django-modelqueryform Documentation

Release 3.0

Chaim Kirby

Nov 07, 2018

Contents

django-modelqueryform is a flexible app that helps you build Q object generating forms.

It is a great tool if you want you users to be able to do filtered searches against your models.

CHAPTER 1

Project

The project can be found at https://github.com/ckirby/django-modelqueryform

CHAPTER 2

Documentation

The full documentation is at https://django-modelqueryform.readthedocs.org.

CHAPTER 3

Requirements

- Django 1.11+
- Python 3.4+

CHAPTER 4

Features

- Useable default FormFields for ModelFields that:
  - Have *.choices* defined or are inherently made of choices (ie. *BooleanField* and *NullBooleanField*)
  - Are represented as numeric types (eg. *IntegerField*, *FloatField*, etc.)
  - Text backed fields need code written to handle them. That is easy though, because:
- Creation of FormFields, Q objects, and User readable query terms are completely customizable. You can target ModelFields:
  - By name (If the field has specific requirements)
  - By field type (Use the same widget or Q object builder for all *CharField*s)
- Can follow Model relationships or treat relationship fields as *.choices*
- Provides a new Field and Widget (*RangeField*, *RangeWidget*). These allow users to generate a *__gte*, *__lte* pair for the orm, optionally also including an *__isnull*
  - RangeField
    - Dynamically generates min and max boundaries. (Aggregate *Min* and *Max* on the model field)
    - If *null=True* on the ModelField allows user to indicate if they want to include null values in the query
  - RangeWidget
    - Returns a *MultiWidget* with 2 *NumberInput* widgets (with min and max attributes)

CHAPTER 5

Contents

5.1 Installation

Install django-modelqueryform using *pip*:

```
pip install django-modelqueryform
```

Add *modelqueryform* to your *INSTALLED_APPS* setting:

```
INSTALLED_APPS = (
    ...
    'modelqueryform',
)
```

5.2 Usage

For these examples we will use the following models:

```
class MyModel(models.Model):
    age = models.IntegerField()
    employed = models.NullBooleanField()#Yes,No,Unknown
    degree = models.CharField(max_length = 2, choices = [["HS", "High School"],
                                                         ["C", "College"],
                                                         ["G", "Graduate"],
                                                         ["PG", "Post Graduate"]]
                              )

class MyInstitution(models.Model):
    name = models.CharField(max_length=50)
    accredited = models.BooleanField()
```

(continues on next page)

(continued from previous page)

```
    def __str__(self):
        return "%s" % self.name
```

To use **django-modelqueryform** in a project import it into forms.py:

```
import modelqueryform
```

Then we can use it as a Base class for your forms:

```
class MyModelQueryForm(modelqueryform.ModelQueryForm):
    model = MyModel
    inclue = ['age','employed','degree']
```

Thats it! Instantiating MyModelQueryForm gives us a form with 3 widgets

- Age (*RangeField* using a *RangeWidget*)
- Employed (*MultipleChoiceField* using a *CheckboxSelectMultiple* widget)
- Degree (*MultipleChoiceField* using a *CheckboxSelectMultiple* widget)

Once the form is POSTed to the view it is used to filter your model:

```
query_form = MyModelQueryForm(request.POST)
my_models = query_form.process()
```

process([data_set=None]) generates a Q object which is a logical AND of the Q objects generated for each widget. It uses the resulting Q object to filter the associated model class.

Note: *process()* optionally accepts a QuerySet of a model class 'x' where isinstance(x, 'form model class') is True If no QuerySet is passed, the Q object will run against model.objects.all()

Using *pretty_print_query()* you get a dict() of the form {str(field.label): str(field values)} to parse into a template:

```
query_form = MyModelQueryForm(request.POST)
query_parameters = query_form.pretty_print_query()
```

pretty_print_query() also accepts an argument *fields_to_print*, a list of names that must be a subset of *self.changed_data*.

5.2.1 Working with Relations

django-modelqueryform can work with realtionship fields in two different ways, either following the relation or using the relationship field as a choice field.

Let's add a new field to *MyModel* from the example above:

```
class MyModel(models.Model):
    ...
    institution = models.ForeignKey('MyInstitution')
```

If we want our users to be able to select for (non)-accredited institions we would instantiate the form like so:

```
class MyModelQueryForm(modelqueryform.ModelQueryForm):
    model = MyModel
    inclue = ['age','employed','degree', 'institution__accredited']
    traverse_fields = ['institution',]
```

Alternatively we can use the relationship field as a *MultipleChoiceField*:

```
class MyModelQueryForm(modelqueryform.ModelQueryForm):
    model = MyModel
    inclue = ['age','employed','degree', 'institution']
```

Warning: To make the choices for a relationship field, **django-modelqueryform** does an *objects.distinct()* call. Be aware of the size of the resulting QuerySet

5.2.2 Defaults

djagno-modelqueryform tries to provide meaningful default where it can. Default widgets, Q objects, and print representation exist for model fields that are stored as numeric values or have choices (either defined or by default, ie. BooleanField(s))

Note: See *Customization* for how to handle field type that don't have defaults

Default Fields

Model Field	Form Field/Widget	Q Object	Print Representation
AutoField	*RangeField* / *RangeWidget*	AND([field__gte=min],[field__lte=max]);OR(field__isnull=True)	'MIN - MAX [(include empty values)]'
BigIntegerField	*RangeField* / *RangeWidget*	AND([field__gte=min],[field__lte=max]);OR(field__isnull=True)	'MIN - MAX [(include empty values)]'
BinaryField			
BooleanField	*MultipleChoiceField* / *CheckboxSelectMultiple*	OR([field=value],...)	'CHOICE1,CHOICE2,...CHOICEn'
CharField			
CommaSeparatedIntegerField			
DateField			
DateTimeField			
DecimalField	*RangeField* / *RangeWidget*	AND([field__gte=min],[field__lte=max]);OR(field__isnull=True)	'MIN - MAX [(include empty values)]'
EmailField			
FileField			
FilePathField			
FloatField	*RangeField* / *RangeWidget*	AND([field__gte=min],[field__lte=max]);OR(field__isnull=True)	'MIN - MAX [(include empty values)]'
ImageField			
IntegerField	*RangeField* / *RangeWidget*	AND([field__gte=min],[field__lte=max]);OR(field__isnull=True)	'MIN - MAX [(include empty values)]'
IPAddressField			
GenericIPAddressField			
NullBooleanField	*MultipleChoiceField* / *CheckboxSelectMultiple*	OR([field=value],...)	'CHOICE1,CHOICE2,...CHOICEn'
PositiveIntegerField	*RangeField* / *RangeWidget*	AND([field__gte=min],[field__lte=max]);OR(field__isnull=True)	'MIN - MAX [(include empty values)]'
PositiveSmallIntegerField	*RangeField* / *RangeWidget*	AND([field__gte=min],[field__lte=max]);OR(field__isnull=True)	'MIN - MAX [(include empty values)]'
SlugField			
SmallIntegerField	*RangeField* / *RangeWidget*	AND([field__gte=min],[field__lte=max]);OR(field__isnull=True)	'MIN - MAX [(include empty values)]'
TextField			
TimeField			
URLField			
ForeignKey	*MultipleChoiceField* / *CheckboxSelectMultiple*	OR([field=value],...)	'CHOICE1,CHOICE2,...CHOICEn'
ManyToManyField	*MultipleChoiceField* / *CheckboxSelectMultiple*	OR([field=value],...)	'CHOICE1,CHOICE2,...CHOICEn'
OneToOneField	*MultipleChoiceField* / *CheckboxSelectMultiple*	OR([field=value],...)	'CHOICE1,CHOICE2,...CHOICEn'

5.3 Customization

Customization is neccessary in **django-modelqueryform** in instances where the default FormField and filters are insufficient or not avilable for model fields that you want to expose to querying

Note: There are no defaults for Model fields that are represented as text and have no choices

You can customize three different aspects of **django-modelqueryform** Each of these aspects can be customized either by Model Field or Model Field type

1. Form field builder
    - *build_FIELD(model_field)*
    - *build_type_FIELDTYPE(model_field)*
2. Filter builder
    - *filter_FIELD(field_name, values)*
    - *filter_type_FIELDTYPE(field_name, values)*
3. Pretty Print builder
    - *print_FIELD(field_name, values)*
    - *print_type_FIELDTYPE(field_name, values)*

Warning: For fields that have no default you must implement a field builder and a filter builder

For these examples we will use the following Model:

```
class MyModel(models.Model):
   first_name = models.CharField(max_length=15)
   last_name = models.CharField(max_length=15)
```

And the following ModelQueryForm:

```
class MyModelQueryForm(modelqueryform.ModelQueryForm):
    model = MyModel
    inclue = ['first_name','last_name']
```

5.3.1 Form Field Builder

This should return a form field object.

By Name:

```
def build_first_name(model_field):
   return CharField(label=model_field.verbose_name,
                    max_length=model_field.max_length,
                    required=False
   )
```

Note: If this is all we customize for the example MyModelQueryForm it will raise a NotImplementedError because last_name does not have a field builder

By Type:

```
def build_type_charfield(model_field):
   return CharField(label=model_field.verbose_name,
                    max_length=model_field.max_length,
                    required=False
   )
```

Note: No NotImplementedError because this covers the type for both first_name and last_name If there is a name based builder and a type based builder for a field the named builder takes precedence

5.3.2 Filter Builder

This should return a Q object.

By Name:

```
def filter_first_name(field_name, values):
   return Q(**{field_name + '__iexact': values})
```

By Type:

```
def filter_type_charfield(field_name, values):
   return Q(**{field_name + '__contains': values})
```

5.3.3 Pretty Print Builder

By Name:

```
def print_first_name(field_name, values):
   return "Matches %s" % values
```

By Type:

```
def print_type_charfield(field_name, values):
   return "Contains %s" % values
```

5.4 Reference

5.4.1 ModelQueryField

class modelqueryform.forms.**ModelQueryForm**(**args*, ***kwargs*)

ModelQueryForm builds a django form that allows complex filtering against a model.

Variables

- **model** (*Model*) – Model to be filtered
- **include** (*list*) – Field names to be included using the standard orm naming

Raises **ImproperlyConfigured** – If *model* is missing

_build_form(*model*, *field_prepend=None*)
Iterates through model fields to generate modelqueryform fields matching *self.include* Recursively called to correctly build relationship spanning form fields

Parameters

- **model** (*django.db.model*) – Current model to inspect. Alwasy starts with *self.model*
- **field_prepend** (*str*) – Relation field name if using *self.traverse*

_build_form_field(*model_field*, *name*)
Build a form field for a given model field

Parameters

- **model_field** (*django.db.models.fields*) – field that the resulting form field will filter
- **name** (*String*) – The name for the form field (will match a value in self.include)

They type of FormField built is determined in the following order:

1. *build_FIELD(model_field)* (FIELD is the ModelField name)
2. *build_type_FIELD(model_field)* (FIELD is the ModelField type .lower() eg. 'integerfield', charfield', etc.)
3. *modelqueryform.utils.get_multiplechoice_field()* if model_field has .choices
4. *modelqueryform.utils.get_range_field()* if the ModelField type is in *self.numeric_fields()*
5. *modelqueryform.utils.get_multiplechoice_field()* if the ModelField type is in *self.choice_fields()*
6. *modelqueryform.utils.get_multiplechoice_field()* if the ModelField type is in *self.rel_fields()*

> **Warning:** You must define either *build_FIELD(model_field)* or *build_type_FIELD(model_field)* for ModelFields that do not use a *RangeField* or *MultipleChoiceField*

Returns FormField

Raises **NotImplementedError** – For fields that do not have a default *ModelQueryForm* field builder and no custom

field builder can be found

_test_filter_func_is_Q(*filter_func*)
Make sure that a filter is a Q object

Parameters **filter_func** (*Q*) – Object to test

Raises **TypeError** – if filter is not a Q object

build_query_from_filters(*filters*)
Generate a Q object that is a logical AND of a list of Q objects

> **Note:** Override this method to build a more complex Q object than AND(filters.values())

Parameters **filters** (*dict*) – Dict of {Form field name: Q object,...}

Returns Q AND(filters.values())

Raises **TypeError** – if any value in the filters dict is not a Q object

choice_fields()
Get a list of model fields backed by choice values (Boolean types)

Returns list Model Field types that are backed by a boolean

get_filters()
Get a dict of the POSTed form values as Q objects Form fields will be evaluated in the following order to generate a Q object:

1. *filter_FIELD(field_name, values)* (FIELD is the ModelField name)

#. *filter_type_FIELD(field_name, values)* (FIELD is the ModelField type .lower() eg. 'integerfield', charfield', etc.) #. *modelqueryform.utils.get_range_field_filter()* if the FormField is a RangeField #. *modelqueryform.utils.get_multiplechoice_field_filter()* if the FormField is a MultipleChoiceField

> **Warning:** You must define either *filter_FIELD(field, values)* or *filter_type_FIELD(field, values)* for ModelFields that do not use a *RangeField* or *MultipleChoiceField*

Returns Dict {Form field name: Q object,...}

Raises **NotImplementedError** – For fields that do not have a default *ModelQueryForm* filter builder and no

custom filter builder can be found

get_multichoice_field_print(*form_field*, *cleaned_field_data*)
Default string representation of multichoice field

Parameters

- **form_field** – FormField
- **cleaned_field_data** (*dict*) – the cleaned_data for the field

Returns str Comma delimited get_display_FIELD() for selected choices

get_range_field_print(*form_field*, *cleaned_field_data*)
Default string representation of multichoice field

Parameters

- **form_field** – FormField (Unused)
- **cleaned_field_data** (*dict*) – the cleaned_data for the field

Returns str "MIN - MAX [(include empty values)]"

get_related_choices(*model_field*)
Make choices from a related

Parameters **model_field** (*ForeignKey, OneToOneField, ManyToManyField*) – Field to generate choices from

Returns list [[field.pk, field.__str__()],...]

Raises **TypeError** – If *model_field* is not a relationship type

numeric_fields()
Get a list of model fields backed by numeric values

Returns list Model Field types that are backed by a numeric

pretty_print_query(*fields_to_print=None*)
Get an OrderedDict to facilitate printing of generated filter

Parameters **fields_to_print** (*list*) – List of names in changed_data

Note: If fields_to_print == None, self.changed_data is used

Returns dict {form field name: string representation of filter,. . . }

Raises **NotImplementedError** – For fields that do not have a default print builder and no custom print builder

can be found :raises ValueError: if any name in the field_to_print is not in self.changed_data

process(*data_set=None*)
Filter a QuerySet with the POSTed form values

Parameters **data_set** (*QuerySet (Same Model class as self.model)*) – QuerySet to filter against

Note: If data_set == None, self.model.objects.all() is used

Returns QuerySet data_set.filter(Q object)

Raises

- **ImproperlyConfigured** – No *data_set* to filter
- **TypeError** – *data_set* is not an instance (using *isinstance()*) of *self.model*

query_hash()
Get an md5 hexdigest of the pretty_print_query().

Note: Useful for caching results of a given query

Returns str 32 char md5.hexdigest()

rel_fields()
Get a list of related model fields

Returns list Model Field types that are relationships

5.4.2 RangeField

class modelqueryform.widgets.**RangeField**(*model*, *field*, **args*, ***kwargs*)

5.4.3 RangeWidget

class modelqueryform.widgets.**RangeWidget**(*allow_null=False*, *attrs=None*, *mode=0*)

Build a MultiWidget with 3 fields: TextInput with a "min" attribute TextInput with a "max" attribute Checkbox to include/exclude None values

5.4.4 Utils

modelqueryform.utils.**get_choices_from_distinct**(*model*, *field*)
Generate a list of choices from a distinct() call.

Parameters

- **model** (*django.db.models.Model*) – Model to use
- **field** (*django Model Field*) – Field whose .distinct values you want

Returns list – the distinct values of the field in the model

modelqueryform.utils.**get_multiplechoice_field**(*field*, *choices*)
Generate a MultipleChoiceField form element

Parameters

- **field** (*django model field*) – Model Field to use
- **choices** (*iterable*) – List of choices for form field

Returns *MultipleChoiceField*

Raises ValueError

modelqueryform.utils.**get_multiplechoice_field_filter**(*field*, *values*)
Generate a model filter from a POSTed MultipleChoiceField

Parameters

- **field** (*string*) – orm field name
- **values** (*list*) – Selected values

Returns Q – (OR(field: value),...)

modelqueryform.utils.**get_range_field**(*model*, *field*, *name*)
Generate a RangeField form element

Parameters

- **model** (*django.db.models.Model*) – Model to generate a form element for
- **field** (*django model field*) – Model Field to use
- **name** – Name to use for the form field
- **name** – string

Returns *RangeField*

modelqueryform.utils.**get_range_field_filter**(*field*, *values*)
Generate a model filter from a POSTed RangeField

Parameters

- **field** (*string*) – orm field name

- **values** (*dict*) – *RangeField* values dict

Returns Q – AND(OR(field__gte: min, field__lte: max),(field__isnull: allow_empty)

modelqueryform.utils.**traverse_related_to_field**(*field_name*, *model*)
Given an orm relational representation 'relational_field__field_name' and the base model of the relation, return the actual terminal Field

5.5 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.5.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/ckirby/django-modelqueryform/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

django-modelqueryform could always use more documentation, whether as part of the official django-modelqueryform docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/ckirby/django-modelqueryform/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.5.2 Get Started!

Ready to contribute? Here's how to set up *django-modelqueryform* for local development.

1. Fork the *django-modelqueryform* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/django-modelqueryform.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv django-modelqueryform
$ cd django-modelqueryform/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 modelqueryform tests
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, and 3.4, and for PyPy. Check https://travis-ci.org/ckirby/django-modelqueryform/pull_requests and make sure that the tests pass for all supported Python versions.

5.5.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_modelqueryform
```

5.6 Credits

5.6.1 Development Lead

- Chaim Kirby <chaim.kirby@gmail.com>

5.6.2 Contributors

None yet. Why not be the first?

Python Module Index

m

Index

Symbols

B

C

G

M

N

P

Q

R

T