drf-haystack documentation

Release 1.8.4

Inonit

Aug 12, 2018

Contents

Contents:

CHAPTER 1

Basic Usage

Usage is best demonstrated with some simple examples.

Warning: The code here is for demonstration purposes only! It might work (or not, I haven't tested), but as always, don't blindly copy code from the internet.

1.1 Examples

1.1.1 models.py

Let's say we have an app which contains a model *Location*. It could look something like this.

```
#
# models.py
#

from django.db import models
from haystack.utils.geo import Point


class Location(models.Model):

    latitude = models.FloatField()
    longitude = models.FloatField()
    address = models.CharField(max_length=100)
    city = models.CharField(max_length=30)
    zip_code = models.CharField(max_length=10)

    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
```

(continues on next page)

(continued from previous page)

```
    def __str__(self):
        return self.address

    @property
    def coordinates(self):
        return Point(self.longitude, self.latitude)
```

1.1.2 search_indexes.py

We would have to make a `search_indexes.py` file for haystack to pick it up.

```
#
# search_indexes.py
#

from django.utils import timezone
from haystack import indexes
from .models import Location


class LocationIndex(indexes.SearchIndex, indexes.Indexable):

    text = indexes.CharField(document=True, use_template=True)
    address = indexes.CharField(model_attr="address")
    city = indexes.CharField(model_attr="city")
    zip_code = indexes.CharField(model_attr="zip_code")

    autocomplete = indexes.EdgeNgramField()
    coordinates = indexes.LocationField(model_attr="coordinates")

    @staticmethod
    def prepare_autocomplete(obj):
        return " ".join((
            obj.address, obj.city, obj.zip_code
        ))

    def get_model(self):
        return Location

    def index_queryset(self, using=None):
        return self.get_model().objects.filter(
            created__lte=timezone.now()
        )
```

1.1.3 views.py

For a generic Django REST Framework view, you could do something like this.

```
#
# views.py
#
```

(continues on next page)

(continued from previous page)

```
from drf_haystack.serializers import HaystackSerializer
from drf_haystack.viewsets import HaystackViewSet

from .models import Location
from .search_indexes import LocationIndex


class LocationSerializer(HaystackSerializer):

    class Meta:
        # The `index_classes` attribute is a list of which search indexes
        # we want to include in the search.
        index_classes = [LocationIndex]

        # The `fields` contains all the fields we want to include.
        # NOTE: Make sure you don't confuse these with model attributes. These
        # fields belong to the search index!
        fields = [
            "text", "address", "city", "zip_code", "autocomplete"
        ]


class LocationSearchView(HaystackViewSet):

    # `index_models` is an optional list of which models you would like to include
    # in the search result. You might have several models indexed, and this provides
    # a way to filter out those of no interest for this particular view.
    # (Translates to `SearchQuerySet().models(*index_models)` behind the scenes.
    index_models = [Location]

    serializer_class = LocationSerializer
```

1.1.4 urls.py

Finally, hook up the views in your *urls.py* file.

Note: Make sure you specify the *base_name* attribute when wiring up the view in the router. Since we don't have any single *model* for the view, it is impossible for the router to automatically figure out the base name for the view.

```
#
# urls.py
#

from django.conf.urls import patterns, url, include
from rest_framework import routers

from .views import LocationSearchView
```

(continues on next page)

(continued from previous page)

```
router = routers.DefaultRouter()
router.register("location/search", LocationSearchView, base_name="location-
↪search")


urlpatterns = patterns(
    "",
    url(r"/api/v1/", include(router.urls)),
)
```

1.1.5 Query time!

Now that we have a view wired up, we can start using it. By default, the *HaystackViewSet* (which, more importantly inherits the *HaystackGenericAPIView* class) is set up to use the *HaystackFilter*. This is the most basic filter included and can do basic search by querying any of the field included in the *fields* attribute on the *Serializer*.

```
http://example.com/api/v1/location/search/?city=Oslo
```

Would perform a query looking up all documents where the *city field* equals "Oslo".

Field Lookups

You can also use field lookups in your field queries. See the Haystack field lookups documentation for info on what lookups are available. A query using a lookup might look like the following:

```
http://example.com/api/v1/location/search/?city__startswith=Os
```

This would perform a query looking up all documents where the *city field* started with "Os". You might get "Oslo", "Osaka", and "Ostrava".

Term Negation

You can also specify terms to exclude from the search results using the negation keyword. The default keyword is `not`, but is configurable via settings using `DRF_HAYSTACK_NEGATION_KEYWORD`.

```
http://example.com/api/v1/location/search/?city__not=Oslo
http://example.com/api/v1/location/search/?city__not__contains=Los
http://example.com/api/v1/location/search/?city__contains=Los&city__not__
↪contains=Angeles
```

CHAPTER 2

Autocomplete

Some kind of data such as ie. cities and zip codes could be useful to autocomplete. We have a Django REST Framework filter for performing autocomplete queries. It works quite like the regular *drf_haystack.filters.HaystackFilter* but *must* be run against an `NgramField` or `EdgeNgramField` in order to work properly. The main difference is that while the HaystackFilter performs a bitwise `OR` on terms for the same parameters, the *drf_haystack.filters.HaystackAutocompleteFilter* reduce query parameters down to a single filter (using an `SQ` object), and performs a bitwise `AND`.

By adding a list or tuple of `ignore_fields` to the serializer's Meta class, we can tell the REST framework to ignore these fields. This is handy in cases, where you do not want to serialize and transfer the content of a text, or n-gram index down to the client.

An example using the autocomplete filter might look something like this.

```
from drf_haystack.filters import HaystackAutocompleteFilter
from drf_haystack.serializers import HaystackSerializer
from drf_haystack.viewsets import HaystackViewSet

class AutocompleteSerializer(HaystackSerializer):

    class Meta:
        index_classes = [LocationIndex]
        fields = ["address", "city", "zip_code", "autocomplete"]
        ignore_fields = ["autocomplete"]

        # The `field_aliases` attribute can be used in order to alias a
        # query parameter to a field attribute. In this case a query like
        # /search/?q=oslo would alias the `q` parameter to the␣
↪`autocomplete`
        # field on the index.
        field_aliases = {
            "q": "autocomplete"
        }

class AutocompleteSearchViewSet(HaystackViewSet):
```

(continues on next page)

(continued from previous page)

```
    index_models = [Location]
    serializer_class = AutocompleteSerializer
    filter_backends = [HaystackAutocompleteFilter]
```

CHAPTER 3

GEO spatial locations

Some search backends support geo spatial searching. In order to take advantage of this we have the *drf_haystack.filters.HaystackGEOSpatialFilter*.

Note: The `HaystackGEOSpatialFilter` depends on `geopy` and `libgeos`. Make sure to install these libraries in order to use this filter.

```
$ pip install geopy
$ apt-get install libgeos-c1 (for debian based linux distros)
  or
$ brew install geos (for homebrew on OS X)
```

The geospatial filter is somewhat special, and for the time being, relies on a few assumptions.

1. The index model **must** to have a `LocationField` (See *search_indexes.py* for example). If your `LocationField` is named something other than `coordinates`, subclass the `HaystackGEOSpatialFilter` and make sure to set the *drf_haystack.filters.HaystackGEOSpatialFilter.point_field* to the name of the field.
2. The query **must** contain a `unit` parameter where the unit is a valid `UNIT` in the `django.contrib.gis.measure.Distance` class.
3. The query **must** contain a `from` parameter which is a comma separated longitude and latitude value.

You may also change the query param `from` by defining `DRF_HAYSTACK_SPATIAL_QUERY_PARAM` on your settings.

Example Geospatial view

```
class DistanceSerializer(serializers.Serializer):
    m = serializers.FloatField()
    km = serializers.FloatField()
```

(continues on next page)

(continued from previous page)

```
class LocationSerializer(HaystackSerializer):

    distance = SerializerMethodField()

    class Meta:
        index_classes = [LocationIndex]
        fields = ["address", "city", "zip_code"]

    def get_distance(self, obj):
        if hasattr(obj, "distance"):
            return DistanceSerializer(obj.distance, many=False).data


class LocationGeoSearchViewSet(HaystackViewSet):

    index_models = [Location]
    serializer_class = LocationSerializer
    filter_backends = [HaystackGEOSpatialFilter]
```

Example subclassing the HaystackGEOSpatialFilter

Assuming that your `LocationField` is named `location`.

```
from drf_haystack.filters import HaystackGEOSpatialFilter

class CustomHaystackGEOSpatialFilter(HaystackGEOSpatialFilter):
    point_field = 'location'


class LocationGeoSearchViewSet(HaystackViewSet):

    index_models = [Location]
    serializer_class = LocationSerializer
    filter_backends = [CustomHaystackGEOSpatialFilter]
```

Assuming the above code works as it should, we would be able to do queries like this:

```
/api/v1/search/?zip_code=0351&km=10&from=59.744076,10.152045
```

The above query would return all entries with zip_code 0351 within 10 kilometers from the location with latitude 59.744076 and longitude 10.152045.

CHAPTER 4

Highlighting

Haystack supports two kinds of Highlighting, and we support them both.

1. SearchQuerySet highlighting. This kind of highlighting requires a search backend which has support for highlighting, such as Elasticsearch or Solr.
2. Pure python highlighting. This implementation is somewhat slower, but enables highlighting support even if your search backend does not support it.

Note: The highlighter will always use the `document=True` field on your index to hightlight on. See examples below.

4.1 SearchQuerySet Highlighting

In order to add support for `SearchQuerySet().highlight()`, all you have to do is to add the *`drf_haystack.filters.HaystackHighlightFilter`* to the `filter_backends` in your view. The `HaystackSerializer` will check if your queryset has highlighting enabled, and render an additional `highlighted` field to your result. The highlighted words will be encapsulated in an `<em>words go here</em>` html tag.

Example view with highlighting enabled

```
from drf_haystack.viewsets import HaystackViewSet
from drf_haystack.filters import HaystackHighlightFilter

from .models import Person
from .serializers import PersonSerializer


class SearchViewSet(HaystackViewSet):
    index_models = [Person]
```

(continues on next page)

(continued from previous page)

```
    serializer_class = PersonSerializer
    filter_backends = [HaystackHighlightFilter]
```

Given a query like below

```
/api/v1/search/?firstname=jeremy
```

We would get a result like this

```
[
    {
        "lastname": "Rowland",
        "full_name": "Jeremy Rowland",
        "firstname": "Jeremy",
        "highlighted": "<em>Jeremy</em> Rowland\nCreated: May 19, 2015,␣
↪10:48 a.m.\nLast modified: May 19, 2015, 10:48 a.m.\n"
    },
    {
        "lastname": "Fowler",
        "full_name": "Jeremy Fowler",
        "firstname": "Jeremy",
        "highlighted": "<em>Jeremy</em> Fowler\nCreated: May 19, 2015,␣
↪10:48 a.m.\nLast modified: May 19, 2015, 10:48 a.m.\n"
    }
]
```

4.2 Pure Python Highlighting

This implementation make use of the haystack `Highlighter()` class. It is implemented as *`drf_haystack.serializers.HighlighterMixin`* mixin class, and must be applied on the `Serializer`. This is somewhat slower, but more configurable than the *`drf_haystack.filters.HaystackHighlightFilter`* filter class.

The Highlighter class will be initialized with the following default options, but can be overridden by changing any of the following class attributes.

```
highlighter_class = Highlighter
highlighter_css_class = "highlighted"
highlighter_html_tag = "span"
highlighter_max_length = 200
highlighter_field = None
```

The Highlighter class will usually highlight the `document_field` (the field marked `document=True` on your search index class), but this may be overridden by changing the `highlighter_field`.

You can of course also use your own `Highlighter` class by overriding the `highlighter_class = MyFancyHighLighter` class attribute.

Example serializer with highlighter support

```
from drf_haystack.serializers import HighlighterMixin, HaystackSerializer
```

(continues on next page)

(continued from previous page)

```
class PersonSerializer(HighlighterMixin, HaystackSerializer):

    highlighter_css_class = "my-highlighter-class"
    highlighter_html_tag = "em"

    class Meta:
        index_classes = [PersonIndex]
        fields = ["firstname", "lastname", "full_name"]
```

Response

```
[
    {
        "full_name": "Jeremy Rowland",
        "lastname": "Rowland",
        "firstname": "Jeremy",
        "highlighted": "<em class=\"my-highlighter-class\">Jeremy</em>␣
↪Rowland\nCreated: May 19, 2015, 10:48 a.m.\nLast modified: May 19, 2015,␣
↪10:48 a.m.\n"
    },
    {
        "full_name": "Jeremy Fowler",
        "lastname": "Fowler",
        "firstname": "Jeremy",
        "highlighted": "<em class=\"my-highlighter-class\">Jeremy</em>␣
↪Fowler\nCreated: May 19, 2015, 10:48 a.m.\nLast modified: May 19, 2015,␣
↪10:48 a.m.\n"
    }
]
```

CHAPTER 5

More Like This

Some search backends supports `More Like This` features. In order to take advantage of this, we have a mixin class *drf_haystack.mixins.MoreLikeThisMixin*, which will append a `more-like-this` detail route to the base name of the ViewSet. Lets say you have a router which looks like this:

```
router = routers.DefaultRouter()
router.register("search", viewset=SearchViewSet, base_name="search")  #␣
↪MLT name will be 'search-more-like-this'.

urlpatterns = patterns(
    "",
    url(r"^", include(router.urls))
)
```

The important thing here is that the `SearchViewSet` class inherits from the *drf_haystack.mixins.MoreLikeThisMixin* class in order to get the `more-like-this` route automatically added. The view name will be `{base_name}-more-like-this`, which in this case would be for example `search-more-like-this`.

5.1 Serializing the More Like This URL

In order to include the `more-like-this` url in your result you only have to add a `HyperlinkedIdentityField` to your serializer. Something like this should work okay.

Example serializer with More Like This

```
class SearchSerializer(HaystackSerializer):

    more_like_this = serializers.HyperlinkedIdentityField(view_name=
↪"search-more-like-this", read_only=True)

    class Meta:
```

(continues on next page)

(continued from previous page)

```
        index_classes = [PersonIndex]
        fields = ["firstname", "lastname", "full_name"]


class SearchViewSet(MoreLikeThisMixin, HaystackViewSet):
    index_models = [Person]
    serializer_class = SearchSerializer
```

Now, every result you render with this serializer will include a `more_like_this` field containing the url for similar results.

Example response

```
[
    {
        "full_name": "Jeremy Rowland",
        "lastname": "Rowland",
        "firstname": "Jeremy",
        "more_like_this": "http://example.com/search/5/more-like-this/"
    }
]
```

CHAPTER 6

Term Boost

Warning: BIG FAT WARNING

As far as I can see, the term boost functionality is implemented by the specs in the Haystack documentation, however it does not really work as it should!

When applying term boost, results are discarded from the search result, and not re-ordered by boost weight as they should. These are known problems and there exists open issues for them:

- https://github.com/inonit/drf-haystack/issues/21
- https://github.com/django-haystack/django-haystack/issues/1235
- https://github.com/django-haystack/django-haystack/issues/508

Please do not use this unless you really know what you are doing!

(And please let me know if you know how to fix it!)

Term boost is achieved on the SearchQuerySet level by calling `SearchQuerySet().boost()`. It is implemented as a *`drf_haystack.filters.HaystackBoostFilter`* filter backend. The `HaystackBoostFilter` does not perform any filtering by itself, and should therefore be combined with some other filter that does, for example the *`drf_haystack.filters.HaystackFilter`*.

```
from drf_haystack.filters import HaystackBoostFilter

class SearchViewSet(HaystackViewSet):
    ...
    filter_backends = [HaystackFilter, HaystackBoostFilter]
```

The filter expects the query string to contain a `boost` parameter, which is a comma separated string of the term to boost and the boost value. The boost value must be either an integer or float value.

Example query

```
/api/v1/search/?firstname=robin&boost=hood,1.1
```

The query above will first filter on `firstname=robin` and next apply a slight boost on any document containing the word `hood`.

Note: Term boost are only applied on terms existing in the `document field`.

CHAPTER 7

Faceting

Faceting is a way of grouping and narrowing search results by a common factor, for example we can group all results which are registered on a certain date. Similar to *More Like This*, the faceting functionality is implemented by setting up a special `^search/facets/$` route on any view which inherits from the *`drf_haystack.mixins.FacetMixin`* class.

Note: Options used for faceting is **not** portable across search backends. Make sure to provide options suitable for the backend you're using.

First, read the Haystack faceting docs and set up your search index for faceting.

7.1 Serializing faceted counts

Faceting is a little special in terms that it *does not* care about SearchQuerySet filtering. Faceting is performed by calling the `SearchQuerySet().facet(field, **options)` and `SearchQuerySet().date_facet(field, **options)` methods, which will apply facets to the SearchQuerySet. Next we need to call the `SearchQuerySet().facet_counts()` in order to retrieve a dictionary with all the *counts* for the faceted fields. We have a special *`drf_haystack.serializers.HaystackFacetSerializer`* class which is designed to serialize these results.

Tip: It *is* possible to perform faceting on a subset of the queryset, in which case you'd have to override the `get_queryset()` method of the view to limit the queryset before it is passed on to the `filter_facet_queryset()` method.

Any serializer subclassed from the `HaystackFacetSerializer` is expected to have a `field_options` dictionary containing a set of default options passed to `facet()` and `date_facet()`.

Facet serializer example

```
class PersonFacetSerializer(HaystackFacetSerializer):

    serialize_objects = False  # Setting this to True will serialize the
                               # queryset into an `objects` list. This
                               # is useful if you need to display the␣
↪faceted
                               # results. Defaults to False.
    class Meta:
        index_classes = [PersonIndex]
        fields = ["firstname", "lastname", "created"]
        field_options = {
            "firstname": {},
            "lastname": {},
            "created": {
                "start_date": datetime.now() - timedelta(days=3 * 365),
                "end_date": datetime.now(),
                "gap_by": "month",
                "gap_amount": 3
            }
        }
```

The declared `field_options` will be used as default options when faceting is applied to the queryset, but can be overridden by supplying query string parameters in the following format.

```
?firstname=limit:1&created=start_date:20th May 2014,gap_by:year
```

Each field can be fed options as `key:value` pairs. Multiple `key:value` pairs can be supplied and will be separated by the `view.lookup_sep` attribute (which defaults to comma). Any `start_date` and `end_date` parameters will be parsed by the python-dateutil parser() (which can handle most common date formats).

Note:

- The `HaystackFacetFilter` parses query string parameter options, separated with the `view.lookup_sep` attribute. Each option is parsed as `key:value` pairs where the `:` is a hardcoded separator. Setting the `view.lookup_sep` attribute to `":"` will raise an AttributeError.
- The date parsing in the `HaystackFacetFilter` does intentionally blow up if fed a string format it can't handle. No exception handling is done, so make sure to convert values to a format you know it can handle before passing it to the filter. Ie., don't let your users feed their own values in here ;)

Warning: Do *not* use the `HaystackFacetFilter` in the regular `filter_backends` list on the serializer. It will almost certainly produce errors or weird results. Faceting filters should go in the `facet_filter_backends` list.

Example serialized content

The serialized content will look a little different than the default Haystack faceted output. The top level items will *always* be **queries**, **fields** and **dates**, each containing a subset of fields matching the category. In the example below, we have faceted on the fields *firstname* and *lastname*, which will make

them appear under the **fields** category. We also have faceted on the date field *created*, which will show up under the **dates** category. Next, each faceted result will have a `text`, `count` and `narrow_url` attribute which should be quite self explaining.


```
{
  "queries": {},
  "fields": {
    "firstname": [
      {
        "text": "John",
        "count": 3,
        "narrow_url": "http://example.com/api/v1/search/facets/?
↪selected_facets=firstname_exact%3AJohn"
      },
      {
        "text": "Randall",
        "count": 2,
        "narrow_url": "http://example.com/api/v1/search/facets/?
↪selected_facets=firstname_exact%3ARandall"
      },
      {
        "text": "Nehru",
        "count": 2,
        "narrow_url": "http://example.com/api/v1/search/facets/?
↪selected_facets=firstname_exact%3ANehru"
      }
    ],
    "lastname": [
      {
        "text": "Porter",
        "count": 2,
        "narrow_url": "http://example.com/api/v1/search/facets/?
↪selected_facets=lastname_exact%3APorter"
      },
      {
        "text": "Odonnell",
        "count": 2,
        "narrow_url": "http://example.com/api/v1/search/facets/?
↪selected_facets=lastname_exact%3AOdonnell"
      },
      {
        "text": "Hood",
        "count": 2,
        "narrow_url": "http://example.com/api/v1/search/facets/?
↪selected_facets=lastname_exact%3AHood"
      }
    ]
  },
  "dates": {
    "created": [
      {
        "text": "2015-05-15T00:00:00",
        "count": 100,
        "narrow_url": "http://example.com/api/v1/search/facets/?
↪selected_facets=created_exact%3A2015-05-15+00%3A00%3A00"
      }
    ]
```


(continues on next page)

(continued from previous page)

```
    }
}
```

7.2 Serializing faceted results

When a `HaystackFacetSerializer` class determines what fields to serialize, it will check the `serialize_objects` class attribute to see if it is `True` or `False`. Setting this value to `True` will add an additional `objects` field to the serialized results, which will contain the results for the faceted `SearchQuerySet`. The results will by default be serialized using the view's `serializer_class`. If you wish to use a different serializer for serializing the results, set the *drf_haystack.mixins.FacetMixin.facet_objects_serializer_class* class attribute to whatever serializer you want to use, or override the *drf_haystack.mixins.FacetMixin.get_facet_objects_serializer_class()* method.

Example faceted results with paginated serialized objects

```
{
  "fields": {
    "firstname": [
      {"...": "..."}
    ],
    "lastname": [
      {"...": "..."}
    ]
  },
  "dates": {
    "created": [
      {"...": "..."}
    ]
  },
  "queries": {},
  "objects": {
    "count": 3,
    "next": "http://example.com/api/v1/search/facets/?page=2&selected_
↪facets=firstname_exact%3AJohn",
    "previous": null,
    "results": [
      {
        "lastname": "Baker",
        "firstname": "John",
        "full_name": "John Baker",
        "text": "John Baker\n"
      },
      {
        "lastname": "McClane",
        "firstname": "John",
        "full_name": "John McClane",
        "text": "John McClane\n"
      }
    ]
  }
}
```

7.3 Setting up the view

Any view that inherits the *drf_haystack.mixins.FacetMixin* will have a special action route added as `^<view-url>/facets/$`. This view action will not care about regular filtering but will by default use the `HaystackFacetFilter` to perform filtering.

Note: In order to avoid confusing the filtering mechanisms in Django Rest Framework, the `FacetMixin` class has a couple of hooks for dealing with faceting, namely:

- *drf_haystack.mixins.FacetMixin.facet_filter_backends* - A list of filter backends that will be used to apply faceting to the queryset. Defaults to :class:drf_haystack.filters.HaystackFacetFilter‘, which should be sufficient in most cases.
- *drf_haystack.mixins.FacetMixin.facet_serializer_class* - The *drf_haystack.serializers.HaystackFacetSerializer* instance that will be used for serializing the result.
- *drf_haystack.mixins.FacetMixin.facet_objects_serializer_class* - Optional. Set to the serializer class which should be used for serializing faceted objects. If not set, defaults to `self.serializer_class`.
- *drf_haystack.mixins.FacetMixin.filter_facet_queryset()* - Works exactly as the normal *drf_haystack.generics.HaystackGenericAPIView.filter_queryset()* method, but will only filter on backends in the `self.facet_filter_backends` list.
- *drf_haystack.mixins.FacetMixin.get_facet_serializer_class()* - Returns the `self.facet_serializer_class` class attribute.
- *drf_haystack.mixins.FacetMixin.get_facet_serializer()* - Instantiates and returns the *drf_haystack.serializers.HaystackFacetSerializer* class returned from *drf_haystack.mixins.FacetMixin.get_facet_serializer_class()* method.
- *drf_haystack.mixins.FacetMixin.get_facet_objects_serializer()* - Instantiates and returns the serializer class which will be used to serialize faceted objects.
- *drf_haystack.mixins.FacetMixin.get_facet_objects_serializer_class()* - Returns the `self.facet_objects_serializer_class`, or if not set, the `self.serializer_class`.

In order to set up a view which can respond to regular queries under ie `^search/$` and faceted queries under `^search/facets/$`, we could do something like this.

We can also change the query param text from `selected_facets` to our own choice like `params` or `p`. For this to make happen please provide `facet_query_params_text` attribute as shown in the example.

```
class SearchPersonViewSet(FacetMixin, HaystackViewSet):

    index_models = [MockPerson]

    # This will be used to filter and serialize regular queries as well
    # as the results if the `facet_serializer_class` has the
    # `serialize_objects = True` set.
```

(continues on next page)

(continued from previous page)

```
    serializer_class = SearchSerializer
    filter_backends = [HaystackHighlightFilter, HaystackAutocompleteFilter]

    # This will be used to filter and serialize faceted results
    facet_serializer_class = PersonFacetSerializer  # See example above!
    facet_filter_backends = [HaystackFacetFilter]   # This is the default␣
↪facet filter, and
                                                    # can be left out.
    facet_query_params_text = 'params' #Default is 'selected_facets'
```

7.4 Narrowing

As we have seen in the examples above, the `HaystackFacetSerializer` will add a `narrow_url` attribute to each result it serializes. Follow that link to narrow the search result.

The `narrow_url` is constructed like this:

- Read all query parameters from the request
- Get a list of `selected_facets`
- Update the query parameters by adding the current item to `selected_facets`
- Pop the *drf_haystack.serializers.HaystackFacetSerializer.paginate_by_param* parameter if any in order to always start at the first page if returning a paginated result.
- Return a `serializers.Hyperlink` with URL encoded query parameters

This means that for each drill-down performed, the original query parameters will be kept in order to make the `HaystackFacetFilter` happy. Additionally, all the previous `selected_facets` will be kept and applied to narrow the `SearchQuerySet` properly.

Example narrowed result

```
{
  "queries": {},
  "fields": {
    "firstname": [
      {
        "text": "John",
        "count": 1,
        "narrow_url": "http://example.com/api/v1/search/facets/?
↪selected_facets=firstname_exact%3AJohn&selected_facets=lastname_
↪exact%3AMcLaughlin"
      }
    ],
    "lastname": [
      {
        "text": "McLaughlin",
        "count": 1,
        "narrow_url": "http://example.com/api/v1/search/facets/?
↪selected_facets=firstname_exact%3AJohn&selected_facets=lastname_
↪exact%3AMcLaughlin"
      }
```

(continues on next page)

(continued from previous page)

```
    ]
  },
  "dates": {
    "created": [
      {
        "text": "2015-05-15T00:00:00",
        "count": 1,
        "narrow_url": "http://example.com/api/v1/search/facets/?
↪selected_facets=firstname_exact%3AJohn&selected_facets=lastname_
↪exact%3AMcLaughlin&selected_facets=created_exact%3A2015-05-15+00
↪%3A00%3A00"
      }
    ]
  }
}
```

CHAPTER 8

Permissions

Django REST Framework allows setting certain `permission_classes` in order to control access to views. The generic `HaystackGenericAPIView` defaults to `rest_framework.permissions.AllowAny` which enforce no restrictions on the views. This can be overridden on a per-view basis as you would normally do in a regular REST Framework APIView.

Note: Since we have no Django model or queryset, the following permission classes are *not* supported:

- `rest_framework.permissions.DjangoModelPermissions`
- `rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly`
- `rest_framework.permissions.DjangoObjectPermissions`

`POST`, `PUT`, `PATCH` and `DELETE` are not supported since Haystack Views are read-only. So if you are using the `rest_framework.permissions.IsAuthenticatedOrReadOnly` , this will act just as the `AllowAny` permission.

Example overriding permission classes

```
...
from rest_framework.permissions import IsAuthenticated

class SearchViewSet(HaystackViewSet):
    ...
    permission_classes = [IsAuthenticated]
```

CHAPTER 9

Multiple search indexes

So far, we have only used one class in the `index_classes` attribute of our serializers. However, you are able to specify a list of them. This can be useful when your search engine has indexed multiple models and you want to provide aggregate results across two or more of them. To use the default multiple index support, simply add multiple indexes the `index_classes` list

```
class PersonIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    firstname = indexes.CharField(model_attr="first_name")
    lastname = indexes.CharField(model_attr="last_name")

    def get_model(self):
        return Person

class PlaceIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    address = indexes.CharField(model_attr="address")

    def get_model(self):
        return Place

class ThingIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    name = indexes.CharField(model_attr="name")

    def get_model(self):
        return Thing

class AggregateSerializer(HaystackSerializer):

    class Meta:
        index_classes = [PersonIndex, PlaceIndex, ThingIndex]
        fields = ["firstname", "lastname", "address", "name"]


class AggregateSearchViewSet(HaystackViewSet):
```

(continues on next page)

(continued from previous page)

```
    serializer_class = AggregateSerializer
```

Note: The `AggregateSearchViewSet` class above omits the optional `index_models` attribute. This way results from all the models are returned.

The result from searches using multiple indexes is a list of objects, each of which contains only the fields appropriate to the model from which the result came. For instance if a search returned a list containing one each of the above models, it might look like the following:

```
[
    {
        "text": "John Doe",
        "firstname": "John",
        "lastname": "Doe"
    },
    {
        "text": "123 Doe Street",
        "address": "123 Doe Street"
    },
    {
        "text": "Doe",
        "name": "Doe"
    }
]
```

9.1 Declared fields

You can include field declarations in the serializer class like normal. Depending on how they are named, they will be treated as common fields and added to every result or as specific to results from a particular index.

Common fields are declared as you would any serializer field. Index-specific fields must be prefixed with "_<index class name>__". The following example illustrates this usage:

```
class AggregateSerializer(HaystackSerializer):
    extra = serializers.CharField()
    _ThingIndex__number = serializers.IntegerField()

    class Meta:
        index_classes = [PersonIndex, PlaceIndex, ThingIndex]
        fields = ["firstname", "lastname", "address", "name"]

    def get_extra(self):
        return "whatever"

    def get_ThingIndex__number(self):
        return 42
```

The results of a search might then look like the following:

```
[
    {
        "text": "John Doe",
        "firstname": "John",
        "lastname": "Doe",
        "extra": "whatever"
    },
    {
        "text": "123 Doe Street",
        "address": "123 Doe Street",
        "extra": "whatever"
    },
    {
        "text": "Doe",
        "name": "Doe",
        "extra": "whatever",
        "number": 42
    }
]
```

9.2 Multiple Serializers

Alternatively, you can specify a 'serializers' attribute on your Meta class to use a different serializer class for different indexes as show below:

```
class AggregateSearchSerializer(HaystackSerializer):
    class Meta:
        serializers = {
            PersonIndex: PersonSearchSerializer,
            PlaceIndex: PlaceSearchSerializer,
            ThingIndex: ThingSearchSerializer
        }
```

The `serializers` attribute is the important thing here, It's a dictionary with `SearchIndex` classes as keys and `Serializer` classes as values. Each result in the list of results from a search that contained items from multiple indexes would be serialized according to the appropriate serializer.

Warning: If a field name is shared across serializers, and one serializer overrides the field mapping, the overridden mapping will be used for *all* serializers. See the example below for more details.

```
from rest_framework import serializers

class PersonSearchSerializer(HaystackSerializer):
    # NOTE: This override will be used for both Person and Place objects.
    name = serializers.SerializerMethodField()

    class Meta:
        fields = ['name']

class PlaceSearchSerializer(HaystackSerializer):
    class Meta:
```

(continues on next page)

(continued from previous page)

```
        fields = ['name']

class AggregateSearchSerializer(HaystackSerializer):
    class Meta:
        serializers = {
            PersonIndex: PersonSearchSerializer,
            PlaceIndex: PlaceSearchSerializer,
            ThingIndex: ThingSearchSerializer
        }
```

CHAPTER 10

Tips'n Tricks

10.1 Reusing Model serializers

It may be useful to be able to use existing model serializers to return data from search requests in the same format as used elsewhere in your API. This can be done by modifying the `to_representation` method of your serializer to use the `instance.object` instead of the search result instance. As a convenience, a mixin class is provided that does just that.

class drf_haystack.serializers.**HaystackSerializerMixin**

An example using the mixin might look like the following:

```
class PersonSerializer(serializers.ModelSerializer):
    class Meta:
        model = Person
        fields = ("id", "firstname", "lastname")

class PersonSearchSerializer(HaystackSerializerMixin, PersonSerializer):
    class Meta(PersonSerializer.Meta):
        search_fields = ("text", )
```

The results from a search would then contain the fields from the `PersonSerializer` rather than fields from the search index.

Note: If your model serializer specifies a `fields` attribute in its Meta class, then the search serializer must specify a `search_fields` attribute in its Meta class if you intend to search on any search index fields that are not in the model serializer fields (e.g. 'text')

Warning: It should be noted that doing this will retrieve the underlying object which means a database hit. Thus, it will not be as performant as only retrieving data from the search index. If

performance is a concern, it would be better to recreate the desired data structure and store it in the search index.

10.2 Regular Search View

Sometimes you might not need all the bells and whistles of a `ViewSet`, but can do with a regular view. In such scenario you could do something like this.

```
#
# views.py
#

from rest_framework.mixins import ListModelMixin
from drf_haystack.generics import HaystackGenericAPIView


class SearchView(ListModelMixin, HaystackGenericAPIView):

    serializer_class = LocationSerializer

    def get(self, request, *args, **kwargs):
        return self.list(request, *args, **kwargs)


#
# urls.py
#

urlpatterns = (
    ...
    url(r'^search/', SearchView.as_view()),
    ...
)
```

You can also use *FacetMixin* or *MoreLikeThisMixin* in your regular views as well.

```
#
# views.py
#

from rest_framework.mixins import ListModelMixin
from drf_haystack.mixins import FacetMixin
from drf_haystack.generics import HaystackGenericAPIView


class SearchView(ListModelMixin, FacetMixin, HaystackGenericAPIView):
    index_models = [Project]
    serializer_class = ProjectListSerializer
    facet_serializer_class = ProjectListFacetSerializer

    pagination_class = BasicPagination
    permission_classes = (AllowAny,)
```

(continues on next page)

(continued from previous page)

```
    def get(self, request, *args, **kwargs):
        return self.facets(request, *args, **kwargs)
```

CHAPTER 11

API Docs

11.1 drf_haystack.fields

class drf_haystack.fields.**DRFHaystackFieldMixin**(***kwargs*)
Bases: object

bind(*field_name*, *parent*)
Initializes the field name and parent for the field instance. Called when a field is added to the parent serializer instance. Taken from DRF and modified to support drf_haystack multiple index functionality.

convert_field_name(*field_name*)

prefix_field_names = False

class drf_haystack.fields.**FacetDictField**(**args*, ***kwargs*)
Bases: rest_framework.fields.DictField

A special DictField which passes the key attribute down to the children's `to_representation()` in order to let the serializer know what field they're currently processing.

to_representation(*value*)
List of object instances -> List of dicts of primitive datatypes.

class drf_haystack.fields.**FacetListField**(**args*, ***kwargs*)
Bases: rest_framework.fields.ListField

The `FacetListField` just pass along the key derived from `FacetDictField`.

to_representation(*key*, *data*)
List of object instances -> List of dicts of primitive datatypes.

class drf_haystack.fields.**HaystackBooleanField**(***kwargs*)
Bases: *drf_haystack.fields.DRFHaystackFieldMixin*, rest_framework.fields.BooleanField

class drf_haystack.fields.**HaystackCharField**(***kwargs*)
Bases: *drf_haystack.fields.DRFHaystackFieldMixin*, rest_framework.fields.CharField

class drf_haystack.fields.**HaystackDateField**(***kwargs*)
Bases: *drf_haystack.fields.DRFHaystackFieldMixin*, rest_framework.fields.DateField

class drf_haystack.fields.**HaystackDateTimeField**(***kwargs*)
Bases: *drf_haystack.fields.DRFHaystackFieldMixin*, rest_framework.fields.DateTimeField

class drf_haystack.fields.**HaystackDecimalField**(***kwargs*)
Bases: *drf_haystack.fields.DRFHaystackFieldMixin*, rest_framework.fields.DecimalField

class drf_haystack.fields.**HaystackFloatField**(***kwargs*)
Bases: *drf_haystack.fields.DRFHaystackFieldMixin*, rest_framework.fields.FloatField

class drf_haystack.fields.**HaystackIntegerField**(***kwargs*)
Bases: *drf_haystack.fields.DRFHaystackFieldMixin*, rest_framework.fields.IntegerField

class drf_haystack.fields.**HaystackMultiValueField**(***kwargs*)
Bases: *drf_haystack.fields.DRFHaystackFieldMixin*, rest_framework.fields.ListField

11.2 drf_haystack.filters

class drf_haystack.filters.**BaseHaystackFilterBackend**
Bases: rest_framework.filters.BaseFilterBackend

A base class from which all Haystack filter backend classes should inherit.

apply_filters(*queryset*, *applicable_filters=None*, *applicable_exclusions=None*)
Apply constructed filters and excludes and return the queryset

Parameters

- **queryset** – queryset to filter
- **applicable_filters** – filters which are passed directly to queryset.filter()
- **applicable_exclusions** – filters which are passed directly to queryset.exclude()

:returns filtered queryset

build_filters(*view*, *filters=None*)
Get the query builder instance and return constructed query filters.

filter_queryset(*request*, *queryset*, *view*)
Return the filtered queryset.

get_query_builder(**args*, ***kwargs*)
Return the query builder class instance that should be used to build the query which is passed to the search engine backend.

get_query_builder_class()
Return the class to use for building the query. Defaults to using *self.query_builder_class*.

You may want to override this if you need to provide different methods of building the query sent to the search engine backend.

static get_request_filters(*request*)

process_filters(*filters*, *queryset*, *view*)
Convenient hook to do any post-processing of the filters before they are applied to the queryset.

query_builder_class = None

class drf_haystack.filters.**HaystackAutocompleteFilter**
Bases: *drf_haystack.filters.HaystackFilter*

A filter backend to perform autocomplete search.

Must be run against fields that are either *NgramField* or *EdgeNgramField*.

process_filters(*filters*, *queryset*, *view*)
Convenient hook to do any post-processing of the filters before they are applied to the queryset.

class drf_haystack.filters.**HaystackBoostFilter**
Bases: *drf_haystack.filters.BaseHaystackFilterBackend*

Filter backend for applying term boost on query time.

Apply by adding a comma separated `boost` query parameter containing a the term you want to boost and a floating point or integer for the boost value. The boost value is based around `1.0` as 100% - no boost.

Gives a slight increase in relevance for documents that include "banana":
/api/v1/search/?boost=banana,1.1

apply_filters(*queryset*, *applicable_filters=None*, *applicable_exclusions=None*)
Apply constructed filters and excludes and return the queryset

Parameters

- **queryset** – queryset to filter
- **applicable_filters** – filters which are passed directly to queryset.filter()
- **applicable_exclusions** – filters which are passed directly to queryset.exclude()

:returns filtered queryset

filter_queryset(*request*, *queryset*, *view*)
Return the filtered queryset.

query_builder_class
alias of *drf_haystack.query.BoostQueryBuilder*

query_param = 'boost'

class drf_haystack.filters.**HaystackFacetFilter**
Bases: *drf_haystack.filters.BaseHaystackFilterBackend*

Filter backend for faceting search results. This backend does not apply regular filtering.

Faceting field options can be set by using the `field_options` attribute on the serializer, and can be overridden by query parameters. Dates will be parsed by the `python-dateutil.parser()` which can handle most date formats.

Query parameters is parsed in the following format: ?field1=option1:value1,option2:value2&field2=option1:v

where each options `key:value` pair is separated by the `view.lookup_sep` attribute.

apply_filters(*queryset*, *applicable_filters=None*, *applicable_exclusions=None*)
Apply faceting to the queryset

filter_queryset(*request*, *queryset*, *view*)
Return the filtered queryset.

query_builder_class
alias of *drf_haystack.query.FacetQueryBuilder*

class drf_haystack.filters.**HaystackFilter**
Bases: *drf_haystack.filters.BaseHaystackFilterBackend*

A filter backend that compiles a haystack compatible filtering query.

default_operator()
and_(a, b) – Same as a & b.

query_builder_class
alias of *drf_haystack.query.FilterQueryBuilder*

class drf_haystack.filters.**HaystackGEOSpatialFilter**
Bases: *drf_haystack.filters.BaseHaystackFilterBackend*

A base filter backend for doing geo spatial filtering. If using this filter make sure to provide a *point_field* with the name of your the *LocationField* of your index.

We'll always do the somewhat slower but more accurate *dwithin* (radius) filter.

apply_filters(*queryset*, *applicable_filters=None*, *applicable_exclusions=None*)
Apply constructed filters and excludes and return the queryset

Parameters

- **queryset** – queryset to filter
- **applicable_filters** – filters which are passed directly to queryset.filter()
- **applicable_exclusions** – filters which are passed directly to queryset.exclude()

:returns filtered queryset

filter_queryset(*request*, *queryset*, *view*)
Return the filtered queryset.

point_field = 'coordinates'

query_builder_class
alias of *drf_haystack.query.SpatialQueryBuilder*

class drf_haystack.filters.**HaystackHighlightFilter**
Bases: *drf_haystack.filters.HaystackFilter*

A filter backend which adds support for `highlighting` on the SearchQuerySet level (the fast one). Note that you need to use a search backend which supports highlighting in order to use this.

This will add a `hightlighted` entry to your response, encapsulating the highlighted words in an *highlighted results* block.

filter_queryset(*request*, *queryset*, *view*)
Return the filtered queryset.

class drf_haystack.filters.**HaystackOrderingFilter**
Bases: `rest_framework.filters.OrderingFilter`

Some docstring here!

get_default_valid_fields(*queryset*, *view*, *context={}*)

get_valid_fields(*queryset*, *view*, *context={}*)

11.3 drf_haystack.generics

class drf_haystack.generics.**HaystackGenericAPIView**(***kwargs*)
Bases: `rest_framework.generics.GenericAPIView`

Base class for all haystack generic views.

document_uid_field = 'id'

filter_backends = [<class 'drf_haystack.filters.HaystackFilter'>]

filter_queryset(*queryset*)
Given a queryset, filter it with whichever filter backend is in use.

You are unlikely to want to override this method, although you may need to call it either from a list view, or from a custom *get_object* method if you want to apply the configured filtering backend to the default queryset.

get_object()
Fetch a single document from the data store according to whatever unique identifier is available for that document in the SearchIndex.

In cases where the view has multiple `index_models`, add a `model` query parameter containing a single *app_label.model* name to the request in order to override which model to include in the SearchQuerySet.

Example: /api/v1/search/42/?model=myapp.person

get_queryset(*index_models=[]*)
Get the list of items for this view. Returns `self.queryset` if defined and is a `self.object_class` instance.

@:param index_models: override *self.index_models*

index_models = []

load_all = False

lookup_sep = ','

object_class
alias of `haystack.query.SearchQuerySet`

query_object
alias of `haystack.backends.SQ`

11.4 drf_haystack.mixins

class drf_haystack.mixins.**FacetMixin**
Bases: `object`

Mixin class for supporting faceting on an API View.

facet_filter_backends = [<class 'drf_haystack.filters.HaystackFacetFilter'>]

facet_objects_serializer_class = None

facet_query_params_text = 'selected_facets'

facet_serializer_class = None

facets(*request*)
Sets up a list route for `faceted` results. This will add ie ^search/facets/$ to your existing ^search pattern.

filter_facet_queryset(*queryset*)
Given a search queryset, filter it with whichever facet filter backends in use.

get_facet_objects_serializer(**args*, ***kwargs*)
Return the serializer instance which should be used for serializing faceted objects.

get_facet_objects_serializer_class()
Return the class to use for serializing faceted objects. Defaults to using the views `self.serializer_class` if not `self.facet_objects_serializer_class` is set.

get_facet_serializer(**args*, ***kwargs*)
Return the facet serializer instance that should be used for serializing faceted output.

get_facet_serializer_class()
Return the class to use for serializing facets. Defaults to using `self.facet_serializer_class`.

class drf_haystack.mixins.**MoreLikeThisMixin**
Bases: `object`

Mixin class for supporting "more like this" on an API View.

more_like_this(*request*, *pk=None*)
Sets up a detail route for `more-like-this` results. Note that you'll need backend support in order to take advantage of this.

This will add ie. ^search/{pk}/more-like-this/$ to your existing ^search pattern.

11.5 drf_haystack.query

class drf_haystack.query.**BaseQueryBuilder**(*backend*, *view*)
Bases: object

Query builder base class.

build_query(**filters*)

Parameters **list[str]] filters** (*dict[str,*) – is an expanded QueryDict or a mapping of keys to a list of parameters.

static tokenize(*stream*, *separator*)
Tokenize and yield query parameter values.

Parameters

- **stream** – Input value
- **separator** – Character to use to separate the tokens.

Returns

class drf_haystack.query.**BoostQueryBuilder**(*backend*, *view*)
Bases: *drf_haystack.query.BaseQueryBuilder*

Query builder class for adding boost to queries.

build_query(**filters*)

Parameters **list[str]] filters** (*dict[str,*) – is an expanded QueryDict or

a mapping of keys to a list of parameters.

class drf_haystack.query.**FacetQueryBuilder**(*backend*, *view*)
Bases: *drf_haystack.query.BaseQueryBuilder*

Query builder class suitable for constructing faceted queries.

build_query(**filters*)
Creates a dict of dictionaries suitable for passing to the SearchQuerySet *facet*, *date_facet* or *query_facet* method. All key word arguments should be wrapped in a list.

Parameters

- **view** – API View
- **list[str]] filters** (*dict[str,*) – is an expanded QueryDict or a mapping

of keys to a list of parameters.

parse_field_options(**options*)
Parse the field options query string and return it as a dictionary.

class drf_haystack.query.**FilterQueryBuilder**(*backend*, *view*)
Bases: *drf_haystack.query.BaseQueryBuilder*

Query builder class suitable for doing basic filtering.

build_query(***filters*)
Creates a single SQ filter from querystring parameters that correspond to the SearchIndex fields that have been "registered" in *view.fields*.

Default behavior is to *OR* terms for the same parameters, and *AND* between parameters. Any querystring parameters that are not registered in *view.fields* will be ignored.

Parameters **list[str]] filters** (*dict[str,*) – is an expanded QueryDict or a mapping of keys to a list of

parameters.

class drf_haystack.query.**SpatialQueryBuilder**(*backend*, *view*)
Bases: *drf_haystack.query.BaseQueryBuilder*

Query builder class suitable for construction spatial queries.

build_query(***filters*)
Build queries for geo spatial filtering.

Expected query parameters are:

- a *unit=value* parameter where the unit is a valid UNIT in the *django.contrib.gis.measure.Distance* class.
- *from* which must be a comma separated latitude and longitude.

Example query: /api/v1/search/?km=10&from=59.744076,10.152045

Will perform a *dwithin* query within 10 km from the point with latitude 59.744076 and longitude 10.152045.

11.6 drf_haystack.serializers

class drf_haystack.serializers.**FacetFieldSerializer**(**args*, ***kwargs*)
Bases: rest_framework.serializers.Serializer

Responsible for serializing a faceted result.

get_count(*instance*)
Haystack facets are returned as a two-tuple (value, count). The count field should contain the faceted count.

get_narrow_url(*instance*)
Return a link suitable for narrowing on the current item.

get_paginate_by_param()
Returns the paginate_by_param for the (root) view paginator class. This is needed in order to remove the query parameter from faceted narrow urls.

If using a custom pagination class, this class attribute needs to be set manually.

get_text(*instance*)
Haystack facets are returned as a two-tuple (value, count). The text field should contain the faceted value.

parent_field

to_representation(*field*, *instance*)
Set the `parent_field` property equal to the current field on the serializer class, so that each field can query it to see what kind of attribute they are processing.

class drf_haystack.serializers.**HaystackFacetSerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)
Bases: `rest_framework.serializers.Serializer`

The `HaystackFacetSerializer` is used to serialize the `facet_counts()` dictionary results on a `SearchQuerySet` instance.

facet_dict_field_class
alias of *drf_haystack.fields.FacetDictField*

facet_field_serializer_class
alias of *FacetFieldSerializer*

facet_list_field_class
alias of *drf_haystack.fields.FacetListField*

facet_query_params_text

get_count(*queryset*)
Determine an object count, supporting either querysets or regular lists.

get_fields()
This returns a dictionary containing the top most fields, `dates`, `fields` and `queries`.

get_objects(*instance*)
Return a list of objects matching the faceted result.

paginate_by_param = None

serialize_objects = False

class drf_haystack.serializers.**HaystackSerializer**(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)
Bases: `rest_framework.serializers.Serializer`

A *HaystackSerializer* which populates fields based on which models that are available in the SearchQueryset.

get_fields()
Get the required fields for serializing the result.

multi_serializer_representation(*instance*)

to_representation(*instance*)
If we have a serializer mapping, use that. Otherwise, use standard serializer behavior Since we might be dealing with multiple indexes, some fields might not be valid for all results. Do not render the fields which don't belong to the search result.

class drf_haystack.serializers.**HaystackSerializerMeta**
Bases: `rest_framework.serializers.SerializerMetaclass`

Metaclass for the HaystackSerializer that ensures that all declared subclasses implemented a Meta.

class drf_haystack.serializers.**HaystackSerializerMixin**
Bases: object

This mixin can be added to a serializer to use the actual object as the data source for serialization rather than the data stored in the search index fields. This makes it easy to return data from search results in the same format as elsewhere in your API and reuse your existing serializers

to_representation(*instance*)

class drf_haystack.serializers.**HighlighterMixin**
Bases: object

This mixin adds support for `highlighting` (the pure python, portable version, not SearchQuerySet().highlight()). See Haystack docs for more info).

static get_document_field(*instance*)
Returns which field the search index has marked as it's *document=True* field.

get_highlighter()

highlighter_class
alias of `haystack.utils.highlighting.Highlighter`

highlighter_css_class = 'highlighted'

highlighter_field = None

highlighter_html_tag = 'span'

highlighter_max_length = 200

to_representation(*instance*)

class drf_haystack.serializers.**Meta**
Bases: type

Template for the HaystackSerializerMeta.Meta class.

exclude = ()

field_aliases = {}

field_options = {}

fields = ()

ignore_fields = ()

index_aliases = {}

index_classes = ()

search_fields = ()

serializers = ()

11.7 drf_haystack.utils

drf_haystack.utils.**merge_dict**(*a*, *b*)
Recursively merges and returns dict a with dict b. Any list values will be combined and returned sorted.

Parameters

- **a** – dictionary object
- **b** – dictionary object

Returns merged dictionary object

11.8 drf_haystack.viewsets

class drf_haystack.viewsets.**HaystackViewSet**(***kwargs*)

Bases: rest_framework.mixins.RetrieveModelMixin, rest_framework.mixins.ListModelMixin, rest_framework.viewsets.ViewSetMixin, *drf_haystack.generics.HaystackGenericAPIView*

The HaystackViewSet class provides the default `list()` and `retrieve()` actions with a haystack index as it's data source.

CHAPTER 12

About

Small library aiming to simplify using Haystack with Django REST Framework

CHAPTER 13

Features

Supported Python and Django versions:

- Python 2.7+ and Python 3.4+
- All supported versions of Django

CHAPTER 14

Installation

It's in the cheese shop!

```
$ pip install drf-haystack
```

CHAPTER 15

Requirements

- A Supported Django install
- Django REST Framework v3.2.0 and later
- Haystack v2.5 and later
- A supported search engine such as Solr, Elasticsearch, Whoosh, etc.
- Python bindings for the chosen backend (see below).
- (geopy and libgeos if you want to use geo spatial filtering)

15.1 Python bindings

You will also need to install python bindings for the search engine you'll use.

15.1.1 Elasticsearch

See haystack Elasticsearch docs for details

```
$ pip install elasticsearch<2.0.0           # For Elasticsearch 1.x
$ pip install elasticsearch>=2.0.0,<3.0.0   # For Elasticsearch 2.x
```

15.1.2 Solr

See haystack Solr docs for details.

```
$ pip install pysolr
```

15.1.3 Whoosh

See haystack Whoosh docs for details.

```
$ pip install whoosh
```

15.1.4 Xapian

See haystack Xapian docs for details.

CHAPTER 16

Contributors

This library has mainly been written by me while working at Inonit. I have also had some help from these amazing people! Thanks guys!

- See the full list of contributors.

CHAPTER 17

Changelog

17.1 v1.8.3

Release date: 2018-06-16

- Fixed issues with *__in=[...]* and *__range=[...]* filters. Closes #128.

17.2 v1.8.2

Release date: 2018-05-22

- Fixed issue with *_get_count* for DRF v3.8

17.3 v1.8.1

Release date: 2018-04-20

- Fixed errors in test suite which caused all tests to run on Elasticsearch 1.x

17.4 v1.8.0

Release date: 2018-04-16

This release was pulled because of critical errors in the test suite.

- Dropped support for Django v1.10.x and added support for Django v2.0.x
- Updated minimum Django REST Framework requirement to v3.7
- Updated minimum Haystack requirements to v2.8

17.5 v1.7.1rc2

Release date: 2018-01-30

- Fixes issues with building documentation.
- Fixed some minor typos in documentation.
- Dropped unittest2 in favor of standard lib unittest

17.6 v1.7.1rc1

Release date: 2018-01-06

- Locked Django versions in order to comply with Haystack requirements.
- Requires development release of Haystack (v2.7.1dev0).

17.7 v1.7.0

Release date: 2018-01-06 (Removed from pypi due to critical bugs)

- Bumping minimum support for Django to v1.10.
- Bumping minimum support for Django REST Framework to v1.6.0
- Adding support for Elasticsearch 2.x Haystack backend

17.8 v1.6.1

Release date: 2017-01-13

- Updated docs with correct name for libgeos-c1.
- Updated `.travis.yml` with correct name for libgeos-c1.
- Fixed an issue where queryset in the whould be evaluated if attribute is set but has no results, thus triggering the wrong clause in condition check. PR#88 closes #86.

17.9 v1.6.0

Release date: 2016-11-08

- Added Django 1.10 compatibility.
- Fixed multiple minor issues.

17.10 v1.6.0rc3

Release date: 2016-06-29

- Fixed #61. Introduction of custom serializers for serializing faceted objects contained a breaking change.

17.11 v1.6.0rc2

Release date: 2016-06-28

- Restructured and updated documentation
- Added support for using a custom serializer when serializing faceted objects.

17.12 v1.6.0rc1

Release date: 2016-06-24

Note: This release include breaking changes to the API

- Dropped support for Python 2.6, Django 1.5, 1.6 and 1.7
- Will follow Haystack's supported versions

- Removed deprecated `SQHighlighterMixin`.
- Removed redundant `BaseHaystackGEOSpatialFilter`. If name of `indexes.LocationField` needs to be changed, subclass the `HaystackGEOSpatialFilter` directly.
- **Reworked filters:**
  - More consistent naming of methods.
  - All filters follow the same logic for building and applying filters and exclusions.
  - All filter classes use a `QueryBuilder` class for working out validation and building queries which are to be passed to the `SearchQuerySet`.
  - Most filters does *not* inherit from `HaystackFilter` anymore (except `HaystackAutocompleteFilter` and `HaystackHighlightFilter`) and will no longer do basic field filtering. Filters should be properly placed in the `filter_backends` class attribute in their respective order to be applied. This solves issues where inherited filters responds to query parameters they should ignore.
- HaystackFacetSerializer `narrow_url` now returns an absolute url.
- HaystackFacetSerializer now properly serializes `MultiValueField` and `FacetMultiValueField` items as a JSON Array.

- `HaystackGenericAPIView.get_object()` optional `model` query parameter now requires a `app_label.model` instead of just the `model`.
- Extracted internal fields and serializer from the `HaystackFacetSerializer` in order to ease customization.
- `HaystackFacetSerializer` now supports all three builtin pagination classes, and a hook to support custom pagination classes.
- **Extracted the `more-like-this` detail route and `facets` list route from the generic HaystackView**

  - Support for `more-like-this` is available as a *`drf_haystack.mixins.MoreLikeThisMixin`* class.
  - Support for `facets` is available as a *`drf_haystack.mixins.FacetMixin`* class.

17.13 v1.5.6

Release date: 2015-12-02

- Fixed a bug where `ignore_fields` on `HaystackSerializer` did not work unless `exclude` evaluates to `True`.
- Removed `elasticsearch` from `install_requires`. Elasticsearch should not be a mandatory requirement, since it's useless if not using Elasticsearch as backend.

17.14 v1.5.5

Release date: 2015-10-31

- Added support for Django REST Framework 3.3.0 (Only for Python 2.7/Django 1.7 and above)
- Locked elasticsearch < 2.0.0 (See #29)

17.15 v1.5.4

Release date: 2015-10-08

- Added support for serializing faceted results. Closing #27.

17.16 v1.5.3

Release date: 2015-10-05

- Added support for *Faceting* (Github #11).

17.17 v1.5.2

Release date: 2015-08-23

- Proper support for *Multiple search indexes* (Github #22).
- Experimental support for *Term Boost* (This seems to have some issues upstreams, so unfortunately it does not really work as expected).
- Support for negate in filters.

17.18 v1.5.1

Release date: 2015-07-28

- Support for More Like This results (Github #10).
- Deprecated `SQHighlighterMixin` in favor of `HaystackHighlightFilter`.
- `HaystackGenericAPIView` now returns 404 for detail views if more than one entry is found (Github #19).

17.19 v1.5.0

Release date: 2015-06-29

- Added support for field lookups in queries, such as `field__contains=foobar`. Check out Haystack docs for details.
- Added default `permission_classes` on `HaystackGenericAPIView` in order to avoid crash when using global permission classes on REST Framework. See *Permissions* for details.

17.20 v1.4

Release date: 2015-06-15

- Fixed issues for Geo spatial filtering on django-haystack v2.4.x with Elasticsearch.
- A serializer class now accepts a list or tuple of `ignore_field` to bypass serialization.
- Added support for Highlighting.

17.21 v1.3

Release date: 2015-05-19

- `HaystackGenericAPIView().get_object()` now returns Http404 instead of an empty `SearchQueryset` if no object is found. This mimics the behaviour from `GenericAPIView().get_object()`.
- Removed hard dependencies for `geopy` and `libgeos` (See Github #5). This means that if you want to use the `HaystackGEOSpatialFilter`, you have to install these libraries manually.

17.22 v1.2

Release date: 2015-03-23

- Fixed `MissingDependency` error when using another search backend than Elasticsearch.
- Fixed converting distance to D object before filtering in HaystackGEOSpatialFilter.
- Added Python 3 classifier.

17.23 v1.1

Release date: 2015-02-16

- Full coverage (almost) test suite
- Documentation
- Beta release Development classifier

17.24 v1.0

Release date: 2015-02-14

- Initial release.

CHAPTER 18

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

G

H

I

L

M

O

P

Q

S

T