django-filter Documentation

Release 1.0.4

Alex Gaynor and others.

Jul 16, 2017

User Guide

Django-filter is a generic, reusable application to alleviate writing some of the more mundane bits of view code. Specifically, it allows users to filter down a queryset based on a model's fields, displaying the form to let them do this.

CHAPTER 1

Installation

Django-filter can be installed from PyPI with tools like `pip`:

```
$ pip install django-filter
```

Then add `'django_filters'` to your `INSTALLED_APPS`.

```
INSTALLED_APPS = [
    ...
    'django_filters',
]
```

Requirements

Django-filter is tested against all supported versions of Python and Django, as well as the latest version of Django REST Framework (DRF).

- **Python**: 2.7, 3.3, 3.4, 3.5
- **Django**: 1.8, 1.9, 1.10, 1.11
- **DRF**: 3.5

CHAPTER 2

Getting Started

Django-filter provides a simple way to filter down a queryset based on parameters a user provides. Say we have a `Product` model and we want to let our users filter which products they see on a list page.

Note: If you're using django-filter with Django Rest Framework, it's recommended that you read the integration docs after this guide.

The model

Let's start with our model:

```
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=255)
    price = models.DecimalField()
    description = models.TextField()
    release_date = models.DateField()
    manufacturer = models.ForeignKey(Manufacturer)
```

The filter

We have a number of fields and we want to let our users filter based on the name, the price or the release_date. We create a `FilterSet` for this:

```
import django_filters

class ProductFilter(django_filters.FilterSet):
    name = django_filters.CharFilter(lookup_expr='iexact')
```

```
    class Meta:
        model = Product
        fields = ['price', 'release_date']
```

As you can see this uses a very similar API to Django's `ModelForm`. Just like with a `ModelForm` we can also override filters, or add new ones using a declarative syntax.

Declaring filters

The declarative syntax provides you with the most flexibility when creating filters, however it is fairly verbose. We'll use the below example to outline the *core filter arguments* on a `FilterSet`:

```
class ProductFilter(django_filters.FilterSet):
    price = django_filters.NumberFilter()
    price__gt = django_filters.NumberFilter(name='price', lookup_expr='gt')
    price__lt = django_filters.NumberFilter(name='price', lookup_expr='lt')

    release_year = django_filters.NumberFilter(name='release_date', lookup_expr='year
↪')
    release_year__gt = django_filters.NumberFilter(name='release_date', lookup_expr=
↪'year__gt')
    release_year__lt = django_filters.NumberFilter(name='release_date', lookup_expr=
↪'year__lt')

    manufacturer__name = django_filters.CharFilter(lookup_expr='icontains')

    class Meta:
        model = Product
```

There are two main arguments for filters:

- `name`: The name of the model field to filter on. You can traverse "relationship paths" using Django's `__` syntax to filter fields on a related model. ex, `manufacturer__name`.
- `lookup_expr`: The field lookup to use when filtering. Django's `__` syntax can again be used in order to support lookup transforms. ex, `year__gte`.

Together, the field `name` and `lookup_expr` represent a complete Django lookup expression. A detailed explanation of lookup expressions is provided in Django's lookup reference. django-filter supports expressions containing both transforms and a final lookup for version 1.9 of Django and above. For Django version 1.8, transformed expressions are not supported.

Generating filters with Meta.fields

The FilterSet Meta class provides a `fields` attribute that can be used for easily specifying multiple filters without significant code duplication. The base syntax supports a list of multiple field names:

```
import django_filters

class ProductFilter(django_filters.FilterSet):
    class Meta:
        model = Product
        fields = ['price', 'release_date']
```

The above generates 'exact' lookups for both the 'price' and 'release_date' fields.

Additionally, a dictionary can be used to specify multiple lookup expressions for each field:

```
import django_filters

class ProductFilter(django_filters.FilterSet):
    class Meta:
        model = Product
        fields = {
            'price': ['lt', 'gt'],
            'release_date': ['exact', 'year__gt'],
        }
```

The above would generate 'price__lt', 'price__gt', 'release_date', and 'release_date__year__gt' filters.

Note: The filter lookup type 'exact' is an implicit default and therefore never added to a filter name. In the above example, the release date's exact filter is 'release_date', not 'release_date__exact'.

Items in the `fields` sequence in the `Meta` class may include "relationship paths" using Django's ___ syntax to filter on fields on a related model:

```
class ProductFilter(django_filters.FilterSet):
    class Meta:
        model = Product
        fields = ['manufacturer__country']
```

Overriding default filters

Like `django.contrib.admin.ModelAdmin`, it is possible to override default filters for all the models fields of the same kind using `filter_overrides` on the `Meta` class:

```
class ProductFilter(django_filters.FilterSet):

    class Meta:
        model = Product
        fields = {
            'name': ['exact'],
            'release_date': ['isnull'],
        }
        filter_overrides = {
            models.CharField: {
                'filter_class': django_filters.CharFilter,
                'extra': lambda f: {
                    'lookup_expr': 'icontains',
                },
            },
            models.BooleanField: {
                'filter_class': django_filters.BooleanFilter,
                'extra': lambda f: {
                    'widget': forms.CheckboxInput,
                },
            },
        }
```

Request-based filtering

The `FilterSet` may be initialized with an optional `request` argument. If a request object is passed, then you may access the request during filtering. This allows you to filter by properties on the request, such as the currently logged-in user or the `Accepts-Languages` header.

Note: It is not guaranteed that a *request* will be provied to the *FilterSet* instance. Any code depending on a request should handle the *None* case.

Filtering the primary `.qs`

To filter the primary queryset by the `request` object, simply override the `FilterSet.qs` property. For example, you could filter blog articles to only those that are published and those that are owned by the logged-in user (presumably the author's draft articles).

```
class ArticleFilter(django_filters.FilterSet):

    class Meta:
        model = Article
        fields = [...]

    @property
    def qs(self):
        parent = super(ArticleFilter, self).qs
        author = getattr(self.request, 'user', None)

        return parent.filter(is_published=True) \
            | parent.filter(author=author)
```

Filtering the related queryset for `ModelChoiceFilter`

The `queryset` argument for `ModelChoiceFilter` and `ModelMultipleChoiceFilter` supports callable behavior. If a callable is passed, it will be invoked with the `request` as its only argument. This allows you to perform the same kinds of request-based filtering without resorting to overriding `FilterSet.__init__`.

```
def departments(request):
    if request is None:
        return Department.objects.none()

    company = request.user.company
    return company.department_set.all()

class EmployeeFilter(filters.FilterSet):
    department = filters.ModelChoiceFilter(queryset=departments)
    ...
```

Customize filtering with `Filter.method`

You can control the behavior of a filter by specifying a `method` to perform filtering. View more information in the *method reference*. Note that you may access the filterset's properties, such as the `request`.

```
class F(django_filters.FilterSet):
    username = CharFilter(method='my_custom_filter')

    class Meta:
        model = User
        fields = ['username']

    def my_custom_filter(self, queryset, name, value):
        return queryset.filter(**{
            name: value,
        })
```

The view

Now we need to write a view:

```
def product_list(request):
    f = ProductFilter(request.GET, queryset=Product.objects.all())
    return render(request, 'my_app/template.html', {'filter': f})
```

If a queryset argument isn't provided then all the items in the default manager of the model will be used.

If you want to access the filtered objects in your views, for example if you want to paginate them, you can do that. They are in f.qs

The URL conf

We need a URL pattern to call the view:

```
url(r'^list$', views.product_list)
```

The template

And lastly we need a template:

```
{% extends "base.html" %}

{% block content %}
    <form action="" method="get">
        {{ filter.form.as_p }}
        <input type="submit" />
    </form>
    {% for obj in filter.qs %}
        {{ obj.name }} - ${{ obj.price }}<br />
    {% endfor %}
{% endblock %}
```

And that's all there is to it! The `form` attribute contains a normal Django form, and when we iterate over the `FilterSet.qs` we get the objects in the resulting queryset.

Generic view & configuration

In addition to the above usage there is also a class-based generic view included in django-filter, which lives at `django_filters.views.FilterView`. You must provide either a `model` or `filterset_class` argument, similar to `ListView` in Django itself:

```
# urls.py
from django.conf.urls import url
from django_filters.views import FilterView
from myapp.models import Product

urlpatterns = [
    url(r'^list/$', FilterView.as_view(model=Product)),
]
```

If you provide a `model` optionally you can set `filter_fields` to specify a list or a tuple of the fields that you want to include for the automatic construction of the filterset class.

You must provide a template at `<app>/<model>_filter.html` which gets the context parameter `filter`. Additionally, the context will contain `object_list` which holds the filtered queryset.

A legacy functional generic view is still included in django-filter, although its use is deprecated. It can be found at `django_filters.views.object_filter`. You must provide the same arguments to it as the class based view:

```
# urls.py
from django.conf.urls import url
from django_filters.views import object_filter
from myapp.models import Product

urlpatterns = [
    url(r'^list/$', object_filter, {'model': Product}),
]
```

The needed template and its context variables will also be the same as the class-based view above.

CHAPTER 3

Integration with DRF

Integration with Django Rest Framework is provided through a DRF-specific `FilterSet` and a filter backend. These may be found in the `rest_framework` sub-package.

Quickstart

Using the new `FilterSet` simply requires changing the import path. Instead of importing from `django_filters`, import from the `rest_framework` sub-package.

```
from django_filters import rest_framework as filters

class ProductFilter(filters.FilterSet):
    ...
```

Your view class will also need to add `DjangoFilterBackend` to the `filter_backends`.

```
from django_filters import rest_framework as filters

class ProductList(generics.ListAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    filter_backends = (filters.DjangoFilterBackend,)
    filter_fields = ('category', 'in_stock')
```

If you want to use the django-filter backend by default, add it to the `DEFAULT_FILTER_BACKENDS` setting.

```
# settings.py
INSTALLED_APPS = [
    ...
    'rest_framework',
    'django_filters',
]
```

```
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS': (
        'django_filters.rest_framework.DjangoFilterBackend',
        ...
    ),
}
```

Adding a FilterSet with `filter_class`

To enable filtering with a `FilterSet`, add it to the `filter_class` parameter on your view class.

```
from rest_framework import generics
from django_filters import rest_framework as filters
from myapp import Product


class ProductFilter(filters.FilterSet):
    min_price = django_filters.NumberFilter(name="price", lookup_expr='gte')
    max_price = django_filters.NumberFilter(name="price", lookup_expr='lte')

    class Meta:
        model = Product
        fields = ['category', 'in_stock', 'min_price', 'max_price']


class ProductList(generics.ListAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    filter_backends = (filters.DjangoFilterBackend,)
    filter_class = ProductFilter
```

Using the `filter_fields` shortcut

You may bypass creating a `FilterSet` by instead adding `filter_fields` to your view class. This is equivalent to creating a FilterSet with just *Meta.fields*.

```
from rest_framework import generics
from django_filters import rest_framework as filters
from myapp import Product


class ProductList(generics.ListAPIView):
    queryset = Product.objects.all()
    filter_backends = (filters.DjangoFilterBackend,)
    filter_fields = ('category', 'in_stock')


# Equivalent FilterSet:
class ProductFilter(filters.FilterSet):
    class Meta:
        model = Product
        fields = ('category', 'in_stock')
```

Schema Generation with Core API

The backend class integrates with DRF's schema generation by implementing `get_schema_fields()`. This is automatically enabled when Core API is installed. Schema generation usually functions seamlessly, however the implementation does expect to invoke the view's `get_queryset()` method. There is a caveat in that views are artificially constructed during schema generation, so the `args` and `kwargs` attributes will be empty. If you depend on arguments parsed from the URL, you will need to handle their absence in `get_queryset()`.

For example, your get queryset method may look like this:

```
class IssueViewSet(views.ModelViewSet):
    queryset = models.Issue.objects.all()

    def get_project(self):
        return models.Project.objects.get(pk=self.kwargs['project_id'])

    def get_queryset(self):
        project = self.get_project()

        return self.queryset \
            .filter(project=project) \
            .filter(author=self.request.user)
```

This could be rewritten like so:

```
class IssueViewSet(views.ModelViewSet):
    queryset = models.Issue.objects.all()

    def get_project(self):
        try:
            return models.Project.objects.get(pk=self.kwargs['project_id'])
        except models.Project.DoesNotExist:
            return None

    def get_queryset(self):
        project = self.get_project()

        if project is None:
            return self.queryset.none()

        return self.queryset \
            .filter(project=project) \
            .filter(author=self.request.user)
```

Or more simply as:

```
class IssueViewSet(views.ModelViewSet):
    queryset = models.Issue.objects.all()

    def get_queryset(self):
        # project_id may be None
        return self.queryset \
            .filter(project_id=self.kwargs.get('project_id')) \
            .filter(author=self.request.user)
```

Crispy Forms

If you are using DRF's browsable API or admin API you may also want to install `django-crispy-forms`, which will enhance the presentation of the filter forms in HTML views, by allowing them to render Bootstrap 3 HTML. Note that this isn't actively supported, although pull requests for bug fixes are welcome.

```
pip install django-crispy-forms
```

With crispy forms installed and added to Django's `INSTALLED_APPS`, the browsable API will present a filtering control for `DjangoFilterBackend`, like so:

Additional `FilterSet` Features

The following features are specific to the rest framework FilterSet:

- `BooleanFilter`'s use the API-friendly `BooleanWidget`, which accepts lowercase `true`/`false`.
- Filter generation uses `IsoDateTimeFilter` for datetime model fields.
- Raised `ValidationError`'s are reraised as their DRF equivalent. This behavior is useful when setting FilterSet strictness to `STRICTNESS.RAISE_VALIDATION_ERROR`.

CHAPTER 4

Tips and Solutions

Common problems for declared filters

Below are some of the common problem that occur when declaring filters. It is recommended that you read this as it provides a more complete understanding of how filters work.

Filter `name` and `lookup_expr` not configured

While `name` and `lookup_expr` are optional, it is recommended that you specify them. By default, if `name` is not specified, the filter's name on the filterset class will be used. Additionally, `lookup_expr` defaults to `exact`. The following is an example of a misconfigured price filter:

```
class ProductFilter(django_filters.FilterSet):
    price__gt = django_filters.NumberFilter()
```

The filter instance will have a field name of `price__gt` and an `exact` lookup type. Under the hood, this will incorrectly be resolved as:

```
Produce.objects.filter(price__gt__exact=value)
```

The above will most likely generate a `FieldError`. The correct configuration would be:

```
class ProductFilter(django_filters.FilterSet):
    price__gt = django_filters.NumberFilter(name='price', lookup_expr='gt')
```

Missing `lookup_expr` for text search filters

It's quite common to forget to set the lookup expression for `CharField` and `TextField` and wonder why a search for "foo" does not return results for "foobar". This is because the default lookup type is `exact`, but you probably want to perform an `icontains` lookup.

Filter and lookup expression mismatch (in, range, isnull)

It's not always appropriate to directly match a filter to its model field's type, as some lookups expect different types of values. This is a commonly found issue with `in`, `range`, and `isnull` lookups. Let's look at the following product model:

```
class Product(models.Model):
    category = models.ForeignKey(Category, null=True)
```

Given that `category` is optional, it's reasonable to want to enable a search for uncategorized products. The following is an incorrectly configured `isnull` filter:

```
class ProductFilter(django_filters.FilterSet):
    uncategorized = django_filters.NumberFilter(name='category', lookup_expr='isnull')
```

So what's the issue? While the underlying column type for `category` is an integer, `isnull` lookups expect a boolean value. A `NumberFilter` however only validates numbers. Filters are not *'expression aware'* and won't change behavior based on their `lookup_expr`. You should use filters that match the data type of the lookup expression *instead* of the data type underlying the model field. The following would correctly allow you to search for both uncategorized products and products for a set of categories:

```
class NumberInFilter(django_filters.BaseInFilter, django_filters.NumberFilter):
    pass

class ProductFilter(django_filters.FilterSet):
    categories = NumberInFilter(name='category', lookup_expr='in')
    uncategorized = django_filters.BooleanFilter(name='category', lookup_expr='isnull
↪')
```

More info on constructing `in` and `range` csv *filters*.

Filtering by empty values

There are a number of cases where you may need to filter by empty or null values. The following are some common solutions to these problems:

Filtering by null values

As explained in the above "Filter and lookup expression mismatch" section, a common problem is how to correctly filter by null values on a field.

Solution 1: Using a `BooleanFilter` with `isnull`

Using `BooleanFilter` with an `isnull` lookup is a builtin solution used by the FilterSet's automatic filter generation. To do this manually, simply add:

```
class ProductFilter(django_filters.FilterSet):
    uncategorized = django_filters.BooleanFilter(name='category', lookup_expr='isnull
↪')
```

Note: Remember that the filter class is validating the input value. The underlying type of the mode field is not relevant here.

You may also reverse the logic with the `exclude` parameter.

```
class ProductFilter(django_filters.FilterSet):
    has_category = django_filters.BooleanFilter(name='category', lookup_expr='isnull',
↪ exclude=True)
```

Solution 2: Using `ChoiceFilter`'s null choice

If you're using a ChoiceFilter, you may also filter by null values by enabling the `null_label` parameter. More details in the `ChoiceFilter` reference *docs*.

```
class ProductFilter(django_filters.FilterSet):
    category = django_filters.ModelChoiceFilter(
        name='category', lookup_expr='isnull',
        null_label='Uncategorized',
        queryset=Category.objects.all(),
    )
```

Solution 3: Combining fields w/ `MultiValueField`

An alternative approach is to use Django's `MultiValueField` to manually add in a `BooleanField` to handle null values. Proof of concept: https://github.com/carltongibson/django-filter/issues/446

Filtering by an empty string

It's not currently possible to filter by an empty string, since empty values are interpreted as a skipped filter.

GET http://localhost/api/my-model?myfield=

Solution 1: Magic values

You can override the `filter()` method of a filter class to specifically check for magic values. This is similar to the `ChoiceFilter`'s null value handling.

GET http://localhost/api/my-model?myfield=EMPTY

```
class MyCharFilter(filters.CharFilter):
    empty_value = 'EMPTY'

    def filter(self, qs, value):
        if value != self.empty_value:
            return super(MyCharFilter, self).filter(qs, value)

        qs = self.get_method(qs)(**{'%s__%s' % (self.name, self.lookup_expr): ""})
        return qs.distinct() if self.distinct else qs
```

Solution 2: Empty string filter

It would also be possible to create an empty value filter that exhibits the same behavior as an `isnull` filter.

GET http://localhost/api/my-model?myfield__isempty=false

```
from django.core.validators import EMPTY_VALUES

class EmptyStringFilter(filters.BooleanFilter):
    def filter(self, qs, value):
        if value in EMPTY_VALUES:
            return qs

        exclude = self.exclude ^ (value is False)
        method = qs.exclude if exclude else qs.filter

        return method(**{self.name: ""})


class MyFilterSet(filters.FilterSet):
    myfield__isempty = EmptyStringFilter(name='myfield')

    class Meta:
        model = MyModel
```

Using `initial` values as defaults

In pre-1.0 versions of django-filter, a filter field's `initial` value was used as a default when no value was submitted. This behavior was not officially supported and has since been removed.

Warning: It is recommended that you do **NOT** implement the below as it adversely affects usability. Django forms don't provide this behavior for a reason.

- Using initial values as defaults is inconsistent with the behavior of Django forms.
- Default values prevent users from filtering by empty values.
- Default values prevent users from skipping that filter.

If defaults are necessary though, the following should mimic the pre-1.0 behavior:

```
class BaseFilterSet(FilterSet):

    def __init__(self, data=None, *args, **kwargs):
        # if filterset is bound, use initial values as defaults
        if data is not None:
            # get a mutable copy of the QueryDict
            data = data.copy()

            for name, f in self.base_filters.items():
                initial = f.extra.get('initial')

                # filter param is either missing or empty, use initial as default
                if not data.get(name) and initial:
                    data[name] = initial
```

```
        super(BaseFilterSet, self).__init__(data, *args, **kwargs)
```

CHAPTER 5

Migrating to 1.0

The 1.0 release of django-filter introduces several API changes and refinements that break forwards compatibility. Below is a list of deprecations and instructions on how to migrate to the 1.0 release. A forwards-compatible 0.15 release has also been created to help with migration. It is compatible with both the existing and new APIs and will raise warnings for deprecated behavior.

Enabling warnings

To view the deprecations, you may need to enable warnings within python. This can be achieved with either the `-W` flag, or with `PYTHONWARNINGS` environment variable. For example, you could run your test suite like so:

```
$ python -W once manage.py test
```

The above would print all warnings once when they first occur. This is useful to know what violations exist in your code (or occasionally in third party code). However, it only prints the last line of the stack trace. You can use the following to raise the full exception instead:

```
$ python -W error manage.py test
```

MethodFilter and Filter.action replaced by Filter.method

Details: https://github.com/carltongibson/django-filter/pull/382

The functionality of `MethodFilter` and `Filter.action` has been merged together and replaced by the `Filter.method` parameter. The `method` parameter takes either a callable or the name of a `FilterSet` method. The signature now takes an additional `name` argument that is the name of the model field to be filtered on.

Since `method` is now a parameter of all filters, inputs are validated and cleaned by its `field_class`. The function will receive the cleaned value instead of the raw value.

```
# 0.x
class UserFilter(FilterSet):
    last_login = filters.MethodFilter()

    def filter_last_login(self, qs, value):
        # try to convert value to datetime, which may fail.
        if value and looks_like_a_date(value):
            value = datetime(value)

        return qs.filter(last_login=value})


# 1.0
class UserFilter(FilterSet):
    last_login = filters.CharFilter(method='filter_last_login')

    def filter_last_login(self, qs, name, value):
        return qs.filter(**{name: value})
```

QuerySet methods are no longer proxied

Details: https://github.com/carltongibson/django-filter/pull/440

The `__iter__()`, `__len__()`, `__getitem__()`, `count()` methods are no longer proxied from the queryset. To fix this, call the methods on the `.qs` property itself.

```
f = UserFilter(request.GET, queryset=User.objects.all())

# 0.x
for obj in f:
    ...

# 1.0
for obj in f.qs:
    ...
```

Filters no longer autogenerated when Meta.fields is not specified

Details: https://github.com/carltongibson/django-filter/pull/450

FilterSets had an undocumented behavior of autogenerating filters for all model fields when either `Meta.fields` was not specified or when set to `None`. This can lead to potentially unsafe data or schema exposure and has been deprecated in favor of explicitly setting `Meta.fields` to the `'__all__'` special value. You may also blacklist fields by setting the `Meta.exclude` attribute.

```
class UserFilter(FilterSet):
    class Meta:
        model = User
        fields = '__all__'

# or
class UserFilter(FilterSet):
    class Meta:
```

```
        model = User
        exclude = ['password']
```

Move FilterSet options to Meta class

Details: https://github.com/carltongibson/django-filter/issues/430

Several `FilterSet` options have been moved to the `Meta` class to prevent potential conflicts with declared filter names. This includes:

- `filter_overrides`
- `strict`
- `order_by_field`

```
# 0.x
class UserFilter(FilterSet):
    filter_overrides = {}
    strict = STRICTNESS.RAISE_VALIDATION_ERROR
    order_by_field = 'order'
    ...

# 1.0
class UserFilter(FilterSet):
    ...

    class Meta:
        filter_overrides = {}
        strict = STRICTNESS.RAISE_VALIDATION_ERROR
        order_by_field = 'order'
```

FilterSet ordering replaced by OrderingFilter

Details: https://github.com/carltongibson/django-filter/pull/472

The FilterSet ordering options and methods have been deprecated and replaced by *OrderingFilter*. Deprecated options include:

- `Meta.order_by`
- `Meta.order_by_field`

These options retain backwards compatibility with the following caveats:

- `order_by` asserts that `Meta.fields` is not using the dict syntax. This previously was undefined behavior, however the migration code is unable to support it.
- Prior, if no ordering was specified in the request, the FilterSet implicitly filtered by the first param in the `order_by` option. This behavior cannot be easily emulated but can be fixed by ensuring that the passed in queryset explicitly calls `.order_by()`.

```
filterset = MyFilterSet(queryset=MyModel.objects.order_by('field'))
```

The following methods are deprecated and will raise an assertion if present on the FilterSet:

- `.get_order_by()`
- `.get_ordering_field()`

To fix this, simply remove the methods from your class. You can subclass `OrderingFilter` to migrate any custom logic.

Deprecated `FILTERS_HELP_TEXT_FILTER` and `FILTERS_HELP_TEXT_EXCLUDE`

Details: https://github.com/carltongibson/django-filter/pull/437

Generated filter labels in 1.0 will be more descriptive, including humanized text about the lookup being performed and if the filter is an exclusion filter.

These settings will no longer have an effect and will be removed in the 1.0 release.

DRF filter backend raises `TemplateDoesNotExist` exception

Templates are now provided by django-filter. If you are receiving this error, you may need to add `'django_filters'` to your `INSTALLED_APPS` setting. Alternatively, you could provide your own templates.

CHAPTER 6

FilterSet Options

This document provides a guide on using additional FilterSet features.

Meta options

- *model*
- *fields*
- *exclude*
- *form*
- *together*
- *filter_overrides*
- *strict*

Automatic filter generation with `model`

The `FilterSet` is capable of automatically generating filters for a given `model`'s fields. Similar to Django's `ModelForm`, filters are created based on the underlying model field's type. This option must be combined with either the `fields` or `exclude` option, which is the same requirement for Django's `ModelForm` class, detailed here.

```
class UserFilter(django_filters.FilterSet):
    class Meta:
        model = User
        fields = ['username', 'last_login']
```

Declaring filterable `fields`

The `fields` option is combined with `model` to automatically generate filters. Note that generated filters will not overwrite filters declared on the `FilterSet`. The `fields` option accepts two syntaxes:

- a list of field names
- a dictionary of field names mapped to a list of lookups

```
class UserFilter(django_filters.FilterSet):
    class Meta:
        model = User
        fields = ['username', 'last_login']

# or

class UserFilter(django_filters.FilterSet):
    class Meta:
        model = User
        fields = {
            'username': ['exact', 'contains'],
            'last_login': ['exact', 'year__gt'],
        }
```

The list syntax will create an `exact` lookup filter for each field included in `fields`. The dictionary syntax will create a filter for each lookup expression declared for its corresponding model field. These expressions may include both transforms and lookups, as detailed in the lookup reference.

Disable filter fields with `exclude`

The `exclude` option accepts a blacklist of field names to exclude from automatic filter generation. Note that this option will not disable filters declared directly on the `FilterSet`.

```
class UserFilter(django_filters.FilterSet):
    class Meta:
        model = User
        exclude = ['password']
```

Custom Forms using `form`

The inner `Meta` class also takes an optional `form` argument. This is a form class from which `FilterSet.form` will subclass. This works similar to the `form` option on a `ModelAdmin`.

Group fields with `together`

The inner `Meta` class also takes an optional `together` argument. This is a list of lists, each containing field names. For convenience can be a single list/tuple when dealing with a single set of fields. Fields within a field set must either be all or none present in the request for `FilterSet.form` to be valid:

```
import django_filters

class ProductFilter(django_filters.FilterSet):
    class Meta:
        model = Product
```

```
        fields = ['price', 'release_date', 'rating']
        together = ['rating', 'price']
```

Customise filter generation with `filter_overrides`

The inner `Meta` class also takes an optional `filter_overrides` argument. This is a map of model fields to filter classes with options:

```
class ProductFilter(django_filters.FilterSet):

    class Meta:
        model = Product
        fields = ['name', 'release_date']
        filter_overrides = {
            models.CharField: {
                'filter_class': django_filters.CharFilter,
                'extra': lambda f: {
                    'lookup_expr': 'icontains',
                },
            },
            models.BooleanField: {
                'filter_class': django_filters.BooleanFilter,
                'extra': lambda f: {
                    'widget': forms.CheckboxInput,
                },
            },
        }
```

Handling validation errors with `strict`

The `strict` option determines the filterset's behavior when filters fail to validate. Example use:

```
from django_filters import FilterSet, STRICTNESS

class ProductFilter(FilterSet):
    class Meta:
        model = Product
        fields = ['name', 'release_date']
        strict = STRICTNESS.RETURN_NO_RESULTS
```

Currently, there are three different behaviors:

- `STRICTNESS.RETURN_NO_RESULTS` (default) This returns an empty queryset. The filterset form can then be rendered to display the input errors.
- `STRICTNESS.IGNORE` Instead of returning an empty queryset, invalid filters effectively become a noop. Valid filters are applied to the queryset however.
- `STRICTNESS.RAISE_VALIDATION_ERROR` This raises a `ValidationError` for all invalid filters. This behavior is generally useful with APIs.

If the `strict` option is not provided, then the filterset will default to the value of the `FILTERS_STRICTNESS` setting.

Overriding `FilterSet` methods

`filter_for_lookup()`

Prior to version 0.13.0, filter generation did not take into account the `lookup_expr` used. This commonly caused malformed filters to be generated for 'isnull', 'in', and 'range' lookups (as well as transformed lookups). The current implementation provides the following behavior:

- 'isnull' lookups return a `BooleanFilter`
- 'in' lookups return a filter derived from the CSV-based `BaseInFilter`.
- 'range' lookups return a filter derived from the CSV-based `BaseRangeFilter`.

If you want to override the `filter_class` and `params` used to instantiate filters for a model field, you can override `filter_for_lookup()`. Ex:

```
class ProductFilter(django_filters.FilterSet):
    class Meta:
        model = Product
        fields = {
            'release_date': ['exact', 'range'],
        }

    @classmethod
    def filter_for_lookup(cls, f, lookup_type):
        # override date range lookups
        if isinstance(f, models.DateField) and lookup_type == 'range':
            return django_filters.DateRangeFilter, {}

        # use default behavior otherwise
        return super(ProductFilter, cls).filter_for_lookup(f, lookup_type)
```

CHAPTER 7

Filter Reference

This is a reference document with a list of the filters and their arguments.

Core Arguments

The following are the core arguments that apply to all filters.

`name`

The name of the field this filter is supposed to filter on, if this is not provided it automatically becomes the filter's name on the `FilterSet`. You can traverse "relationship paths" using Django's __ syntax to filter fields on a related model. eg, `manufacturer__name`.

`label`

The label as it will apear in the HTML, analogous to a form field's label argument. If a label is not provided, a verbose label will be generated based on the field `name` and the parts of the `lookup_expr`. (See: *FILTERS_VERBOSE_LOOKUPS*).

`widget`

The django.form Widget class which will represent the `Filter`. In addition to the widgets that are included with Django that you can use there are additional ones that django-filter provides which may be useful:

- *LinkWidget* – this displays the options in a manner similar to the way the Django Admin does, as a series of links. The link for the selected option will have `class="selected"`.
- *BooleanWidget* – this widget converts its input into Python's True/False values. It will convert all case variations of `True` and `False` into the internal Python values.

- *CSVWidget* – this widget expects a comma separated value and converts it into a list of string values. It is expected that the field class handle a list of values as well as type conversion.
- *RangeWidget* – this widget is used with `RangeFilter` to generate two form input elements using a single field.

method

An optional argument that tells the filter how to handle the queryset. It can accept either a callable or the name of a method on the `FilterSet`. The method receives a `QuerySet`, the name of the model field to filter on, and the value to filter with. It should return a `Queryset` that is filtered appropriately.

The passed in value is validated and cleaned by the filter's `field_class`, so raw value transformation and empty value checking should be unnecessary.

```
class F(FilterSet):
    """Filter for Books by if books are published or not"""
    published = BooleanFilter(name='published_on', method='filter_published')

    def filter_published(self, queryset, name, value):
        # construct the full lookup expression.
        lookup = '__'.join([name, 'isnull'])
        return queryset.filter(**{lookup: False})

        # alternatively, it may not be necessary to construct the lookup.
        return queryset.filter(published_on__isnull=False)

    class Meta:
        model = Book
        fields = ['published']


# Callables may also be defined out of the class scope.
def filter_not_empty(queryset, name, value):
    lookup = '__'.join([name, 'isnull'])
    return queryset.filter(**{lookup: False})

class F(FilterSet):
    """Filter for Books by if books are published or not"""
    published = BooleanFilter(name='published_on', method=filter_not_empty)

    class Meta:
        model = Book
        fields = ['published']
```

lookup_expr

The lookup expression that should be performed using Django's ORM.

A `list` or `tuple` of lookup types is also accepted, allowing the user to select the lookup from a dropdown. The list of lookup types are filtered against `filters.LOOKUP_TYPES`. If *lookup_expr=None* is passed, then a list of all lookup types will be generated:

```
class ProductFilter(django_filters.FilterSet):
    name = django_filters.CharFilter(lookup_expr=['exact', 'iexact'])
```

You can enable custom lookups by adding them to `LOOKUP_TYPES`:

```
from django_filters import filters

filters.LOOKUP_TYPES = ['gt', 'gte', 'lt', 'lte', 'custom_lookup_type']
```

Additionally, you can provide human-friendly help text by overriding `LOOKUP_TYPES`:

```
# filters.py
from django_filters import filters

filters.LOOKUP_TYPES = [
    ('', '---------'),
    ('exact', 'Is equal to'),
    ('not_exact', 'Is not equal to'),
    ('lt', 'Lesser than'),
    ('gt', 'Greater than'),
    ('gte', 'Greater than or equal to'),
    ('lte', 'Lesser than or equal to'),
    ('startswith', 'Starts with'),
    ('endswith', 'Ends with'),
    ('contains', 'Contains'),
    ('not_contains', 'Does not contain'),
]
```

distinct

A boolean value that specifies whether the Filter will use distinct on the queryset. This option can be used to eliminate duplicate results when using filters that span related models. Defaults to `False`.

exclude

A boolean value that specifies whether the Filter should use `filter` or `exclude` on the queryset. Defaults to `False`.

**kwargs

Any additional keyword arguments are stored as the `extra` parameter on the filter. They are provided to the accompanying form Field and can be used to provide arguments like `choices`.

ModelChoiceFilter and ModelMultipleChoiceFilter arguments

These arguments apply specifically to ModelChoiceFilter and ModelMultipleChoiceFilter only.

queryset

`ModelChoiceFilter` and `ModelMultipleChoiceFilter` require a queryset to operate on which must be passed as a kwarg.

to_field_name

If you pass in `to_field_name` (which gets forwarded to the Django field), it will be used also in the default `get_filter_predicate` implementation as the model's attribute.

Filters

CharFilter

This filter does simple character matches, used with `CharField` and `TextField` by default.

UUIDFilter

This filter matches UUID values, used with `models.UUIDField` by default.

BooleanFilter

This filter matches a boolean, either `True` or `False`, used with `BooleanField` and `NullBooleanField` by default.

ChoiceFilter

This filter matches values in its `choices` argument. The `choices` must be explicitly passed when the filter is declared on the `FilterSet`. For example,

```
class User(models.Model):
    username = models.CharField(max_length=255)
    first_name = SubCharField(max_length=100)
    last_name = SubSubCharField(max_length=100)

    status = models.IntegerField(choices=STATUS_CHOICES, default=0)

STATUS_CHOICES = (
    (0, 'Regular'),
    (1, 'Manager'),
    (2, 'Admin'),
)

class F(FilterSet):
    status = ChoiceFilter(choices=STATUS_CHOICES)
    class Meta:
        model = User
        fields = ['status']
```

`ChoiceFilter` also has arguments that enable a choice for not filtering, as well as a choice for filtering by `None` values. Each of the arguments have a corresponding global setting (*Settings Reference*).

- `empty_label`: The display label to use for the select choice to not filter. The choice may be disabled by setting this argument to `None`. Defaults to `FILTERS_EMPTY_CHOICE_LABEL`.
- `null_label`: The display label to use for the choice to filter by `None` values. The choice may be disabled by setting this argument to `None`. Defaults to `FILTERS_NULL_CHOICE_LABEL`.

- `null_value`: The special value to match to enable filtering by `None` values. This value defaults `FILTERS_NULL_CHOICE_VALUE` and needs to be a non-empty value (`''`, `None`, `[]`, `()`, `{}`).

TypedChoiceFilter

The same as `ChoiceFilter` with the added possibility to convert value to match against. This could be done by using *coerce* parameter. An example use-case is limiting boolean choices to match against so only some predefined strings could be used as input of a boolean filter:

```
import django_filters
from distutils.util import strtobool

BOOLEAN_CHOICES = (('false', 'False'), ('true', 'True'),)

class YourFilterSet(django_filters.FilterSet):
    ...
    flag = django_filters.TypedChoiceFilter(choices=BOOLEAN_CHOICES,
                                            coerce=strtobool)
```

MultipleChoiceFilter

The same as `ChoiceFilter` except the user can select multiple choices and the filter will form the OR of these choices by default to match items. The filter will form the AND of the selected choices when the `conjoined=True` argument is passed to this class.

Multiple choices are represented in the query string by reusing the same key with different values (e.g. ‘’?status=Regular&status=Admin’‘).

`distinct` defaults to `True` as to-many relationships will generally require this.

Advanced Use: Depending on your application logic, when all or no choices are selected, filtering may be a noop. In this case you may wish to avoid the filtering overhead, particularly of the *distinct* call.

Set *always_filter* to False after instantiation to enable the default *is_noop* test.

Override *is_noop* if you require a different test for your application.

TypedMultipleChoiceFilter

Like `MultipleChoiceFilter`, but in addition accepts the `coerce` parameter, as in `TypedChoiceFilter`.

DateFilter

Matches on a date. Used with `DateField` by default.

TimeFilter

Matches on a time. Used with `TimeField` by default.

DateTimeFilter

Matches on a date and time. Used with `DateTimeField` by default.

IsoDateTimeFilter

Uses `IsoDateTimeField` to support filtering on ISO 8601 formatted dates, as are often used in APIs, and are employed by default by Django REST Framework.

Example:

```
class F(FilterSet):
    """Filter for Books by date published, using ISO 8601 formatted dates"""
    published = IsoDateTimeFilter()

    class Meta:
        model = Book
        fields = ['published']
```

DurationFilter

Matches on a duration. Used with `DurationField` by default.

Supports both Django ('%d %H:%M:%S.%f') and ISO 8601 formatted durations (but only the sections that are accepted by Python's timedelta, so no year, month, and week designators, e.g. 'P3DT10H22M').

ModelChoiceFilter

Similar to a `ChoiceFilter` except it works with related models, used for `ForeignKey` by default.

If automatically instantiated, `ModelChoiceFilter` will use the default `QuerySet` for the related field. If manually instantiated you **must** provide the `queryset` kwarg.

Example:

```
class F(FilterSet):
    """Filter for books by author"""
    author = ModelChoiceFilter(queryset=Author.objects.all())

    class Meta:
        model = Book
        fields = ['author']
```

The `queryset` argument also supports callable behavior. If a callable is passed, it will be invoked with `Filterset.request` as its only argument. This allows you to easily filter by properties on the request object without having to override the `FilterSet.__init__`.

Note: You should expect that the *request* object may be *None*.

```
def departments(request):
    if request is None:
        return Department.objects.none()

    company = request.user.company
    return company.department_set.all()

class EmployeeFilter(filters.FilterSet):
    department = filters.ModelChoiceFilter(queryset=departments)
    ...
```

ModelMultipleChoiceFilter

Similar to a `MultipleChoiceFilter` except it works with related models, used for `ManyToManyField` by default.

As with `ModelChoiceFilter`, if automatically instantiated, `ModelMultipleChoiceFilter` will use the default `QuerySet` for the related field. If manually instantiated you **must** provide the `queryset` kwarg. Like `ModelChoiceFilter`, the `queryset` argument has callable behavior.

To use a custom field name for the lookup, you can use `to_field_name`:

```
class FooFilter(BaseFilterSet):
    foo = django_filters.filters.ModelMultipleChoiceFilter(
        name='attr__uuid',
        to_field_name='uuid',
        queryset=Foo.objects.all(),
    )
```

If you want to use a custom queryset, e.g. to add annotated fields, this can be done as follows:

```
class MyMultipleChoiceFilter(django_filters.ModelMultipleChoiceFilter):
    def get_filter_predicate(self, v):
        return {'annotated_field': v.annotated_field}

    def filter(self, qs, value):
        if value:
            qs = qs.annotate_with_custom_field()
            qs = super().filter(qs, value)
        return qs

foo = MyMultipleChoiceFilter(
    to_field_name='annotated_field',
    queryset=Model.objects.annotate_with_custom_field(),
)
```

The `annotate_with_custom_field` method would be defined through a custom QuerySet, which then gets used as the model's manager:

```
class CustomQuerySet(models.QuerySet):
    def annotate_with_custom_field(self):
        return self.annotate(
            custom_field=Case(
                When(foo__isnull=False,
                     then=F('foo__uuid')),
                When(bar__isnull=False,
                     then=F('bar__uuid')),
                default=None,
            ),
        )

class MyModel(models.Model):
    objects = CustomQuerySet.as_manager()
```

NumberFilter

Filters based on a numerical value, used with `IntegerField`, `FloatField`, and `DecimalField` by default.

NumericRangeFilter

Filters where a value is between two numerical values, or greater than a minimum or less than a maximum where only one limit value is provided. This filter is designed to work with the Postgres Numerical Range Fields, including `IntegerRangeField`, `BigIntegerRangeField` and `FloatRangeField` (available since Django 1.8). The default widget used is the `RangeField`.

Regular field lookups are available in addition to several containment lookups, including `overlap`, `contains`, and `contained_by`. More details in the Django docs.

If the lower limit value is provided, the filter automatically defaults to `startswith` as the lookup and `endswith` if only the upper limit value is provided.

RangeFilter

Filters where a value is between two numerical values, or greater than a minimum or less than a maximum where only one limit value is provided.

```
class F(FilterSet):
    """Filter for Books by Price"""
    price = RangeFilter()

    class Meta:
        model = Book
        fields = ['price']

qs = Book.objects.all().order_by('title')

# Range: Books between 5€ and 15€
f = F({'price_0': '5', 'price_1': '15'}, queryset=qs)

# Min-Only: Books costing more the 11€
f = F({'price_0': '11'}, queryset=qs)

# Max-Only: Books costing less than 19€
f = F({'price_1': '19'}, queryset=qs)
```

DateRangeFilter

Filter similar to the admin changelist date one, it has a number of common selections for working with date fields.

DateFromToRangeFilter

Similar to a `RangeFilter` except it uses dates instead of numerical values. It can be used with `DateField`. It also works with `DateTimeField`, but takes into consideration only the date.

Example of using the `DateField` field:

```
class Comment(models.Model):
    date = models.DateField()
    time = models.TimeField()

class F(FilterSet):
    date = DateFromToRangeFilter()

    class Meta:
        model = Comment
        fields = ['date']

# Range: Comments added between 2016-01-01 and 2016-02-01
f = F({'date_0': '2016-01-01', 'date_1': '2016-02-01'})

# Min-Only: Comments added after 2016-01-01
f = F({'date_0': '2016-01-01'})

# Max-Only: Comments added before 2016-02-01
f = F({'date_1': '2016-02-01'})

.. note::
    When filtering ranges that occurs on DST transition dates␣
↪``DateFromToRangeFilter`` will use the first valid hour of the day for start␣
↪datetime and the last valid hour of the day for end datetime.
    This is OK for most applications, but if you want to customize this behavior you␣
↪must extend ``DateFromToRangeFilter`` and make a custom field for it.

.. warning::
    If you're using Django prior to 1.9 you may hit ``AmbiguousTimeError`` or␣
↪``NonExistentTimeError`` when start/end date matches DST start/end respectively.
    This occurs because versions before 1.9 don't allow to change the DST behavior␣
↪for making a datetime aware.
```

Example of using the `DateTimeField` field:

```
class Article(models.Model):
    published = models.DateTimeField()

class F(FilterSet):
    published = DateFromToRangeFilter()

    class Meta:
        model = Article
        fields = ['published']

Article.objects.create(published='2016-01-01 8:00')
Article.objects.create(published='2016-01-20 10:00')
Article.objects.create(published='2016-02-10 12:00')

# Range: Articles published between 2016-01-01 and 2016-02-01
f = F({'published_0': '2016-01-01', 'published_1': '2016-02-01'})
assert len(f.qs) == 2

# Min-Only: Articles published after 2016-01-01
f = F({'published_0': '2016-01-01'})
assert len(f.qs) == 3

# Max-Only: Articles published before 2016-02-01
```

```
f = F({'published_1': '2016-02-01'})
assert len(f.qs) == 2
```

DateTimeFromToRangeFilter

Similar to a `RangeFilter` except it uses datetime format values instead of numerical values. It can be used with `DateTimeField`.

Example:

```
class Article(models.Model):
    published = models.DateTimeField()

class F(FilterSet):
    published = DateTimeFromToRangeFilter()

    class Meta:
        model = Article
        fields = ['published']

Article.objects.create(published='2016-01-01 8:00')
Article.objects.create(published='2016-01-01 9:30')
Article.objects.create(published='2016-01-02 8:00')

# Range: Articles published 2016-01-01 between 8:00 and 10:00
f = F({'published_0': '2016-01-01 8:00', 'published_1': '2016-01-01 10:00'})
assert len(f.qs) == 2

# Min-Only: Articles published after 2016-01-01 8:00
f = F({'published_0': '2016-01-01 8:00'})
assert len(f.qs) == 3

# Max-Only: Articles published before 2016-01-01 10:00
f = F({'published_1': '2016-01-01 10:00'})
assert len(f.qs) == 2
```

TimeRangeFilter

Similar to a `RangeFilter` except it uses time format values instead of numerical values. It can be used with `TimeField`.

Example:

```
class Comment(models.Model):
    date = models.DateField()
    time = models.TimeField()

class F(FilterSet):
    time = TimeRangeFilter()

    class Meta:
        model = Comment
        fields = ['time']

# Range: Comments added between 8:00 and 10:00
```

```
f = F({'time_0': '8:00', 'time_1': '10:00'})

# Min-Only: Comments added after 8:00
f = F({'time_0': '8:00'})

# Max-Only: Comments added before 10:00
f = F({'time_1': '10:00'})
```

AllValuesFilter

This is a `ChoiceFilter` whose choices are the current values in the database. So if in the DB for the given field you have values of 5, 7, and 9 each of those is present as an option. This is similar to the default behavior of the admin.

AllValuesMultipleFilter

This is a `MultipleChoiceFilter` whose choices are the current values in the database. So if in the DB for the given field you have values of 5, 7, and 9 each of those is present as an option. This is similar to the default behavior of the admin.

BaseInFilter

This is a base class used for creating IN lookup filters. It is expected that this filter class is used in conjunction with another filter class, as this class **only** validates that the incoming value is comma-separated. The secondary filter is then used to validate the individual values.

Example:

```
class NumberInFilter(BaseInFilter, NumberFilter):
    pass

class F(FilterSet):
    id__in = NumberInFilter(name='id', lookup_expr='in')

    class Meta:
        model = User

User.objects.create(username='alex')
User.objects.create(username='jacob')
User.objects.create(username='aaron')
User.objects.create(username='carl')

# In: User with IDs 1 and 3.
f = F({'id__in': '1,3'})
assert len(f.qs) == 2
```

BaseRangeFilter

This is a base class used for creating RANGE lookup filters. It behaves identically to `BaseInFilter` with the exception that it expects only two comma-separated values.

Example:

```
class NumberRangeFilter(BaseInFilter, NumberFilter):
    pass

class F(FilterSet):
    id__range = NumberRangeFilter(name='id', lookup_expr='range')

    class Meta:
        model = User

User.objects.create(username='alex')
User.objects.create(username='jacob')
User.objects.create(username='aaron')
User.objects.create(username='carl')

# Range: User with IDs between 1 and 3.
f = F({'id__range': '1,3'})
assert len(f.qs) == 3
```

OrderingFilter

Enable queryset ordering. As an extension of `ChoiceFilter` it accepts two additional arguments that are used to build the ordering choices.

- `fields` is a mapping of {model field name: parameter name}. The parameter names are exposed in the choices and mask/alias the field names used in the `order_by()` call. Similar to field `choices`, `fields` accepts the 'list of two-tuples' syntax that retains order. `fields` may also just be an iterable of strings. In this case, the field names simply double as the exposed parameter names.
- `field_labels` is an optional argument that allows you to customize the display label for the corresponding parameter. It accepts a mapping of {field name: human readable label}. Keep in mind that the key is the field name, and not the exposed parameter name.

```
class UserFilter(FilterSet):
    account = CharFilter(name='username')
    status = NumberFilter(name='status')

    o = OrderingFilter(
        # tuple-mapping retains order
        fields=(
            ('username', 'account'),
            ('first_name', 'first_name'),
            ('last_name', 'last_name'),
        ),

        # labels do not need to retain order
        field_labels={
            'username': 'User account',
        }
    )

    class Meta:
        model = User
        fields = ['first_name', 'last_name']

>>> UserFilter().filters['o'].field.choices
[
```

```
    ('account', 'User account'),
    ('-account', 'User account (descending)'),
    ('first_name', 'First name'),
    ('-first_name', 'First name (descending)'),
    ('last_name', 'Last name'),
    ('-last_name', 'Last name (descending)'),
]
```

Additionally, you can just provide your own `choices` if you require explicit control over the exposed options. For example, when you might want to disable descending sort options.

```
class UserFilter(FilterSet):
    account = CharFilter(name='username')
    status = NumberFilter(name='status')

    o = OrderingFilter(
        choices=(
            ('account', 'Account'),
        ),
        fields={
            'username': 'account',
        },
    )
```

This filter is also CSV-based, and accepts multiple ordering params. The default select widget does not enable the use of this, but it is useful for APIs.

Adding Custom filter choices

If you wish to sort by non-model fields, you'll need to add custom handling to an `OrderingFilter` subclass. For example, if you want to sort by a computed 'relevance' factor, you would need to do something like the following:

```
class CustomOrderingFilter(django_filters.OrderingFilter):

    def __init__(self, *args, **kwargs):
        super(CustomOrderingFilter, self).__init__(*args, **kwargs)
        self.extra['choices'] += [
            ('relevance', 'Relevance'),
            ('-relevance', 'Relevance (descending)'),
        ]


    def filter(self, qs, value):
        # OrderingFilter is CSV-based, so `value` is a list
        if any(v in ['relevance', '-relevance'] for v in value):
            # sort queryset by relevance
            return ...

        return super(CustomOrderingFilter, self).filter(qs, value)
```

CHAPTER 8

Field Reference

IsoDateTimeField

Extends `django.forms.DateTimeField` to allow parsing ISO 8601 formated dates, in addition to existing formats

Defines a class level attribute `ISO_8601` as constant for the format.

Sets `input_formats = [ISO_8601]` — this means that by default `IsoDateTimeField` will **only** parse ISO 8601 formated dates.

You may set `input_formats` to your list of required formats as per the DateTimeField Docs, using the `ISO_8601` class level attribute to specify the ISO 8601 format.

```
f = IsoDateTimeField()
f.input_formats = [IsoDateTimeField.ISO_8601] + DateTimeField.input_formats
```

CHAPTER 9

Widget Reference

This is a reference document with a list of the provided widgets and their arguments.

LinkWidget

This widget renders each option as a link, instead of an actual <input>. It has one method that you can override for additional customizability. `option_string()` should return a string with 3 Python keyword argument placeholders:

1. `attrs`: This is a string with all the attributes that will be on the final `<a>` tag.
2. `query_string`: This is the query string for use in the `href` option on the `<a>` element.
3. `label`: This is the text to be displayed to the user.

BooleanWidget

This widget converts its input into Python's True/False values. It will convert all case variations of `True` and `False` into the internal Python values. To use it, pass this into the `widgets` argument of the `BooleanFilter`:

```
active = BooleanFilter(widget=BooleanWidget())
```

CSVWidget

This widget expects a comma separated value and converts it into a list of string values. It is expected that the field class handle a list of values as well as type conversion.

RangeWidget

This widget is used with `RangeFilter` and its subclasses. It generates two form input elements which generally act as start/end values in a range. Under the hood, it is django's `forms.TextInput` widget and excepts the same arguments and values. To use it, pass it to `widget` argument of a `RangeField`:

```
date_range = DateFromToRangeFilter(widget=RangeWidget(attrs={'placeholder': 'YYYY/MM/
↪DD'}))
```

CHAPTER 10

Settings Reference

Here is a list of all available settings of django-filters and their default values. All settings are prefixed with `FILTERS_`, although this is a bit verbose it helps to make it easy to identify these settings.

FILTERS_EMPTY_CHOICE_LABEL

Default: `'---------'`

Set the default value for `ChoiceFilter.empty_label`. You may disable the empty choice by setting this to `None`.

FILTERS_NULL_CHOICE_LABEL

Default: `None`

Set the default value for `ChoiceFilter.null_label`. You may enable the null choice by setting a non-`None` value.

FILTERS_NULL_CHOICE_VALUE

Default: `'null'`

Set the default value for `ChoiceFilter.null_value`. You may want to change this value if the default `'null'` string conflicts with an actual choice.

FILTERS_DISABLE_HELP_TEXT

Default: `False`

Some filters provide informational `help_text`. For example, csv-based filters (`filters.BaseCSVFilter`) inform users that "Multiple values may be separated by commas".

You may set this to `True` to disable the `help_text` for **all** filters, removing the text from the rendered form's output.

FILTERS_VERBOSE_LOOKUPS

Note: This is considered an advanced setting and is subject to change.

Default:

```
# refer to 'django_filters.conf.DEFAULTS'
'VERBOSE_LOOKUPS': {
    'exact': _(''),
    'iexact': _(''),
    'contains': _('contains'),
    'icontains': _('contains'),
    ...
}
```

This setting controls the verbose output for generated filter labels. Instead of getting expression parts such as "lt" and "contained_by", the verbose label would contain "is less than" and "is contained by". Verbose output may be disabled by setting this to a falsy value.

This setting also accepts callables. The callable should not require arguments and should return a dictionary. This is useful for extending or overriding the default terms without having to copy the entire set of terms to your settings. For example, you could add verbose output for "exact" lookups.

```
# settings.py
def FILTERS_VERBOSE_LOOKUPS():
    from django_filters.conf import DEFAULTS

    verbose_lookups = DEFAULTS['VERBOSE_LOOKUPS'].copy()
    verbose_lookups.update({
        'exact': 'is equal to',
    })
    return verbose_lookups
```

FILTERS_STRICTNESS

Default: `STRICTNESS.RETURN_NO_RESULTS`

Set the global default for FilterSet *strictness*. If `strict` is not provided to the filterset, it will default to this setting. You can change the setting like so:

```
# settings.py
from django_filters import STRICTNESS

FILTERS_STRICTNESS = STRICTNESS.RETURN_NO_RESULTS
```

CHAPTER 11

Running the Test Suite

The easiest way to run the django-filter tests is to check out the source code and create a virtualenv where you can install the test dependencies. Django-filter uses a custom test runner to configure the environment, so a wrapper script is available to set up and run the test suite.

Note: The following assumes you have virtualenv and git installed.

Clone the repository

Get the source code using the following command:

```
$ git clone https://github.com/carltongibson/django-filter.git
```

Switch to the django-filter directory:

```
$ cd django-filter
```

Set up the virtualenv

Create a new virtualenv to run the test suite in:

```
$ virtualenv venv
```

Then activate the virtualenv and install the test requirements:

```
$ source venv/bin/activate
$ pip install -r requirements/test.txt
```

Execute the test runner

Run the tests with the runner script:

```
$ python runtests.py
```

Test all supported versions

You can also use the excellent tox testing tool to run the tests against all supported versions of Python and Django. Install tox, and then simply run:

```
$ pip install tox
$ tox
```