django-refinery Documentation

Release 0.2dev

Jacob Radford

May 19, 2012

CONTENTS

Django-refinery is a generic, reusable application to alleviate some of the more mundane bits of view code. Specifically allowing the users to filter down a queryset based on a models fields, and displaying the form to let them do this.

Contents:

CHAPTER

ONE

INSTALLING DJANGO-REFINERY

To install, simply place the `refinery` directory somehwere on your `PYTHONPATH`, and then add `'refinery'` to your `INSTALLED_APPS`.

CHAPTER

TWO

USING DJANGO-REFINERY

Django-refinery 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. 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)
```

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

```
import refinery

class ProductFilterTool(refinery.FilterTool):
    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 overide filters, or add new ones using a declarative syntax:

```
import refinery

class ProductFilterTool(refinery.FilterTool):
    price = refinery.NumberFilter(lookup_type='lt')
    class Meta:
        model = Product
        fields = ['price', 'release_date']
```

Filters take a `lookup_type` argument which specifies what lookup type to use with Django's ORM. So here when a user entered a price it would show all Products with a price less than that.

You can also specify the lookup type when specifying the `fields`:

```
import refinery

class ProductFilterTool(refinery.FilterTool):
    class Meta:
        model = Product
        fields = ['price__lt', 'release_date']
```

Filters also take any arbitrary keyword arguments which get passed onto the `django.forms.Field` constructor. These extra keyword arguments get stored in `Filter.extra`, so it's possible to overide the constructor of a `FilterTool` to add extra ones:

```
class ProductFilterTool(refinery.FilterTool):
    class Meta:
        model = Product
        fields = ['manufacturer']

    def __init__(self, *args, **kwargs):
        super(ProductFilterTool, self).__init__(*args, **kwargs)
        self.filters['manufacturer'].extra.update(
            {'empty_label': u'All Manufacturers'})
```

Now we need to write a view:

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

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

And lastly we need a template:

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

{% block content %}
    <form action="" method="get">
        {{ filtertool.form.as_p }}
        <input type="submit" />
    </form>
    {% for obj in filtertool %}
        {{ 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 `FilterTool` we get the objects in the resulting queryset.

You can also allow the user to control ordering, this is done by providing the `order_by` argument in the Filter's Meta class. `order_by` can be either a `list` or `tuple` of field names, in which case those are the options, or it can be a `bool` which, if True, indicates that all fields that have the user can filter on can also be sorted on.

If `order_by` is a list of lists, the inner lists must be in name/label pairs. This lets you override the display names of your ordering fields:

```
order_by = (
    ('name', 'Company Name'),
    ('average_rating', 'Stars'),
)
```

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

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.

If you want to use a custom widget, or in any other way overide the ordering field you can overide the `get_ordering_field()` method on a `FilterTool`. This method just needs to return a Form Field.

2.1 Generic View

In addition to the above usage there is also a generic view included in django-refinery, which lives at `refinery.views.object_filtered_list`. You must provide either a `model` or `filter_class` argument, similar to the `create_update` view in Django itself:

```
url(r'^list/$',
    'refinery.views.object_filtered_list',
    {'model': Product}),
```

You must provide a template at `<app>/<model>_filtered_list.html` which gets the context parameter `filtertool`.

CHAPTER

THREE

INTEGRATING WITH OTHER APPLICATIONS

3.1 Django-pagination

To use django-refinery alongside django-pagination requires 2 minor changes to the code snippets shown in *Using django-refinery*.

Change the template code to:

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

{% block content %}
    <form action="" method="get">
        {{ filtertool.form.as_p }}
        <input type="submit" />
    </form>
    {% autopaginate filtertool.qs 40 as filtered_list %}
    {% for obj in filtered_list %}
        {{ obj.name }} - ${{ obj.price }}<br />
    {% endfor %}
    {% paginate %}
{% endblock %}
```

Also, if you have - for example - selected page 5 of results, and then try to apply a filter, you will be sent to page 5 of the filtered results. To avoid this, don't add django-pagination's 'page' parameter to the FilterTool in the view code:

```
def product_list(request):
    g = request.GET.copy()
    if 'page' in g:
        del g['page']
    f = ProductFilterTool(g, queryset=Product.objects.all())
    return render_to_response('my_app/template.html', {'filtertool': f})
```

CHAPTER

FOUR

FILTER REFERENCE

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

4.1 Filters

4.1.1 `CharFilter`

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

4.1.2 `BooleanFilter`

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

4.1.3 `ChoiceFilter`

This filter matches an item of any type by choices, used with any field that has `choices`.

4.1.4 `MultipleChoiceFilter`

The same as `ChoiceFilter` except the user can select multiple items and it selects the OR of all the choices.

4.1.5 `DateFilter`

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

4.1.6 `DateTimeFilter`

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

4.1.7 `TimeFilter`

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

4.1.8 ModelChoiceFilter

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

4.1.9 ModelMultipleChoiceFilter

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

4.1.10 NumberFilter

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

4.1.11 RangeFilter

Filters where a value is between two numerical values.

4.1.12 DateRangeFilter

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

4.1.13 AllValuesFilter

This is a `ChoiceFilter` who's 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.

4.2 Core Arguments

4.2.1 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 `FilterTool`.

4.2.2 label

The label as it will apear in the HTML, analogous to a form field's label argument.

4.2.3 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-filte provides which may be useful:

- `refinery.widgets.LinkWidget` – this displays the options in a mannner similar to the way the Django Admin does, as a series of links. The link for the selected option will have `class="selected"`.

4.2.4 action

An optional callable that tells the filter how to handle the supplied value. It recieves the value to filter on and should return a `Q` object that is filtered appropriately.

4.2.5 lookup_type

The type of lookup that should be preformed using the Django ORM. All the normal options are allowed, and should be provided as a string. You can also provide either `None` or a `list` or `tuple`, if `None` is provided then user can select the lookup_type from all the ones available in the Django ORM, and if a `list` or `tuple` is provided the user can select from those options.

The `list` or `tuple` can be formatted with labels to present to the user when selecting, the same way as Django's Field.choices are formatted, i.e.:

```
lookup_type=[('lt', 'Less than'), ('gt', 'Greater than')]
```

4.2.6 **kwargs

Any extra keyword arguments will be provided to the accompanying form Field. This can be used to provide arguments like `choices` or `queryset`.

CHAPTER

FIVE

WIDGET REFERENCE

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

5.1 `LinkWidget`

This widget renders each option as a link, intead of an actual <input>. It has one method that you can overide for additional customizability. `option_string()` should return a string with 3 Python keyword arugment 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>'' elemeent.
3. ''label'': This is the text to be displayed to the user.
```

CHAPTER

SIX

RUNNING THE DJANGO-REFINERY TESTS

In order to run the django-refinery tests you must not only add `'refinery'` to your `INSTALLED_APPS` settings, but also `'tests'`, which tells Django to setup the test models. This step is only necessary if you want to run the django-refinery regression tests.

CHAPTER

SEVEN

HISTORY AND CREDITS

7.1 Changelog

The project follows the Semantic Versioning specification for its version numbers. Patch-level increments indicate bug fixes, minor version increments indicate new functionality and major version increments indicate backwards incompatible changes.

7.1.1 Version 0.2 (Unreleased)

7.1.2 Version 0.1 (2012-05-19)

- Initial version of django-refinery (extracted from django-filter v0.5.3).
- Converted all original doctests to unittests.
- Merged the work of various forks of django-filter to add a lot of new functionality (with source at the end):
  - Filtering logic refactored to use Q objects instead of QuerySets (from the django-qfilters project by Steve Yeago).
  - Added ability to specify lookup type in FilterSet meta class (Maurizio Melani).
  - Allow to ovveride form and form fields creation (Marke Wywial via I-DOTCOM LLC).
  - Add Class-based generic view (Alisue).
  - Add empty_label for ChoiceFilter (Vladislav Poluhin).
  - Allow order_by to take a list of lists (or tuples), letting you override the display name of potential ordering columns. (Ross Poulton).
  - Altered test model definitions for field inheritance tests, added inherited field definition for testing, made filters work with inherited fields, added open range filters, added date and time range fields, and added support for derived model fields (Sergiy Kuzmenko).
  - Document how to use alongside django-pagination, fixed problem where blank choice was not clearing the query variable, and added ability for LinkWidget to accept (None,"Label") element for choices tuple which clears the given filter (Richard Barran).
  - Filter instance queryset is directly subscriptable andadded multi-field filter (Stephan Jaekel).
  - Reversed position of field and lookupfield and added the ability to provide 'pretty' options for lookup types (Tino de Bruijn).

7.2 Credits

The django-refinery package was written by Jacob Radford and based on the django-filter project by Alex Gaynor.

7.2.1 django-refinery Project Leads

- Jacob Radford

7.2.2 django-refinery Contributors

For the current list of code contributors to the django-refinery project, visit http://github.com/nkryptic/django-refinery/contributors

Additional concepts were integrated from:

- django-qfilters project by Steve Yeago
- Alisue
- Stephan Jaekel
- Marke Wywial via I-DOTCOM LLC
- Vladislav Poluhin
- Ross Poulton
- Sergiy Kuzmenko
- Tino de Bruijn
- Richard Barran
- Maurizio Melani

7.2.3 django-filter Project Founder

- Alex Gaynor

7.2.4 django-filter Contributors

- Ben Firshman
- Jannis Leidel
- Martin Mahner
- Brian Rosner
- Adam Vandenberg
- Tino de Bruijn
- Vladimir Sidorenko
- Maximillian Dornseif
- Marc Fargas