DjangoRestMultipleModels Documentation

Release 1.7

Matt Nishi-Broach

Dec 31, 2017

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 MultipleModelAPIView
```

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 the `MultipleModelAPIView` generic class-based-view for serializing multiple models. `MultipleModelAPIView` requires a `queryList` attribute, which is a list or tuple of queryset/serializer pairs (in that order). 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 `MultipleModelAPIView` as follows:

```
from drf_multiple_model.views import MultipleModelAPIView

class TextAPIView(MultipleModelAPIView):
    queryList = [
        (Play.objects.all(),PlaySerializer),
        (Poem.objects.filter(style='Sonnet'),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},
                ....
            ],
    }
]
```

1.1.2 Configuration Options

Objectify

When using the results of the `MultipleModelAPIView`, it's often easier to parse the results as a JSON object rather than as an array. To facilitate this, `MultipleModelAPIView` has the `objectify` property, which when

set to `True` returns the results as an object. For example, the following:

```
from drf_multiple_model.views import MultipleModelAPIView

class TextAPIView(MultipleModelAPIView):
    objectify = True
    queryList = [
        (Play.objects.all(),PlaySerializer),
        (Poem.objects.filter(style='Sonnet'),PoemSerializer),
        ....
    ]
```

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},
            ....
        ],
}
```

Labels

By default, `MultipleModelAPIView` uses the model name as a label. If you want to use a custom label, you can add a third string attribute to the queryList tuples, like so:

```
from drf_multiple_model.views import MultipleModelAPIView

class TextAPIView(MultipleModelAPIView):
    queryList = [
        (Play.objects.all(),PlaySerializer,'plays'),
        (Poem.objects.filter(style='Sonnet'),PoemSerializer,'sonnets'),
        ....
    ]
```

which would return:

```
[
    {
        'plays': [
            {'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},
            ....
        ],
```

```
    }
]
```

Flat

Add the attribute `flat = True` to return a single JSON array with all of the objects mixed together. For example:

```
class TextAPIView(MultipleModelAPIView):
    flat = True

    queryList = [
        (Play.objects.all(),PlaySerializer,'plays'),
        (Poem.objects.filter(style='Sonnet'),PoemSerializer,'sonnets'),
        ....
    ]
```

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},
    ....
]
```

sorting_field

When using `flat=True`, 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_field` to your view:

```
class TextAPIView(MultipleModelAPIView):
    flat = True
    sorting_field = 'title'

    queryList = [
        (Play.objects.all(),PlaySerializer,'plays'),
        (Poem.objects.filter(style='Sonnet'),PoemSerializer,'sonnets'),
        ....
    ]
```

would return:

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

As with django field ordering, add '-' to the beginning of the field to enable reverse sorting. Setting `sorting_field='-title'` would sort the title fields in __descending__ order.

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`.

add_model_type

If no label is explicitly specified in your `queryList`, `MultipleModelAPIView` will use the model from each queryset a label. If you don't want any extra labeling and just want your data as is, set `add_model_type = False`:

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

    queryList = [
        (Play.objects.all(),PlaySerializer,'plays'),
        (Poem.objects.filter(style='Sonnet'),PoemSerializer,'sonnets'),
        ....
    ]
```

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},
        ....
    ]
]
```

This works with `flat = True` set as well – the `'type':'myModel'` won't be appended to each data point in that case. **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.

1.1.3 Mixin

If you want to combine `MultipleModelAPIView`'s `list()` function with other views, you can use the included `MultipleModelMixin` 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 MultipleModelAPIView
```

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 Filtering

1.3.1 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(MultipleModelAPIView):
    queryList = ((Play.objects.all(),PlaySerializer),
                 (Poem.objects.filter(style="Sonnet"),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'},
        ]
    }
]
```

1.3.2 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(MultipleModelAPIView):
    def get_queryList(self):
        title = self.request.query_params['play'].replace('-',' ')

        queryList = ((Play.objects.filter(title=title),PlaySerializer),
                     (Poem.objects.filter(style="Sonnet"),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. In order to use custom queryset filtering functions, you need to use the `Query` class, which is used internally by DRF Multiple Model to handle queryList information:

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

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

class FilterFnView(MultipleModelAPIView):
    queryList = (Query(Play.objects.all(), PlaySerializer, filter_fn=title_without_
↪letter),
                 (Poem.objects.all(), 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'}
        ]
    }
]
```

1.4 Pagination

If (and only if) `flat = True` on your view, **drf-multiple-model** supports some of Django Rest Framework's built-in pagination classes, including `PageNumberPagination` and `LimitOffsetPagination`. Implementatation might look like this:

```
class BasicPagination(pagination.PageNumberPagination):
    page_size = 5
    page_size_query_param = 'page_size'
    max_page_size = 10

class PageNumberPaginationView(MultipleModelAPIView):
    queryList = ((Play.objects.all(),PlaySerializer),
                 (Poem.objects.filter(style="Sonnet"),PoemSerializer))
    flat = True
    pagination_class = BasicPagination
```

which would return:

```
{
    'count': 6,
    '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},
            {'genre': 'Comedy', 'title': "The Tempest", 'pages': 250},
            {'title': 'Shall I compare thee to a summer's day?', 'stanzas': 1},
            {'title': 'As a decrepit father takes delight', 'stanzas': 1}
        ]
}
```

WARNING: In its currently implementation, pagination does NOT limit database queries, only the amount of information sent. Due to the way multiple models are serializer and combined, the entire queryList is evaluated and then combined before pagination happens. This means that Pagination in DRF Multiple Models is only useful for formating you API calls, not for creating more efficient queries.

1.5 ViewSets

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

```
from rest_framework import routers

from drf_multiple_model.viewsets import MultipleModelAPIViewSet

class TextAPIView(MultipleModelAPIViewSet):
```

```
    queryList = [
        (Play.objects.all(),PlaySerializer),
        (Poem.objects.filter(style='Sonnet'),PoemSerializer),
        ....
    ]

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

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

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

1.6 Release Notes

1.6.1 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

1.6.2 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)

1.6.3 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

1.6.4 1.6 (2016-02-23)

- Incorporated and expanded on reverse sort implemented by @schweickism

1.6.5 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)

1.6.6 1.3 (2015-12-10)

- Improper context passing bug fixed by @rbreu

1.6.7 1.2 (2015-11-11)

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

1.6.8 1.1 (2015-07-06)

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

1.6.9 1.0 (2016-06-29)

- initial release

1.7 Contributors

1.7.1 Project Maintainer and Founder

- Matt Nishi-Broach <matt@axiologue.org>

1.7.2 Contributors

- rbreu
- Paul Tiplady <Symmetric>
- schweickism
- ELIYAHUT123
- Malcolm Box <mbox>
- Evgen Osiptsov <evgenosiptsov>