Filternaut

Release 0.0.1

February 13, 2017

Filternaut is a simple library which generates arbitrarily complex Django Q-objects from simple data. It fits nicely into situations where users provide data which you want to filter a queryset with.

Filternaut is indirectly a collection of fields, but it differs from Django forms in that you specify the logical relationships between fields, as well their names and types.

Filternaut is similar to Django Filters, but does not provide any machinery for rendering a user interface and does not inspect your models to autogenerate filters. However, Django Filters chains many calls to `.filter()` which means OR-like behaviour with more than one join. Filternaut supports either behaviour.

CHAPTER 1

Quickstart

```
# filters are combined using logical operators
filters = (
    DateTimeFilter('created_date', lookups=['lt', 'gt']) &
    CharFilter('username', lookups=['icontains']))

# they can read their values from anything dict-like
filters.parse(request.GET)

# and have a form-like 'validity pattern'.
if filters.valid:
    queryset = queryset.filter(filters.Q)
else:
    raise HttpResponseBadRequest(filters.errors)
```

CHAPTER 2

Installation

```
$ pip install django-filternaut
```

Filternaut is compatible with:

- Python 2.7 and 3.4
- Django 1.4 through to 1.9
- Django REST Framework 3.3 (optional)

CHAPTER 3

Documentation

3.1 Examples

3.1.1 Using a Simple Filter

Here Filternaut pulls a username from `user_data` and filters a User queryset with it. Filternaut is overkill for such a simple job, but hey, it's the first example.

```
from filternaut import Filter

user_data = {'username': 'nostromo'}
filters = Filter('username')
filters.parse(user_data)
print(User.objects.filter(filters.Q).query)
```

```
SELECT "auth_user"."id", ...
WHERE "auth_user"."username" = nostromo
```

3.1.2 Using Lookups

It's common to require comparisons such as greater than, less than, etc. against one field. You can provide a `lookups` argument to specify these.

```
from filternaut.filters import Filter

filters = Filter('username', lookups=['icontains', 'contains'])
filters.parse({'username__icontains': 'nostromo'})
print(User.objects.filter(filters.Q).query)
```

```
SELECT "auth_user"."id", ...
WHERE "auth_user"."username" LIKE %nostromo%...
```

The default comparison is 'exact', which is the equivalent of using no comparison affix when filtering with Django's ORM. To keep the 'exact' comparison when explicitly listing lookups, you must add `'exact'` to the list:

```
filters = Filter('last_login', lookups=['year', 'exact'])
```

If you provide only one lookup, the source data can optionally omit the lookup from the name of the key:

```
from filternaut.filters import Filter

filters = Filter('email', lookups=['iexact'])

# the lookup can be omitted...
filters.parse({'email': 'Sentence@Case.COM'})

# ...or included
filters.parse({'email__iexact': 'Sentence@Case.COM'})
```

3.1.3 Combining Several Filters

Now Filternaut is used to filter Users with either an email, a username, or a (first name, last name) pair.

```
from filternaut import Filter

user_data = {'email': 'user3@example.org', 'username': 'user3'}
filters = (
    Filter('email') |
    Filter('username') |
    (Filter('first_name') & Filter('last_name')))
filters.parse(user_data)
print(User.objects.filter(filters.Q).query)
```

```
SELECT "auth_user"."id", ...
WHERE ("auth_user"."email" = user3@example.org OR
       "auth_user"."username" = user3)
```

The same filters generate result in different SQL when given different input data:

```
user_data = {'first_name': 'Art', 'last_name': 'Vandelay'}
filters.parse(user_data)
print(User.objects.filter(filters.Q).query)
```

```
SELECT "auth_user"."id", ...
WHERE ("auth_user"."first_name" = Art AND
       "auth_user"."last_name" = Vandelay)
```

3.1.4 Filtering with Multiple Values

Using the lookup `__in` triggers the collection of multiple values from the source. If this is the case, the source must provide a `getlist` method. Django's QueryDict provides such a method.

```
from filternaut.filters import Filter
from django.utils.datastructures import MultiValueDict

filters = Filter(source='groups', dest='groups__name', lookups=['in'])
user_data = MultiValueDict({'groups': ['foo', 'bar']})
filters.parse(user_data)
print(User.objects.filter(filters.Q).query)
```

```
SELECT "auth_user"."id", ...
WHERE "auth_group"."name" IN (foo, bar)
```

If the source does not provide a `getlist` method Filternaut will fall back to a single value, but still deliver it as a list.

3.1.5 Mapping a Different Public API onto your Schema.

In this example, the source data's `last_transaction` value filters on the value of a field across a distant relationship. This allows you to simplify or hide the details of your schema, and to later change them without changing the names you expose.

```
from filternaut import Filter
filters = Filter(
    source='last_payment',
    dest='order__transaction__created_date',
    lookups=['lt', 'lte', 'gt', 'gte'])
```

3.1.6 Default Values for Filters

Filters can be given default values.

```
from filternaut import Filter
filters = Filter('is_active', default=True)
filters.parse({})  # no 'is_active'
print(User.objects.filter(filters.Q).query)
```

```
SELECT "auth_user"."id", ...
WHERE "auth_user"."is_active" = True
```

When a default value is used, lookups are ignored. Most combinations of lookups are mutually exclusive when comparing the same value. For example, filtering by `score__lt=3` and `score__gt=3` does not make any sense. Instead, a lookup of `exact` is used. `default_lookup` may be used to override this.

```
from datetime import datetime
from filternaut import Filter
filters = Filter('last_login', lookups=['lte', 'lt', 'gt', 'gte'],
                 default=datetime.now(), default_lookup='lte')
filters.parse({})  # no 'last_login'
print(User.objects.filter(filters.Q).query)
```

```
SELECT "auth_user"."id", ...
WHERE "auth_user"."last_login" <= ...
```

3.1.7 Requiring Certain Filters

If it's mandatory to provide certain filtering values, you can use the `required` argument. By default, filters are not required.

```
from filternaut import Filter

filters = Filter('username', required=True)
filters.parse({})  # no 'username'

print(filters.valid)
print(filters.errors)
```

```
False
{'username': ['This field is required']}
```

3.1.8 Conditional Requirements

Sometimes a field is required only when another is present. For example, you may say that a value for `last_name` must be accompanied by a value for `first_name`, whilst also allowing neither. Additionally, you may say that a value for `middle_name` requires values for `first_name` and `last_name`, but not vice versa. That example is illustrated here:

```
from filternaut import Filter, Optional

filters = (
   Optional(
      Filter('first_name', required=True),
      Filter('middle_name'),
      Filter('last_name', required=True)) &
   Filter('badgers_defeated'))

filters.parse({'first_name': 'Nostromo'})
print(filters.errors['__all__'])
print(filters.errors['last_name'])
```

```
['If any of first_name, last_name, middle_name are provided,
  all must be provided']
['This field is required']
```

Though the middle name filter isn't required itself, when present it triggers the requirement of the filters that are.

```
filters.parse({'middle_name': 'Boone'})
print(filters.errors['__all__'])
print(filters.errors['first_name'])
print(filters.errors['last_name'])
```

```
['If any of first_name, last_name, middle_name are provided,
  all must be provided']
['This field is required']
['This field is required']
```

When all required filters in an Optional group are present, the filters as a whole are valid.

```
filters.parse({
   'first_name': 'Nostromo',
   'last_name': 'Cheradenine'})
assert filters.valid
```

Similarly, when none of the filters in an Optional group are present, the filters as a whole are valid.

```
filters.parse({})
assert filters.valid
```

3.1.9 Validating and Transforming Source Data

Filters can be combined with `django.forms.fields.Field` instances to validate and transform source data.

```
from django.forms import DateTimeField
from filternaut.filters import FieldFilter

filters = FieldFilter('signup_date', field=DateTimeField())
filters.parse({'signup_date': 'potato'})
```

```
print(filters.valid)
print(filters.errors)
```

```
False
{'signup_date': ['Enter a valid date/time.']}
```

Instead of making you provide your own `field` argument, Filternaut pairs most of Django's Field subclasses with Filters. They can be used like so:

```
from filternaut.filters import ChoiceFilter

difficulties = [(4, 'Torment I'), (5, 'Torment II')]
filters = ChoiceFilter('difficulty', choices=difficulties)
filters.parse({'difficulty': 'foo'})

print(filters.valid)
print(filters.errors)
```

```
False
{'difficulty': ['Select a valid choice. foo is not ...']}
```

Filters wrapping fields which require special arguments to instantiate (e.g. `choices` in the example above) also require those arguments. That is, because ChoiceField needs `choices`, so does ChoiceFilter.

The full list of field-specific filter classes is:

- BooleanFilter
- CharFilter
- ChoiceFilter
- ComboFilter
- DateFilter
- DateTimeFilter
- DecimalFilter
- EmailFilter
- FilePathFilter
- FloatFilter
- GenericIPAddressFilter (Django 1.4 and greater)
- IPAddressFilter
- ImageFilter
- FieldFilter
- IntegerFilter
- MultiValueFilter
- MultipleChoiceFilter
- NullBooleanFilter
- RegexFilter
- SlugFilter

- SplitDateTimeFilter
- TimeFilter
- TypedChoiceFilter
- TypedMultipleChoiceFilter (Django 1.4 and greater)
- URLFilter

3.1.10 Django REST Framework

Using Filternaut with Django REST Framework is no more complicated than normal; simply connect, for example, a request's query parameters to a view's queryset:

```
from filternaut.filters import CharFilter, EmailFilter
from rest_framework import generics

class UserListView(generics.ListAPIView):
    model = User

    def filter_queryset(self, queryset):
        filters = CharFilter('username') | EmailFilter('email')
        filters.parse(self.request.query_params)
        queryset = super(UserListView, self).filter_queryset(queryset)
        return queryset.filter(filters.Q)
```

Filternaut also provides a Django REST Framework-compatible filter backend:

```
from filternaut.drf import FilternautBackend
from filternaut.filters import CharFilter, EmailFilter
from rest_framework import views

class MyView(views.APIView):
    filter_backends = (FilternautBackend, )
    filternaut_filters = CharFilter('username') | EmailFilter('email')
```

The attribute `filternaut_filters` should contain one or more Filter instances. Instead of an attribute, it can also be a callable which returns a list of filters, allowing the filters to vary on the current request:

```
from rest_framework import views

class MyView(views.APIView):
    filter_backends = (FilternautBackend, )

    def filternaut_filters(self, request):
        choices = ['guest', 'developer']
        if request.user.is_staff:
            choices.append('manager')
        return ChoiceFilter('account_type', choices=enumerate(choices))
```

3.2 API

class `filternaut.`**`FilterTree`**(*negate=False*, **args*, ***kwargs*)

FilterTrees instances are the result of ORing or ANDing Filter instances, or other FilterTree instances, together.

FilterTree provides a simple API for dealing with a set of filters. They may also be iterated over to gain access to the Filter instances directly.

Q
A Django "Q" object, built by combining input data with the filter definitions. Cannot be read before `parse()` has been called.

errors
A dictionary of errors met while parsing. The errors are keyed by their field's source value. Each key's value is a list of errors.

`errors` cannot be read before `parse()` has been called.

parse(*data*)
Ask all filters to look through `data` and thereby configure themselves.

valid
A boolean indicating whether all filters parsed successfully. Cannot be read before `parse()` has been called.

class `filternaut.`**Filter**(*dest*, ***kwargs*)
A Filter instance builds a django.db.models.Q object by pulling a value from arbitrary native data, e.g. a set of query params.

It can be ORed or ANDed together with other Filter instances in the same way Q objects can.

Q
A Django "Q" object, built by combining input data with this filter's definition. Cannot be read before `parse()` has been called.

clean(*value*)
Validate and normalise `value` for use in filtering. This implementation is a no-op; subclasses may do more work here.

default_dest_value_pair()
Construct a default dest/value pair to be used if no source data was found during parsing (and if this filter has default=True).

dest_value_pairs(*sourcevalue_pairs*)

Return two values:

- A list of two-tuples containing dests (ORM relation/field names) and their values.
- A dictionary of errors, keyed by the source which they originated from.

dict
A dictionary representation of this Filter's filter configuration. Cannot be read before `parse()` has been called.

errors
A dictionary of errors (keyed by source) listing any problems encountered during parsing. Typical entries include validation errors and failures to provide values where required. Raises a ValueError if `parse()` has not been called.

get_source_value(*key*, *data*, *many=False*)
Pull `key` from `data`.

When `many` is True, a list of values is returned. Otherwise, a single value is returned.

parse(*data*)
Look through the provided dict-like data for keys which match this Filter's source. This includes keys containg lookup affixes such as 'contains' or 'lte'.

Once this method has been called, the `errors`, `valid` and `Q` attributes become usable.

`source_dest_pairs`()

For each lookup in self.lookups, such as 'contains' or 'lte', combine it with this field's source and dest, returning e.g. (username__contains, account_name__contains)

If any lookup is None, that pair becomes (source, dest)

If there is only one lookup, two pairs are listed containing the source both with and without the lookup. This allows source data to omit the lookup from the key, e.g. providing 'email' to the filter Filter('email', lookups=['iexact']).

`source_value_pairs`(*data*)

Return a list of two-tuples containing valid sources for this filter – made by combining this filter's source and the various lookups – and their values, as pulled from the data handed to parse.

Sources with no found value are excluded.

`tree_class`

Filters combine into FilterTree instances

alias of `FilterTree`

`valid`

A boolean indicating whether this Filter registered any errors during parsing. Raises a ValueError if `parse()` has not been called.

class `filternaut.filters.`**`FieldFilter`**(*dest*, *field*, ***kwargs*)

FieldFilters use a django.forms.Field to clean their input value when generating a Q object.

This class is designed to be extended by subclasses which provide their own form-field instances. However, you could use it in combination with a custom field like so:

filter = FieldFilter(SpecialField(), dest='...')

class `filternaut.drf.`**`FilternautBackend`**

FilternautBackend is a "custom generic filtering backend" for Django REST framework: http://www.django-rest-framework.org/api-guide/filtering/#custom-generic-filtering

It allows straightforward filtering of a view's queryset using request parameters.

`filter_attr` **= 'filternaut_filters'**

The host view must define filters at this attribute.

`filter_queryset`(*request*, *queryset*, *view*)

Decide whether to apply the filters defined by `view.filternaut_filters` on the argued queryset. If the filters parse correctly, `is_valid` is called. If not, `is_invalid` is called

`is_invalid`(*request*, *queryset*, *filters*)

Raise a ParseError containing the filter errors. This results in a 400 Bad Request whose body details those errors. Provided for convenience when subclassing.

`is_valid`(*request*, *queryset*, *filters*)

Apply `filters` to `queryset`. Provided for convenience when subclassing.

3.3 Tests

First, install the extra dependencies:

```
$ pip install -r requirements/maintainer.txt
```

You can run the test suite in a specific environment via tox. In this example, against Python 2.7 and Django 1.4. (Hint: try `tox -l` for a full list).

```
$ tox -e py27-dj14
```

The full set of environments can be run by providing no arguments to tox. If it's the first time, consider opening a beer.

```
$ tox
```

Finally, you can run the test suite without tox if you prefer. You will need to manually install additional dependencies if you choose to do this.

```
$ nosetests tests
```

CHAPTER 4

Changelog

4.1 0.0.7

- Fix BooleanFilter rejecting falsish values
- Add support for Django 1.9
- Remove support for Django REST Framework 2.x and 3.1; add support for 3.3