryset': Play.objects.all(), 'serializer_class': PlaySerializer},
        {'queryset': Poem.objects.filter(style='Sonnet'), 'serializer_class':␣
↪PoemSerializer},
        ....
    ]

 router = routers.SimpleRouter()
 router.register('texts', TextAPIView, base_name='texts')
```

WARNING: Because the ObjectMultipleModel views do not provide the `queryset` property, you **must** specify the `base_name` property when you register a `ObjectMultipleModelAPIViewSet` with a router.

The `ObjectMultipleModelAPIViewSet` has all the same configuration options as the `ObjectMultipleModelAPIView` object. For more information, see the *basic usage* section.

1.8 Upgrading from 1.x to 2.0

drf_multiple_model went through a substantial re-write from 1.x to 2.0. Not only did much of the underlying code get re-structured and streamlined, but the classes and API changed as well. Here are some of the biggest changes developers need to be aware of.

1.8.1 views/mixins split in two

Earlier iterations of **drf_multiple_model** tried to shoehorn many different formats and functionalities into a single view/mixin. This was making development increasingly difficult, as potentially problematic interactions grew expenentionally with the number of competing options. Instead of the the single `MultipleModelAPIView`, you should use the following views:

1. If your 1.x view had `flat = True`, you should use the `FlatMultipleModelAPIView`
2. If your 1.x view had `objectify = True`, you should use the `ObjectMultipleModelAPIView`
3. If your 1.x view had both `flat = True` and `objectify = True`, your view was broken and likely raised an Exception. Use one of the options above.

4. If your 1.x view had neither `flat = True` nor `objectify = True`, you should reconsider and use one of the options above. The previously default response structure of `list(dict(list( ... )` made no sense, was overly complicated to consume, and has been removed from v2.0.

1.8.2 querylist is no longer camelCased

The bizarrely camelCased `queryList` field has been renamed the much more pythonic `querylist`

1.8.3 querylist items are now dicts, not lists/tuples

If your 1.x querylist looked like this:

```
queryList = (
    (Poem.objects.all(), PoemSerializer),
    (Play.objects.all(), PlaySerializer),
)
```

your 2.0 querlist should look like this:

```
querylist = (
    {'queryset': Poem.objects.all(), 'serializer_class': PoemSerializer},
    {'queryset': Play.objects.all(), 'serializer_class': PlaySerializer},
)
```

Although this structure is slightly more verbose, is **much** more extensible. Consider, for example, what was needed previously in order to add a per-queryset filter function:

```
from drf_multiple_model.views import MultipleModelAPIView
from drf_multiple_model.mixins import Query

def my_custom_filter_fn(queryset, request, *args, **kwargs):
    ....

class FilterFnView(MultipleModelAPIView):
    queryList = (
        Query(Play.objects.all(), PlaySerializer, filter_fn=my_custom_filter_Fn),
        (Poem.objects.all(), PoemSerializer),
    )
```

This requires importing a special `Query` item, and confusingly mixing types (`Query` object and `tuple`) in the querylist. With the `dict` querylist structure, any number of extra parameters can be added simply by adding an extra key:

```
querylist = (
    {'queryset': Poem.objects.all(), 'serializer_class': PoemSerializer, 'filter_fn':␣
↪my_custom_filter_fn},
    {'queryset': Play.objects.all(), 'serializer_class': PlaySerializer},
)
```

1.8.4 pagination uses custom-built paginators

Pagination in 1.x used the built in **rest_framework** paginators, but didn't actually restricted the items being queried; it simply formated the data **after** it had been fetched to remove extra items. Pagination has been re-written to only query the items request in 2.0, but this means paginators had to be re-written/extended to properly handle multiple querysets.

As such, you can longer simply drop in **rest_framework** paginators and should only use the pagination available in `drf_multiple_model.pagination`. See *Limit/Offset Pagination* for more details.

1.9 Release Notes

1.9.1 2.0 (2018-01-18)

- Refactored underlying code structure and API. Changes include:
  - Removed the nonsensical camelCase from querylist
  - Changing querylist items from lists/tupes to dicts (for more parameter flexibility). Eliminated the underlying `Query` model as a result.
  - Breaking the mixin into two separate mixins: `ObjectMultipleModelMixing` and `FlatMultipleModelMixin`, as well as their respective views and viewsets
  - Removing the previously default response structure of `list(dict(list( ... )`
- Adding limit/offset pagination that actually only queries the items it fetches (rather than iterating the whole queryset)
- Removing pagination functionality from the `FlatMultipleModelMixin` and adding it to the `ObjectMultipleModelMixin`

1.9.2 1.8.1 (2017-12-20)

- Dropped support for Django 1.8 and 1.9 (in keeping with Django Rest Framework's support)
- Expanded test coverage for Django 1.11 and Django 2.0

1.9.3 1.8 (2016-09-04)

- Added `objectify` property to return JSON object instead of an array (implemented by @ELIYAHUT123)
- Added `MultipleModelAPIViewSet` for working with Viewsets (credit to Mike Hwang (@mehwang) for working out the implementation)
- implemented tox for simultaneous testing of all relevant python/django combos
- dropped support for Django 1.7 (based on Django Rest Frameworks's concurrent lack of support)

1.9.4 1.7 (2016-06-09)

- Expanded documentation
- Moved to sphynx docs/readthedocs.org
- Moved data formatting to `format_data()` function to allow for custom post-serialization data handling

1.9.5 1.6 (2016-02-23)

- Incorporated and expanded on reverse sort implemented by @schweickism

1.9.6 1.5 (2016-01-28)

- Added support for Django Rest Framework's pagination classes
- Custom filter functions (implemented by @Symmetric)
- Created Query class for handling queryList elements (implemented by @Symmetric)

1.9.7 1.3 (2015-12-10)

- Improper context passing bug fixed by @rbreu

1.9.8 1.2 (2015-11-11)

- Fixed a bug with the Browsable API when using Django Rest Framework >= 3.3

1.9.9 1.1 (2015-07-06)

- Added `get_queryList()` function to support creation of dynamic queryLists

1.9.10 1.0 (2015-06-29)

- initial release

1.10 Contributors

1.10.1 Project Maintainer and Founder

- Matt Nishi-Broach

1.10.2 Contributors

- rbreu
- Paul Tiplady <Symmetric>
- schweickism
- ELIYAHUT123
- Malcolm Box <mbox>
- Evgen Osiptsov <evgenosiptsov>
- Alexander Anikeev <iamanikeev>
- lerela