django-flexquery

Release 4.2.0

Robert Schindler

Apr 06, 2020

CONTENTS

CHAPTER

ONE

Q

The `Q` implementation provided by this library contains a simple addition to that of Django.

Creating a `Q` object works as usual:

```
>>> from django_flexquery import Q
>>> q = Q(size__lt=10)
>>> q
<Q: (AND: ('size__lt', 10))>
```

But this implementation adds a `prefix()` method, which allows prefixing some related field's name to the lookup keys of an existing `Q` object. Since `Q` objects can be nested, this is done recursively.

An example:

```
>>> q.prefix("fruits")
<Q: (AND: ('fruits__size__lt', 10))>
```

Nothing more to it. The real power comes when using these `Q` objects with `FlexQuery`.

1.1 API

Extended Q implementation with support for prefixing lookup keys.

class django_flexquery.q.**Q**(**args*, *_connector=None*, *_negated=False*, ***kwargs*)

A custom Q implementation that allows prefixing existing Q objects with some related field name dynamically.

prefix(*prefix*)

Recursively copies the Q object, prefixing all lookup keys.

The prefix and the existing filter key are delimited by the lookup separator __. Use this feature to delegate existing query constraints to a related field.

Parameters **prefix** (*str*) – Name of the related field to prepend to existing lookup keys. This isn't restricted to a single relation level, something like "tree__fruit" is perfectly valid as well.

Returns Q

CHAPTER

TWO

FLEXQUERY

The `FlexQuery` class provides a decorator for functions declared on a custom `Manager` or `QuerySet`:

```
from django_flexquery import FlexQuery, Manager, Q, QuerySet

# It's crucial to inherit from the QuerySet class of django_flexquery, because
# the FlexQuery's wouldn't make it over to a Manager derived using as_manager()
# with the stock Django implementation. That's the only difference however.
class FruitQuerySet(QuerySet):
    # Declare a function that generates a Q object.
    # base is a copy of the base QuerySet instance. It's not needed most of
    # the time unless you want to embed a sub-query based on the current QuerySet
    # into the Q object.
    @FlexQuery.from_func
    def large(base):
        return Q(size__gte=10)
```

`FruitQuerySet.large` now is a sub-type of `FlexQuery` encapsulating the custom function.

You can then derive a `Manager` from `FruitQuerySet` in two ways, using the known Django API:

```
# Either use from_queryset() of the Manager class provided with this module.
class FruitManager(Manager.from_queryset(FruitQuerySet)):
    ...

# Or, if you don't want to add additional manager-only methods, create a Manager
# instance inside your model definition straight away.
class Fruit(Model):
    objects = FruitQuerySet.as_manager()
    ...
```

When we assume such a `Manager` being the default manager of a `Fruit` model with a `size` field, we can now perform the following queries:

```
Fruit.objects.large()
Fruit.objects.filter(Fruit.objects.large.as_q())
```

Internally, this is made possible by some metaclass and descriptor magic instantiating the `FlexQuery` type whenever it is accessed as class attribute of a `Manager` or `QuerySet` object. The resulting `FlexQuery` instance will be tied to its owner and use that for all its filtering.

A `FlexQuery` instance is directly callable (`Fruit.objects.large()`), which just applies the filters returned by our custom Q function to the base `QuerySet`. This is a well-known usage pattern you might have come across often when writing custom Django model managers or querysets.

However, `FlexQuery` also comes with an `as_q()` method, which lets you access the `Q` object directly (`Fruit.objects.filter(Fruit.objects.large.as_q())`). The `FlexQuery` can mediate between these two and deliver what you need in your particular situation.

2.1 Conversion Costs

Providing a standalone `QuerySet` filtered by the `Q` from a supplied `Q` function is a cheap operation. The `Q` object generated by your custom function is simply applied to the base using `QuerySet.filter()`, resulting in a new `QuerySet` you may either evaluate straight away or use to create a sub-query.

2.2 Why do I Need This?

This approach enables you to declare logic for filtering once with the `Manager` or `QuerySet` of the model it belongs to. When combined with the prefix() method of the extended Q object implementation, related models can then simply fetch the generated Q object and prefix it with the related field's name in order to reuse it in their own filtering code, without needing sub-queries. Think of something like:

```
class TreeQuerySet(QuerySet):
    @FlexQuery.from_func
    def having_ripe_apples(base):
        return Q(kind="apple") & Fruid.objects.large.as_q().prefix("fruits")
```

2.3 API

This module provides a convenient way to declare custom filtering logic with Django's model managers in a reusable fashion using Q objects.

class django_flexquery.flexquery.**FlexQuery**(*base*)

Flexibly provides model-specific query constraints as an attribute of Manager or QuerySet objects.

When a sub-type of FlexQuery is accessed as class attribute of a Manager or QuerySet object, its metaclass, which is implemented as a descriptor, will automatically initialize and return an instance of the FlexQuery type bound to the holding Manager or QuerySet.

__call__(**args*, ***kwargs*)

Filters the base queryset using the provided function, relaying arguments.

Returns QuerySet

as_q(**args*, ***kwargs*)

Returns the result of the configured function, relaying arguments.

Returns Q

call_bound(**args*, ***kwargs*)

Calls the provided function with `self.base.all()` as first argument.

This may be overwritten if arguments need to be preprocessed in some way before being passed to the custom function.

Returns Q

classmethod from_func(*func=None*, ***attrs*)
Creates a `FlexQuery` sub-type from a function.

This classmethod can be used as decorator. As long as `func` is `None`, a `functools.partial` with the given keyword arguments is returned.

Parameters

- **func** (*function | None*) – function taking a base `QuerySet` and returning a `Q` object
- **attrs** – additional attributes to set on the newly created type

Returns InitializedFlexQueryType | functools.partial

Raises TypeError – if `func` is no function

class django_flexquery.flexquery.**Manager**
Use this class' `from_queryset` method if you want to derive a `Manager` from a `QuerySet` with `FlexQuery` members. If Django's native `Manager.from_queryset` was used instead, those members would be lost.

class django_flexquery.flexquery.**QuerySet**(*model=None*, *query=None*, *using=None*, *hints=None*)
Adds support for deriving a `Manager` from a `QuerySet` class via `as_manager`, preserving `FlexQuery` members.

CHAPTER

THREE

CONTRIB.USER_BASED

FlexQuery variant that restricts the base QuerySet for a given user.

class django_flexquery.contrib.user_based.**UserBasedFlexQuery**(*base*)

This is a slightly modified `FlexQuery` implementation, accepting either a `django.http.HttpRequest` or a user object as argument for the custom function and passing the user through.

When no user (or `None`) is given, the behavior is determined by the `no_user_behavior` attribute, which may be set to one of the following constants defined on the `UserBasedFlexQuery` class:

- `NUB_ALL`: don't restrict the queryset
- `NUB_NONE`: restrict to the empty queryset (default)
- `NUB_PASS`: let the custom function handle a `user` of `None`

If the `pass_anonymous_user` attribute is changed to `False`, `django.contrib.auth.models.AnonymousUser` objects are treated as if they were `None` and the configured no-user behavior comes to play.

Because it can handle an `HttpRequest` directly, instances of this `FlexQuery` may also be used in conjunction with the django_filters library as the `queryset` parameter of filters.

call_bound(*user*, **args*, ***kwargs*)

Calls the custom function with a user, followed by the remaining arguments.

Parameters **user** (*django.contrib.auth.models.User | django.http.HttpRequest | None*) – User to filter the queryset for

Returns Q

This library aims to provide a new way of declaring reusable QuerySet filtering logic in your Django project, incorporating the DRY principle and maximizing user experience and performance by allowing you to decide between sub-queries and JOINs.

Its strengths are, among others:

- Easy to learn in minutes
- Cleanly integrates with Django's ORM
- Small code footprint, hard for bugs to hide - ~150 lines of code (LoC)
- 100% test coverage
- Fully documented code, formatted using the excellent Black Code Formatter.

When referencing a related model in a database query, you usually have the choice between using a JOIN (`X.objects.filter(y__z=2)`) or performing a sub-query (`X.objects.filter(y__in=Y.objects.filter(z=2))`).

We don't want to judge which one is better, because that depends on the concrete query and how the database engine in use optimizes it. In many cases, it will hardly make a noticeable difference at all. However, when the amount of data grows, doing queries right can save you and the users of your application several seconds, and that is what django-flexquery is for.

CHAPTER

FOUR

REQUIREMENTS

Continuous integration ensures compatibility with Python 3.7 + Django 2.2 and 3.0.

CHAPTER

FIVE

INSTALLATION

```
pip install django-flexquery
```

No changes to your Django settings are required; no `INSTALLED_APPS`, no `MIDDLEWARE_CLASSES`.

PYTHON MODULE INDEX

d

INDEX

Symbols

A

C

D

F

M

P

Q

U