django-queryable-properties Documentation

Release 1.9.2

Marcus Klöpfel

Mar 27, 2024

CONTENTS

Write Django model properties that can be used in database queries.

CHAPTER

ONE

INTRODUCTION

django-queryable-properties attempts to offer a unified pattern to help with a common and recurring problem:

1. Properties are added to a model class which are based on model field values of its instances. These properties may even be based on some related model objects and therefore perform additional database queries.
2. The code base grows and needs to be able to satisfy new demands.
3. The logic of the properties from step 1 would now be useful in batch operations (read: queryset operations), making the current implementation less feasible, as it would likely perform additional queries per object in a queryset operation. Also, regular properties do of course not offer queryset features like filtering, ordering, etc.

Since Django offers a lot of powerful options when working with querysets (like `select_related`, annotations, etc.), it is generally not an issue to solve these problems and implement a solution, which will likely be based on one of the following options:

- Performing special annotations only in the exact places that they are needed or in utility functions/methods.
- Implementing a custom model manager/queryset class to allow the usage of these special annotations whenever dealing with a queryset.
- Using `queryset.alias()` to build up a collection of available queryset annotations that resemble the properties (requires Django 3.2 or higher).

While especially the latter options are not *wrong*, they do require some boilerplate and will likely split up the business logic into multiple parts (e.g. the property for single objects is implemented on the model class while the corresponding annotation for batch operations is part of a queryset class), making it harder to apply changes to the business logic to all required parts. Solutions like these are genereally also not really reusable unless a lot of effort is put into them. For example, even manager/queryset extensions will likely only work on the exact model they were designed for and will therefore not be usable from other models via relations.

Important: Starting with Django 5.0, GeneratedFields may be used to cover many of the use-cases of *django-queryable-properties*. Since they are native Django fields, the disadvantages mentioned above do not apply to them.

---

django-queryable-properties does, in fact, not remove the general necessity of implementing the business logic in (at least) 2 parts - one for individual objects and one for batch/queryset operations. Instead, it aims to remove as much boilerplate as possible and offers an option to implement said parts in one place - just like the `getter` and `setter` of a regular property are implemented together. On top of that, queryable properties cannot only be used in querysets for the model they were defined on, but can also be accessed through relations when querying via other models.

1.1 Examples in this documentation

All parts of this documentation contain a few simple examples to show how to take advantage of all the features of queryable properties. For consistency, all of those examples are based on a few simple Django models, which are shown in the following code block. They represent models storing data for a version management system for applications, which in this over-simplified case only store which versions of an application exist. While this may not be the best real-world example, it can demonstrate how to work with queryable properties quite well.

```
from django.db import models


class Category(models.Model):
    """Represents a category for applications."""
    name = models.CharField(max_length=255)


class Application(models.Model):
    """Represents a named application."""
    categories = models.ManyToManyField(Category, related_name='applications')
    name = models.CharField(max_length=255)


class ApplicationVersion(models.Model):
    """Represents a version of an application using a major and minor version number."""
    application = models.ForeignKey(Application, on_delete=models.CASCADE, related_name=
↪'versions')
    major = models.PositiveIntegerField()
    minor = models.PositiveIntegerField()
```

CHAPTER

TWO

INSTALLATION

django-queryable-properties is available for installation via `pip` on PyPI:

```
pip install django-queryable-properties
```

To use the features of this package, simply use the classes and functions as described in this documentation. There is no need to add the package to the `INSTALLED_APPS` setting.

2.1 Dependencies

django-queryable-properties supports and is tested against the following Django versions and their corresponding supported Python versions:

Django version	Supported Python versions
5.0	3.12, 3.11, 3.10
4.2	3.12, 3.11, 3.10, 3.9, 3.8
4.1	3.11, 3.10, 3.9, 3.8
4.0	3.10, 3.9, 3.8
3.2	3.10, 3.9, 3.8, 3.7, 3.6
3.1	3.9, 3.8, 3.7, 3.6
3.0	3.9, 3.8, 3.7, 3.6
2.2	3.9, 3.8, 3.7, 3.6, 3.5
2.1	3.7, 3.6, 3.5
2.0	3.7, 3.6, 3.5, 3.4
1.11	3.7, 3.6, 3.5, 3.4, 2.7
1.10	3.5, 3.4, 2.7
1.9	3.5, 3.4, 2.7
1.8	3.5, 3.4, 2.7
1.7	3.4, 2.7
1.6	2.7
1.5	2.7
1.4	2.7

Support for certain Python versions was added to some Django versions retrospectively in a patch version. The tests run against the most recent patch version for each Django release.

Upcoming versions may also work, but are not officially supported as long as they are not added to the test setup.

CHAPTER

THREE

BASICS

3.1 Implementing queryable properties

There are two ways to implement a queryable property:

- Using decorated methods directly on the model class (just like regular properties)
- Implementing the queryable property as a class and using its instances as class attributes on the model class (much like model fields)

Say we'd want to implement a queryable property for the `ApplicationVersion` example model that simply returns the combined version information as a string. The two following sections show how to implement such a queryable property - for the sake of simplicity, the examples only show how to implement a getter and setter (which could also be implemented using a regular property). The following chapters of this documentation will show all available decorators, mixins and implementable methods in detail.

3.1.1 Decorator-based approach

The decorator-based approach uses the class *queryable_properties.properties.queryable_property* and its methods as decorators:

```
from django.db import models
from queryable_properties.properties import queryable_property


class ApplicationVersion(models.Model):
    ...

    @queryable_property
    def version_str(self):
        return '{major}.{minor}'.format(major=self.major, minor=self.minor)

    @version_str.setter
    def version_str(self, value):
        # Don't implement any validation to keep the example simple.
        self.major, self.minor = value.split('.')
```

Using the decorator methods without actually decorating

Python's regular properties also allow to define properties without using `property` as a decorator. To do this, the individual methods that should make up the property can be passed to the `property` constructor:

```
class MyClass(object):

    def get_x(self):
        return self._x

    def set_x(self, value):
        self._x = value

    x = property(get_x, set_x)
```

Queryable properties do **not** allow to do this in the same way because of two reasons:

- To encourage implementing properties using decorators, which is cleaner and makes code more readable.
- Since queryable properties have a lot more functionality and options than regular properties, they would need to support a huge number of constructor parameters, which would make the constructor too complex and harder to maintain.

However, there are use cases where an option similar to the non-decorator usage of regular properties would be nice to have, e.g. when implementing a property without a getter or when the individual getter/setter methods are already present and cannot be easily deprecated in favor of the property. This is why queryable properties do support this form of defining a property - but in a slightly different way: the decorator methods can simply be chained together (this also works for all decorators introduced in later chapters).

```
from django.db import models
from queryable_properties.properties import queryable_property


class ApplicationVersion(models.Model):
    ...

    def get_version_str(self):
        return '{major}.{minor}'.format(major=self.major, minor=self.minor)

    def set_version_str(self, value):
        # Don't implement any validation to keep the example simple.
        self.major, self.minor = value.split('.')

    version_str = queryable_property(get_version_str).setter(set_version_str)
```

By not passing a getter function to the `queryable_property` constructor, a queryable property without a getter can be defined (`queryable_property().setter(set_version_str)` for the example above). This can even be used to make a getter-less queryable property while still decorating the setter (or mixing and matching chaining and decorating in general):

```
from django.db import models
from queryable_properties.properties import queryable_property


class ApplicationVersion(models.Model):
```

(continues on next page)

(continued from previous page)

```
    ...

    version_str = queryable_property()  # Property without a getter

    @version_str.setter
    def version_str(self, value):
        # Don't implement any validation to keep the example simple.
        self.major, self.minor = value.split('.')
```

3.1.2 Class-based approach

Using the class-based approach, the queryable property is implemented as a subclass of *queryable_properties.properties.QueryableProperty*:

```
from django.db import models
from queryable_properties.properties import QueryableProperty, SetterMixin


class VersionStringProperty(SetterMixin, QueryableProperty):

    def get_value(self, obj):
        return '{major}.{minor}'.format(major=obj.major, minor=obj.minor)

    def set_value(self, obj, value):
        # Don't implement any validation to keep the example simple.
        obj.major, obj.minor = value.split('.')


class ApplicationVersion(models.Model):
    ...

    version_str = VersionStringProperty()
```

3.1.3 Common property arguments

Queryable properties that are created using either approach take additional, common keyword arguments that can be used to configure property instances further. These are:

verbose_name
: A human-readable name for the property instance, similar to the verbose name of an instance of one of Django's model fields. Used for UI representations of queryable properties. If no verbose name is set up for a property, one will be generated based on the property's name.

For both the class-based and the decorator-based approach, these keyword arguments can be set via their respective constructor. For the example property above, this could look like the following example:

```
from django.utils.translation import gettext_lazy as _


class ApplicationVersion(models.Model):
    ...
```

(continues on next page)

(continued from previous page)

```
    # Class-based
    version_str = VersionStringProperty(verbose_name=_('Full Version Number'))

    # Decorator-based
    @queryable_property(verbose_name=_('Full Version Number'))
    def version_str(self):
        ...
```

3.1.4 When to use which approach

It all depends on your needs and preferences, but a general rule of thumb is using the class-based approach to implement re-usable queryable properties or to be able to use inheritance. It would also be pretty easy to write parameterizable property classes by adding parameters to their `__init__` methods.

Class-based implementations come, however, with the small disadvantage of having to define the property's logic outside of the actual model class (unlike regular property implementations). It would therefore probably be preferable to use the decorator-based approach for unique, non-reusable implementations.

3.2 Enabling queryset operations

To actually interact with queryable properties in queryset operations, the queryset extensions provided by *django-queryable-properties* must be used since regular querysets cannot deal with queryable properties on their own.

The following sections describe how to properly set this up to either use the extensions by either applying them to querysets of models in general via managers or by creating querysets with the queryable properties extensions on demand.

3.2.1 Defining managers on models

The most common way to use the queryset extensions is by defining a manager that produces querysets with queryable properties functionality. The easiest way to do this is by simply using the *queryable_properties.managers.QueryablePropertiesManager*:

```
from queryable_properties.managers import QueryablePropertiesManager


class ApplicationVersion(models.Model):
    ...

    objects = QueryablePropertiesManager()
```

This manager allows to use the queryable properties in querysets created by this manager (e.g. via `ApplicationVersion.objects.all()`).

For scenarios where querysets or managers need other extensions or base classes, *django-queryable-properties* also offers a queryset class as well as mixins for managers or querysets that can be combined with other base classes:

- Queryset class: *queryable_properties.managers.QueryablePropertiesQuerySet*
- Queryset mixin: *queryable_properties.managers.QueryablePropertiesQuerySetMixin*

- Manager mixin: *queryable_properties.managers.QueryablePropertiesManagerMixin*

When implementing custom queryset classes, a manager class can be generated from the queryset class using `CustomQuerySet.as_manager()` or `CustomManager.from_queryset(CustomQuerySet)`.

Warning: Since queryable property interaction in querysets is tied to the specific extensions, those extensions are also required when trying to access queryable properties on related models. This means that using the manager approach, all models from which queries that interact with queryable properties are performed need to use a manager as described above, even if a model doesn't implement its own queryable properties.

For example, if queryset filtering was implemented for the `version_str` property shown above, it could also be used in querysets of the `Application` model like this:

```
Application.objects.filter(versions__version_str='1.2')
```

To make this work, the `objects` manager of the `Application` model must also be a `QueryablePropertiesManager`, even if the model does not define queryable properties of its own.

If using a special manager just to access queryable properties on related models is not desirable, then the following approaches to apply the queryable properties extensions on demand should offer an alternative.

3.2.2 Creating managers/querysets on demand

The non-mixin classes provided by *django-queryable-properties* also allow to create managers or querysets on demand, regardless of the presence of a manager with queryable properties extensions on the corresponding model. Both the *queryable_properties.managers.QueryablePropertiesManager* and the *queryable_properties.managers.QueryablePropertiesQuerySet* offer a `get_for_model` method for this purpose:

```
from queryable_properties.managers import QueryablePropertiesManager,␣
↪QueryablePropertiesQuerySet

# Create an ad hoc manager that produces querysets with queryable property extensions␣
↪for the given model.
ad_hoc_manager = QueryablePropertiesManager.get_for_model(MyModel)
# Create an ad hoc queryset with queryable property extensions for the given model.
ad_hoc_queryset = QueryablePropertiesQuerySet.get_for_model(MyModel)
```

Note: Querysets created using `QueryablePropertiesQuerySet.get_for_model` use the model's default manager to create the underlying queryset, i.e. the queryset is generated using `model._default_manager.all()` before the queryable properties extensions are applied.

3.2.3 Applying the extensions to existing managers/querysets on demand

There might be scenarios where interacting with queryable properties is desired in an existing queryset or manager. The mixin classes provided by *django-queryable-properties* allow to inject the queryable properties extensions into an existing queryset or manager using their `apply_to` method. Both the *queryable_properties.managers.QueryablePropertiesManagerMixin* and the *queryable_properties.managers.QueryablePropertiesQuerySetMixin* create a copy of the original object in the process, leaving said object untouched.

```
from queryable_properties.managers import QueryablePropertiesManagerMixin,␣
↪QueryablePropertiesQuerySetMixin

# Create an ad hoc manager based off the given manager instance that produces querysets␣
↪with queryable property
# extensions for the given model.
ad_hoc_manager = QueryablePropertiesManagerMixin.apply_to(some_manager)
# Create an ad hoc queryset with queryable property extensions for the given model.
some_queryset = MyModel.objects.filter(...).order_by(...)  # A queryset without␣
↪queryable properties features.
ad_hoc_queryset = QueryablePropertiesQuerySetMixin.apply_to(some_queryset)
ad_hoc_queryset.select_properties(...)  # Now queryable properties features can be used.
```

CHAPTER

FOUR

STANDARD PROPERTY FEATURES

Queryable properties offer almost all the features of regular properties while adding some additional options.

4.1 Getter

Queryable properties define their getter method the same way as regular properties do when using the decorator-based approach:

```
from queryable_properties.properties import queryable_property


class ApplicationVersion(models.Model):
    ...

    @queryable_property
    def version_str(self):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=self.major, minor=self.minor)
```

Using the class-based approach, the queryable property's method `get_value` must be implemented instead, taking the model object to retrieve the value from as its only parameter:

```
from queryable_properties.properties import QueryableProperty


class VersionStringProperty(QueryableProperty):

    def get_value(self, obj):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=obj.major, minor=obj.minor)
```

4.1.1 Cached getter

Getters of queryable properties can be marked as cached, which will make them act similarly to properties decorated with Python's/Django's `cached_property` decorator: The getter's code will only be executed on the first access and then be stored, while subsequent calls of the getter will retrieve the cached value (unless the property is reset on a model object, see below).

To use this feature with the decorator-based approach, simply pass the `cached` parameter with the value `True` to the `queryable_property` constructor:

```
from queryable_properties.properties import queryable_property


class ApplicationVersion(models.Model):
    ...

    @queryable_property(cached=True)
    def version_str(self):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=self.major, minor=self.minor)
```

Using the class-based approach, the class attribute `cached` can be set to `True` instead (it would also be possible to set this attribute on individual instances of the queryable property instead):

```
from queryable_properties.properties import QueryableProperty


class VersionStringProperty(QueryableProperty):

    cached = True

    def get_value(self, obj):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=obj.major, minor=obj.minor)
```

Note: All queryable properties that implement annotation will act like cached properties on the result objects of a queryset after they have been explicitly selected. Read more about this in *Selecting annotations*.

Resetting a cached property

If there's ever a need for an exception from using the cache functionality, the cached value of a queryable property on a particular model instance can be reset at any time. This means that the getter's code will be executed again on the next access and the result will be used as the new cached value (since it's still a queryable property marked as cached). To make this as simple as possible, a method `reset_property`, which takes the name of a defined queryable property as parameter, is automatically added to each model class that defines at least one queryable property. If a model class already defines a method with this name, it will *not* be overridden. Queryable properties on objects of such model classes may instead be cleared using the utility function *queryable_properties.utils.reset_queryable_property()*.

To reset the `version_str` property from the example above on an `ApplicationVersion` instance, both of the variants in the following code block can be used (`obj` is an `ApplicationVersion` instance):

```
from queryable_properties.utils import reset_queryable_property  # Required for variant 2

# Variant 1: using the automatically defined method
obj.reset_property('version_str')

# Variant 2: using the utility function
reset_queryable_property(obj, 'version_str')
```

4.2 Setter

Setter methods can be defined in the exact same way as they would be on regular properties when using the decorator-based approach:

```
from queryable_properties.properties import queryable_property


class ApplicationVersion(models.Model):
    ...

    @queryable_property
    def version_str(self):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=self.major, minor=self.minor)

    @version_str.setter
    def version_str(self, value):
        """Set the version fields from a version string."""
        # Don't implement any validation to keep the example simple.
        self.major, self.minor = value.split('.')
```

Using the class-based approach, the queryable property's method `set_value` must be implemented instead, taking the model object to set the fields on as well as the actual value for the property as parameters. It is recommended to use the *queryable_properties.properties.SetterMixin* for class-based queryable properties that define a setter because it defines the actual stub for the `set_value` method. However, using this mixin is not required - a queryable property can be set as long as the `set_value` method is implemented correctly.

```
from queryable_properties.properties import QueryableProperty, SetterMixin


class VersionStringProperty(SetterMixin, QueryableProperty):

    def get_value(self, obj):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=obj.major, minor=obj.minor)

    def set_value(self, obj, value):
        """Set the version fields from a version string."""
        # Don't implement any validation to keep the example simple.
        obj.major, obj.minor = value.split('.')
```

Just like regular properties, queryable properties with setters can also be used via the initializer arguments of their respective model. With both approaches shown above, an `ApplicationVersion` object could therefore be created

like this:

```
version = ApplicationVersion(version_str='1.2')
```

4.2.1 Setter cache behavior

Since queryable properties can be marked as cached, they also come with options regarding the interaction between cached values and setters.

Note: The setter cache behavior is not only relevant for queryable properties that have been marked as cached. Explicitly selected queryable property annotations also behave like cached properties, which means they also make use of this option if their setter is used after they were selected. Read more about this in *Selecting annotations*.

There are 4 options that can be used via constants (which in reality are functions, much like Django's built-in values for the `on_delete` option of `ForeignKey` fields), which can be imported from `queryable_properties.properties`:

`CLEAR_CACHE` (default)
: After the setter is used, a cached value for this property on the model instance is reset. The next use of the getter will therefore execute the getter code again and then cache the new value (unless the property isn't actually marked as cached).

`CACHE_VALUE`
: After the setter is used, the cache for the queryable property on the model instance will be updated with the value that was passed to the setter.

`CACHE_RETURN_VALUE`
: Like `CACHE_VALUE`, but the *return value* of the function decorated with `@<property>.setter` for the decorator-based approach or the `set_value` method for the class-based approach is cached instead. The function/method should therefore return a value when this option is used, as `None` will be cached on each setter usage otherwise.

`DO_NOTHING`
: As the name suggests, this behavior will not interact with cached values at all after a setter is used. This means that cached values from before the setter was used will remain in the cache and may therefore not reflect the most recent value.

To provide a simple example, the setter of the `version_str` property should now be extended to be able to accept values starting with `'V'` (e.g. `'V2.0'` instead of just `'2.0'`) and the newly set value should be cached after the setter was used. Using `CACHE_VALUE` is therefore not a viable option as it would simply cache the value passed to the setter, which may or may not be prefixed with `'V'`, making the getter unreliable as it would return these unprocessed values. Instead, `CACHE_RETURN_VALUE` will be used to ensure the correct getter format for cached values.

To achieve this using the decorator-based approach, the `cache_behavior` parameter of the `setter` decorator must be used:

```
from queryable_properties.properties import CACHE_RETURN_VALUE, queryable_property


class ApplicationVersion(models.Model):
    ...

    @queryable_property(cached=True)
    def version_str(self):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=self.major, minor=self.minor)
```

(continues on next page)

(continued from previous page)

```
    @version_str.setter(cache_behavior=CACHE_RETURN_VALUE)
    def version_str(self, value):
        """Set the version fields from a version string, which is allowed to be prefixed␣
↪with 'V'."""
        # Don't implement any validation to keep the example simple.
        if value.lower().startswith('v'):
            value = value[1:]
        self.major, self.minor = value.split('.')
        return value  # This value will be cached due to CACHE_RETURN_VALUE
```

For the class-based approach, the class (or instance) attribute `setter_cache_behavior` must be set:

```
from queryable_properties.properties import CACHE_RETURN_VALUE, QueryableProperty,␣
↪SetterMixin


class VersionStringProperty(SetterMixin, QueryableProperty):

    cached = True
    setter_cache_behavior = CACHE_RETURN_VALUE

    def get_value(self, obj):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=obj.major, minor=obj.minor)

    def set_value(self, obj, value):
        """Set the version fields from a version string, which is allowed to be prefixed␣
↪with 'V'."""
        # Don't implement any validation to keep the example simple.
        if value.lower().startswith('v'):
            value = value[1:]
        obj.major, obj.minor = value.split('.')
        return value  # This value will be cached due to CACHE_RETURN_VALUE
```

4.3 Deleter

Unlike regular properties, queryable properties do *not* offer a deleter. This is intentional as queryable properties are supposed to be based on model fields, which can't just be deleted from a model instance either. (Nullable) Fields can, however, be "cleared" by setting their value to `None` - but this can just as easily be achieved by using a setter to set this value.

CHAPTER

FIVE

FILTERING QUERYSETS

One of the most basic demands for a queryable property is the ability to be able to use it to filter querysets. Since it is considered the most basic queryset interaction, filtering is thought of as a default part of every queryable property. The class-based approach does therefore not offer a mixin for this operation - the `QueryableProperty` base class defines the method stub already. This does, however, not mean that filtering *must* be implemented - a queryable property works fine without implementing it, as long as we don't try to filter a queryset by such a property.

Note: Implementing how to filter by a queryable property is not necessary for properties that also implement annotating, because an annotated field in a queryset natively supports filtering. Read more about this in *The AnnotationMixin and custom filter implementations*.

5.1 Implementation

5.1.1 One-for-all filter function/method

The simplest way to implement (custom) filtering is using a single function/method that covers all filter functionality.

To implement the one-for-all filter using the decorator-based approach, the property's `filter` method must be used. The following code block contains an example for the `version_str` property from previous examples:

```
from django.db.models import Model, Q
from queryable_properties.properties import queryable_property


class ApplicationVersion(Model):
    ...

    @queryable_property
    def version_str(self):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=self.major, minor=self.minor)

    @version_str.filter
    @classmethod
    def version_str(cls, lookup, value):
        if lookup != 'exact':  # Only allow equality checks for the simplicity of the␣
↪example
            raise NotImplementedError()
```

(continues on next page)

(continued from previous page)

```
        # Don't implement any validation to keep the example simple.
        major, minor = value.split('.')
        return Q(major=major, minor=minor)
```

Note: The `classmethod` decorator is not required, but makes the function look more natural since it takes the model class as its first argument.

To implement the one-for-all filter using the class-based apprach, the `get_filter` method must be implemented. The following code block contains an example for the `version_str` property from previous examples:

```
from django.db.models import Q
from queryable_properties.properties import QueryableProperty


class VersionStringProperty(QueryableProperty):

    def get_value(self, obj):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=obj.major, minor=obj.minor)

    def get_filter(self, cls, lookup, value):
        if lookup != 'exact':  # Only allow equality checks for the simplicity of the␣
↪example
            raise NotImplementedError()
        # Don't implement any validation to keep the example simple.
        major, minor = value.split('.')
        return Q(major=major, minor=minor)
```

In both cases, the function/method to implement takes 3 arguments:

`cls`
: The model class. Mainly useful to implement custom logic in inheritance scenarios.

`lookup`
: The lookup used for the filter as a string (e.g. `'lt'` or `'contains'`). If a filter call is made without an explicit lookup for an equality comparison (e.g. via `ApplicationVersion.objects.filter(version_str='2.0')`), the lookup will be `'exact'`. If a filter call is made with multiple lookups/transforms (like `field__year__gt` for a date field), the lookup will be the combined string of all lookups/transforms (`'year__gt'` for the date example).

`value`
: The value to filter by.

Using either approach, the function/method is expected to return a `Q` object that contains the correct filter conditions to represent filtering by the queryable property using the given lookup and value.

Note: The returned `Q` object may contain filters using other queryable properties on the same model, which will be resolved accordingly.

5.1.2 Lookup-based filter functions/methods

When trying support a lot of different lookups for a (custom) filter implementation, the one-for-all filter can quickly become unwieldy as it will most likely require a big `if`/`elif`/`else` dispatching structure. To avoid this, *django-queryable-properties* also offers a built-in way to spread the filter implementation across multiple functions or methods while assigning one or more lookups to each of them. This can also be useful for implementations that only support a single lookup as it will guarantee that the filter can only be called with this lookup, while a *queryable_properties.exceptions.QueryablePropertyError* will be raised for any other lookup.

Let's assume that the implementation above should also support the `lt` and `lte` lookups. To achieve this with lookup-based filter functions using the decorator-based approach, the `lookups` argument of the `filter` must be used:

```
from django.db.models import Model, Q
from queryable_properties.properties import queryable_property


class ApplicationVersion(Model):
    ...

    @queryable_property
    def version_str(self):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=self.major, minor=self.minor)

    @version_str.filter(lookups=('exact',))
    @classmethod
    def version_str(cls, lookup, value):  # Only ever called with the 'exact' lookup.
        # Don't implement any validation to keep the example simple.
        major, minor = value.split('.')
        return Q(major=major, minor=minor)

    @version_str.filter(lookups=('lt', 'lte'))
    @classmethod
    def version_str(cls, lookup, value):  # Only ever called with the 'lt' or 'lte' lookup.
        # Don't implement any validation to keep the example simple.
        major, minor = value.split('.')
        return Q(major__lt=major) | Q(**{'major': major, 'minor__{}'.format(lookup):␣
↪minor})
```

Note: The `classmethod` decorator is not required, but makes the functions look more natural since they take the model class as their first argument.

To make use of the lookup-based filters using the class-based approach, the *queryable_properties.properties.LookupFilterMixin* (which implements `get_filter`) must be used in conjunction with the *queryable_properties.properties.lookup_filter()* decorator for the individual filter methods:

```
from django.db.models import Q
from queryable_properties.properties import LookupFilterMixin, lookup_filter,␣
↪QueryableProperty


class VersionStringProperty(LookupFilterMixin, QueryableProperty):
```

(continues on next page)

(continued from previous page)

```
    def get_value(self, obj):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=obj.major, minor=obj.minor)

    @lookup_filter('exact')  # Alternatively: @LookupFilterMixin.lookup_filter(...)
    def filter_equality(self, cls, lookup, value):  # Only ever called with the 'exact'␣
↪lookup.
        # Don't implement any validation to keep the example simple.
        major, minor = value.split('.')
        return Q(major=major, minor=minor)

    @lookup_filter('lt', 'lte')  # Alternatively: @LookupFilterMixin.lookup_filter(...)
    def filter_lower(self, cls, lookup, value):  # Only ever called with the 'lt' or 'lte'␣
↪lookup.
        # Don't implement any validation to keep the example simple.
        major, minor = value.split('.')
        return Q(major__lt=major) | Q(**{'major': major, 'minor__{}'.format(lookup):␣
↪minor})
```

For either approach, the individual filter functions/methods must take the same arguments as a one-for-all filter implementation (see above) and return `Q` objects. To support complex lookups (i.e. combinations of transforms and lookups), the full combined lookup string for each supported option must be specified in the decorators (e.g. `'year__gt'`)

It's also possible to define filter functions/methods that handle all remaining lookups for which no explicit function/method was defined. There are two ways to achieve this:

- Using the `queryable_properties.properties.REMAINING_LOOKUPS` constant instead of a lookup name in the `.filter` or `lookup_filter` decorators above (i.e. `@my_property.filter(lookups=(REMAINING_LOOKUPS,))` or `@lookup_filter(REMAINING_LOOKUPS)`) to explicitly register a function/method for all remaining lookups.
- Setting the class (or instance) attribute `remaining_lookups_via_parent` to `True` for the class-based approach or passing `remaining_lookups_via_parent=True` in the `.filter` decorator for the decorator-based approach. This will result in using the `get_filter` implementation of the parent class for all remaining lookups by essentially performing a `super` call and is therefore useful in inheritance scenarios. This can, for example, be used in conjunction with the `AnnotationMixin` to allow to override the filter implementation for certain lookups while relying on the implementation of the `AnnotationMixin` for all remaining lookups. Refer to *The AnnotationMixin and custom filter implementations* for further information.

Caution: Since the `LookupFilterMixin` simply implements the `get_filter` method to perform the lookup dispatching, care must be taken when using other mixins (most notably the `AnnotationMixin` - see *The AnnotationMixin and custom filter implementations*) that override this method as well (the implementations override each other).

This is also relevant for the decorator-based approach as these mixins are automatically added to such properties when they use annotations or lookup-based filters. The order of the mixins for the class-based approach or the used decorators for the decorator-based approach is therefore important in such cases (the mixin applied last wins).

Boolean filters

Boolean queryable properties/filters are a somewhat special and very simple case: There are only 2 possible filter values (`True` and `False`) and there is only one lookup that really makes sense: `exact`. Because boolean filters can be simplified like this, *django-queryable-properties* also has a way to implement them as simple as possible based on lookup-based filters.

Let's assume that a simple property that simply returns whether an application version is the first stable version of its product is to be implemented (for simplicity's sake, we assume that the first stable version uses the number 1.0).

Using the decorator-based approach, this property could be implemented like this (note the `boolean` argument that is used in the `filter` decorator instead of `lookups`):

```
from django.db.models import Model, Q
from queryable_properties.properties import queryable_property


class ApplicationVersion(Model):
    ...

    @queryable_property
    def is_first_stable_version(self):
        """Return True if this application version represents the first stable version.""
↪"
        return self.major == 1 and self.minor == 0

    @is_first_stable_version.filter(boolean=True)
    @classmethod
    def version_str(cls):  # Only ever called with the 'exact' lookup.
        return Q(major=1, minor=0)
```

Note: The `classmethod` decorator is not required, but makes the functions look more natural since they take the model class as their first argument.

Note: The `boolean` and `lookups` arguments are mutually exclusive.

To implement a boolean filter using the class-based approach, the `LookupFilterMixin` must still be used, but this time in conjunction with the *queryable_properties.properties.boolean_filter()* decorator for the filter method:

```
from django.db.models import Q
from queryable_properties.properties import boolean_filter, LookupFilterMixin,␣
↪QueryableProperty


class StableVersionProperty(LookupFilterMixin, QueryableProperty):

    def get_value(self, obj):
        """Return the combined version info as a string."""
        return obj.major == 1 and obj.minor == 0

    @boolean_filter  # Alternatively: @LookupFilterMixin.boolean_filter
    def filter_equality(self, cls):  # Only ever called with the 'exact' lookup.
```

(continues on next page)

(continued from previous page)

```
        # Don't implement any validation to keep the example simple.
        return Q(major=1, minor=0)
```

Some noteworthy points about the `boolean_filter` decorator and the `boolean` argument:

- Using either of the two automatically restricts the lookups the filter can be called with to `exact` as other kinds of lookups don't make much sense in conjunction with boolean filters (essentially equivalent to using `@lookup_filter('exact')` or `lookups=('exact',)`, respectively).
- The decorated methods **do not** take the `lookup` and `value` arguments that any other filter implementation takes. This is part of the simplification for boolean filters, since the lookup will always be `exact` anyway and the value can only ever be `True` or `False`.
- The filter implementation is expected to always return the condition for the *positive* case, i.e. for the filter value `True`. In the examples above, the filter implementations return the correct filter for a `ApplicationVersion.objects.filter(is_first_stable_version=True)` filter. If the filter is called for the negative case (e.g. in a `ApplicationVersion.objects.filter(is_first_stable_version=False)` query), the boolean filter automatically takes care of negating the condition (essentially transforming it to `~Q(major=1, minor=0)` in the examples above), so that this doesn't have to be implemented manually.

5.2 Usage

With both implementations shown above, the queryable property can be used to filter querysets like any regular model field:

```
from django.db.models import Q

ApplicationVersion.objects.filter(version_str='1.1')
ApplicationVersion.objects.exclude(version_str__exact='1.2')
ApplicationVersion.objects.filter(application__name='My App', version_str='2.0')
ApplicationVersion.objects.filter(Q(version_str='1.9') | Q(major=2))
...
```

In the same manner, the filter can even be used when filtering on related models, e.g. when making queries from the `Application` model:

```
from django.db.models import Q

Application.objects.filter(versions__version_str='1.1')
Application.objects.exclude(versions__version_str__exact='1.2')
Application.objects.filter(name='My App', versions__version_str='2.0')
Application.objects.filter(Q(versions__major=2) | Q(versions__version_str='1.9'))
...
```

CHAPTER

SIX

ANNOTATABLE PROPERTIES

The most powerful feature of queryable properties can be unlocked if a property can be expressed as an annotation. Since annotations in a queryset behave like regular fields, they automatically offer some advantages:

- They can be used for queryset filtering without the need to explicitly implement filter behavior - though queryable properties still offer the option to implement custom filtering, even if a property is annotatable.
- They can be used for queryset ordering.
- They can be selected (which is what normally happens when using `QuerySet.annotate`), meaning their values are computed and returned by the database while still only executing a single query. This will lead to huge performance gains for properties whose getter would normally perform additional queries.

6.1 Implementation

Let's make the simple `version_str` property from previous examples annotatable. Using the decorator-based approach, the property's `annotater` method must be used.

```
from django.db.models import Model, Value
from django.db.models.functions import Concat
from queryable_properties.properties import queryable_property


class ApplicationVersion(Model):
    ...

    @queryable_property
    def version_str(self):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=self.major, minor=self.minor)

    @version_str.annotater
    @classmethod
    def version_str(cls):
        return Concat('major', Value('.'), 'minor')
```

Note: The `classmethod` decorator is not required, but makes the function look more natural since it takes the model class as its first argument.

For the same implementation with the class-based approach, the `get_annotation` method of the property class must be implemented instead. It is recommended to use the `AnnotationMixin` for such properties (more about this below), but it is not required to be used.

```
from django.db.models import Value
from django.db.models.functions import Concat
from queryable_properties.properties import AnnotationMixin, QueryableProperty


class VersionStringProperty(AnnotationMixin, QueryableProperty):

    def get_value(self, obj):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=obj.major, minor=obj.minor)

    def get_annotation(self, cls):
        return Concat('major', Value('.'), 'minor')
```

In both cases, the function/method takes the model class as the single argument (useful to implement custom logic in inheritance scenarios) and must return an annotation - anything that would normally be passed to a `QuerySet.annotate` call, like simple `F` objects, aggregates, `Case` expressions, `Subquery` expressions, etc.

Note: The returned annotation object may reference the names of other annotatable queryable properties on the same model, which will be resolved accordingly.

6.1.1 The `AnnotationMixin` and custom filter implementations

Unlike the `SetterMixin` and the `UpdateMixin`, the *queryable_properties.properties.AnnotationMixin* does a bit more than just define the stub for the `get_annotation` method:

- It automatically implements filtering via the `get_filter` method by simply creating `Q` objects that reference the annotation. It is therefore not necessary to implent filtering for an annotatable queryable property unless some additional custom logic is desired (applies to either approach).
- It sets the class attribute `filter_requires_annotation` of the property class to `True`. As the name suggests, this attribute determines if the annotation must be present in a queryset to be able to use the filter and is therefore automatically set to `True` to make the default filter implementation mentioned in the previous point work. For decorator-based properties using the `annotater` decorator, it also automatically sets `filter_requires_annotation` to `True` unless another value was already set (see the next example).

Caution: Since the `AnnotationMixin` simply implements the `get_filter` method as mentioned above, care must be taken when using other mixins (most notably the `LookupFilterMixin` - see *Lookup-based filter functions/methods*) that override this method as well (the implementations override each other).

This is also relevant for the decorator-based approach as these mixins are automatically added to such properties when they use annotations or lookup-based filters. The order of the mixins for the class-based approach or the used decorators for the decorator-based approach is therefore important in such cases (the mixin applied last wins).

If the filter implementation shown in the *One-for-all filter function/method* part of the filtering chapter (which does not require the annotation and should therefore be configured accordingly) was to be retained despite annotating being implemented, the implementation could look like this using the decorator-based approach (note the `requires_annotation=False`):

```
from django.db.models import Model, Q, Value
from django.db.models.functions import Concat
from queryable_properties.properties import queryable_property


class ApplicationVersion(Model):
    ...

    @queryable_property
    def version_str(self):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=self.major, minor=self.minor)

    @version_str.filter(requires_annotation=False)
    @classmethod
    def version_str(cls, lookup, value):
        if lookup != 'exact':  # Only allow equality checks for the simplicity of the␣
↪example
            raise NotImplementedError()
        # Don't implement any validation to keep the example simple.
        major, minor = value.split('.')
        return Q(major=major, minor=minor)

    @version_str.annotater
    @classmethod
    def version_str(cls):
        return Concat('major', Value('.'), 'minor')
```

Note: If lookup-based filters are used with the decorator-based approach, the `requires_annotation` value can be set on any method decorated with the `filter` decorator. If a value for this parameter is specified in multiple `filter` calls, the last one will be the one that will determine the final value since it's still a global flag for the filter behavior (regardless of lookup).

For the class-based approach, the class (or instance) attribute `filter_requires_annotation` must be changed instead:

```
from django.db.models import Q, Value
from django.db.models.functions import Concat
from queryable_properties.properties import AnnotationMixin, QueryableProperty


class VersionStringProperty(AnnotationMixin, QueryableProperty):

    filter_requires_annotation = False

    def get_value(self, obj):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=obj.major, minor=obj.minor)

    def get_filter(self, cls, lookup, value):
        if lookup != 'exact':  # Only allow equality checks for the simplicity of the␣
```

(continues on next page)

(continued from previous page)

```
↪example
            raise NotImplementedError()
        # Don't implement any validation to keep the example simple.
        major, minor = value.split('.')
        return Q(major=major, minor=minor)

    def get_annotation(self, cls):
        return Concat('major', Value('.'), 'minor')
```

Note: If a custom filter is implemented that does depend on the annotation (with `filter_requires_annotation=True`), the name of the property itself can be referenced in the returned `Q` objects. It will then refer to the annotation for that property instead of leading to an infinite recursion while trying to resolve the property filter.

Using the `LookupFilterMixin` described in *Lookup-based filter functions/methods*, it is also possible to only customize the filter logic for certain lookups while retaining the default filter of the `AnnotationMixin` for all remaining lookups. This is based on the `remaining_lookups_via_parent` feature of the `LookupFilterMixin` and requires the `LookupFilterMixin` to be higher up in the MRO than the `AnnotationMixin`. As an example, the `lt(e)` lookups could be implemented in a custom fashion for the `version_str` property.

For the decorator-based approach, this could look like the following example:

```
from django.db.models import Model, Q, Value
from django.db.models.functions import Concat
from queryable_properties.properties import queryable_property


class ApplicationVersion(Model):
    ...

    @queryable_property
    def version_str(self):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=self.major, minor=self.minor)

    @version_str.annotater
    @classmethod
    def version_str(cls):
        return Concat('major', Value('.'), 'minor')

    @version_str.filter(lookups=('lt', 'lte'), remaining_lookups_via_parent=True)
    @classmethod
    def version_str(cls, lookup, value):  # Only ever called with the 'lt' or 'lte' lookup.
        # Don't implement any validation to keep the example simple.
        major, minor = value.split('.')
        return Q(major__lt=major) | Q(**{'major': major, 'minor__{}'.format(lookup):␣
↪minor})
```

For the class-based approach, this could be achieved the following way:

```
from django.db.models import Q, Value
```

(continues on next page)

(continued from previous page)

```
from django.db.models.functions import Concat
from queryable_properties.properties import AnnotationMixin, LookupFilterMixin,␣
↪QueryableProperty


class VersionStringProperty(LookupFilterMixin, AnnotationMixin, QueryableProperty):

    remaining_lookups_via_parent = True

    def get_value(self, obj):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=obj.major, minor=obj.minor)

    @lookup_filter('lt', 'lte')  # Alternatively: @LookupFilterMixin.lookup_filter(...)
    def filter_lower(self, cls, lookup, value):  # Only ever called with the 'lt' or 'lte'␣
↪lookup.
        # Don't implement any validation to keep the example simple.
        major, minor = value.split('.')
        return Q(major__lt=major) | Q(**{'major': major, 'minor__{}'.format(lookup):␣
↪minor})

    def get_annotation(self, cls):
        return Concat('major', Value('.'), 'minor')
```

In both cases, filtering with the `lt(e)` lookups will call the custom implementation while filtering with any other lookup will fall back to the annotation-based filter implementation of the `AnnotationMixin` due to the `LookupFilterMixin` being higher up in the MRO and the `AnnotationMixin` therefore being considered its base class.

6.2 Automatic (non-selecting) annotation usage

Queryable properties that implement annotating can be used like regular model fields in various queryset operations without the need to explicitly add the annotation to a queryset. This is achieved by automatically adding a queryable property annotation to the queryset in a *non-selecting* way whenever such a property is referenced by name, meaning the annotation's SQL expression will not be part of the `SELECT` clause.

These queryset operations can also be used on related models and include:

- Filtering with an implementation that requires annotation (see above), e.g. `ApplicationVersion.objects.filter(version_str='2.0')` or `Application.objects.filter(versions__version_str='2.0)` for the first examples in this chapter.
- Ordering, e.g. `ApplicationVersion.objects.order_by('-version_str')` or `Application.objects.order_by('-versions__version_str')`.
- Using the queryable property in another annotation or aggregation, e.g. `ApplicationVersion.objects.annotate(same_value=F('version_str'))` or `Application.objects.annotate(related_value=F('versions__version_str'))`.

Caution: In Django versions below 1.8, it was not possible to order by annotations without selecting them at the same time. Queryable property annotations therefore have to be automatically added in a *selecting* manner if they appear in an `.order_by()` call in those versions.

If queryable properties are selected only to allow ordering (i.e. not also selected explicitly), their values will be discarded before returning the results in regular querysets as well as `.values()`/`.values_list()` querysets. This is done because selected queryable properties behave differently (see below), and this behavior is meant to be consistent across all supported Django versions.

However, keep in mind that the additional selection may have performance implications and may also affect `DISTINCT` clauses, `GROUP BY` clauses, aggregates, etc. due to the additional columns that are queried.

Django versions starting from 1.8 do not have this problem as ordering by annotations is possible without selection.

6.2.1 Caution: the order of queryset operations still matters!

When making use of the automatic annotation injection, keep in mind that this is only a convenience feature that simply performs two operations: it adds the queryable property annotation to the queryset (similarly to manually calling `.annotate()`) and then performs the operation that was actually called (filtering, ordering, etc.). Therefore, the order of operations performed on querysets still matters when additionally dealing with other fields or even other queryable properties. A classic example for this is the order of `annotate()` and `filter()` clauses when dealing with aggregates.

This is even more important for operations performed on related objects as it may influence how `JOIN` ed tables are reused (which is standard Django behavior and not a "problem" of queryable properties). To provide an example for this, let's assume the `version_str` queryable property from the first examples in this chapter in conjunction with the following query:

```
Application.objects.filter(versions__version_str='2.0', versions__major=2)
```

While the filter conditions themselves don't make much sense together, they both use the same relation to the version objects and can therefore show the potential problem. Depending on which of the conditions is processed first, the results will be different:

- If the `major` filter is applied first, the actions will be performed in this order: 1. apply the `major` filter 2. automatically add the `version_str` annotation 3. apply the `version_str` filter

  This will lead to only joining the `ApplicationVersion` table once and therefore correctly resulting in the filter combined with `AND` that was most likely intended.

- If the `version_str` filter is applied first, the actions will be performed in this order: 1. automatically add the `version_str` annotation 2. apply the `version_str` filter 3. apply the `major` filter

  This will lead to two independent `JOIN``s of the ``ApplicationVersion` table, where each condition will only be applied to one of the joined tables, leading to more duplicate results and essentially an `OR` conjunction of the filter conditions.

It may therefore be desirable to ensure that the conditions are applied in the correct order. To make sure that the `major` condition will be applied first, multiple options are at hand:

```
from django.db.models import Q

# Using separate filter calls
Application.objects.filter(versions__major=2).filter(versions__version_str='2.0')
# Combining Q objects to represent the AND conjunction
Application.objects.filter(Q(versions__major=2) & Q(versions__version_str='2.0'))
# Passing the keyword arguments in the correct order in Python versions that preserve␣
↪their order (3.7 and above)
Application.objects.filter(versions__major=2, versions__version_str='2.0')
```

6.3 Selecting annotations

Whenever the actual values for queryable properties are to be retrieved while performing a query, they must be explicitly selected using the `select_properties` method defined by the `QueryablePropertiesManager` and the `QueryablePropertiesQuerySet(Mixin)`, which takes any number of queryable property names as its arguments. When this method is used, the specified queryable property annotations will be added to the queryset in a *selecting* manner, meaning the SQL representing an annotation will be part of the `SELECT` clause of the query. For consistency, the `select_properties` method always has to be used to select a queryable property annotation - even when using features like `values` or `values_list` (these methods will not automatically select queryable properties).

The following example shows how to select the `version_str` property from the examples above:

```
for version in ApplicationVersion.objects.select_properties('version_str'):
    print(version.version_str)  # Uses the value directly from the query and does not␣
↪call the getter
```

To be able to make use of this performance-oriented feature, **all explicitly selected queryable properties will always behave like properties with a** *Cached getter* on the model instances returned by the queryset. If this wasn't the case, accessing uncached queryable properties on model instances would always execute their default behavior: calling the getter. This would make the selection of the annotations useless to begin with, as the getter would called regardless and no performance gain could be achieved by the queryset operation. By instead behaving like cached queryable properties, one can make use of the queried values, which will be cached for any number of consecutive accesses of the property on model objects returned by the queryset. If it is desired to not access the cached values anymore, the cached value can always be cleared as described in *Resetting a cached property*.

6.3.1 Queryable properties on related models

Selecting the values of queryable property annotations is the one annotation-based feature that **does not** allow to use queryable properties defined on related models. Therefore, the following example (based on the `version_str` property from the examples above) will **not** work:

```
for app in Application.objects.select_properties('versions__version_str'):
    ...
```

This is intentional for the following reasons:

- Since the queryable property would be defined on another model, the actual annotation in the current queryset would have to use a different name. The only real option for this would be the whole relation path containing the `__` separator(s), e.g. `versions__version_str` in the example above, which would be quite weird and ugly.
- Depending on the type of the relation, getting queryable property values from related models would not always have a clear meaning. This is the case for all `...`-to-many relations, where there would be multiple potential values to choose from.

There is, however, a way to get the annotation values from queryable properties of related models: Since manually added annotations can refer to queryable property annotations even across relations, this can be used to actually select the values. In the simplest case, the property could simply be aliased using an `F` object:

```
from django.db.models import F

for app in Application.objects.annotate(my_annotation=F('versions__version_str')):
    print(app.my_annotation)
```

This solves the problems mentioned above:

- You need to choose a name for the new annotation yourself (`my_annotation` in the example), which eliminates potential weird and ugly annotation names.
- You will have to make sure that the related values in conjunction with the relation type make sense and yield the results you expect.

6.3.2 Querying properties for already loaded model instances

Queryable property values may also be queried for model instances that were previously queried from the database. The utility function *queryable_properties.utils.prefetch_queryable_properties()* can be used for this purpose, which is akin to Django's `prefetch_related_objects` function, which serves a similar purpose for related objects. This function can be used to load the values of one or multiple annotatable queryable properties for a sequence of model instances at once, which is especially useful to improve performance for queryable properties whose getter would otherwise execute a query.

queryable_properties.utils.prefetch_queryable_properties() takes the sequence of model instances as well as any number of query paths to the queryable properties to load the values for. For the `version_str` property from the examples above, this could be achieved like this:

```
from queryable_properties.utils import prefetch_queryable_properties

versions = load_versions()  # A sequence of ApplicationVersion instances
prefetch_queryable_properties(versions, 'version_str')
```

Notes:

- Due to the explicit selections, the selected properties always behave like cached properties as is the case for `select_properties`.
- Unlike the `select_properties` queryset method described above, the query paths supplied to `prefetch_queryable_properties` may contain the lookup separator (`__`) to reference queryable properties on related objects (even via many-to-many relations) and populate the queryable property cache on these objects. This works because the function figures out the property and its corresponding model on its own by accessing the relations on the individual objects and performing the query for the property the model is defined on. Since the related objects are accessed, make sure that they were already loaded beforehand (e.g. via Django's `prefetch_related_objects` function) to avoid additional queries.
- The sequence of model instances may contain objects of different, unrelated models as long as all given query paths are valid for all instances. The function will figure out which models it needs to perform queries for.
- As a consequence of the previous notes, queryable property values may need to be queried for multiple different models. However, `prefetch_queryable_properties` will only ever perform one query per affected model.
- `prefetch_queryable_properties` can even be used when the referenced properties already have cached values on the given model instances. This refreshes the cached values with the current values from the database.

6.4 Regarding aggregate annotations across relations

An annotatable queryable property that is implemented using an aggregate may return unexpected results when using it from a related model in a queryset (regardless for explicit selection or automatic use) since no extended `GROUP BY` setup other than what Django would do on its own takes place.

Consider the following decorator-based example (the effect would be the same for a class-based property), where a queryable property for the number of corresponding versions is added to the `Application` model:

```
from django.db.models import Count, Model
from queryable_properties.properties import queryable_property


class Application(Model):
    ...

    @queryable_property
    def version_count(self):
        return self.versions.count()

    @version_count.annotater
    @classmethod
    def version_count(cls):
        return Count('versions')
```

If there were 2 applications, one having 2 versions and the other having 3, the following queryset would return both of these versions, since the annotation values would be 2 and 3, respectively:

```
Application.objects.filter(version_count__in=(2, 3))  # Finds both applications
```

If both of these applications would belong to the same category, one would probably expect that we following queryset would find that category, since it has 2 applications that fit the filter conditions:

```
Category.objects.filter(applications__version_count__in=(2, 3))
```

However, this is **not** the case - this query will not return that category. This is because the result of the annotation is basically the same as the following manual annotation:

```
from django.db.models import Count

Category.objects.annotate(applications__version_count=Count('applications__versions'))
```

This means that the value `applications__version_count` for the category would be 5, since it simply counts all versions that are associated with this category via an application at all. The reason for this is that Django uses `JOIN` s and `GROUP BY` clauses in order to generate the aggregated values, but they are not automatically grouped by application. Instead, the `GROUP BY` clause only contains the columns of the `Category` model, leading to one total value per category.

There are options to work around this when running into this problem:

- Use `values()` to set the `GROUP BY` clause yourself. For the example above, a `.values('pk', 'applications__pk')` call before the `.filter()` call would be sufficient. Keep in mind that the same category can then be returned multiple times if more than one of its versions matches the filter condition.
- Do not directly use an aggregate like `Count` at all and count the versions per application using a subquery. This subquery will then also be performed correctly when the queryable property is used from a related model.

CHAPTER

SEVEN

ANNOTATION-BASED PROPERTIES

There are various scenarios where even the getter of a (queryable) property must perform a database query to provide its value, e.g. when the property:

- is based on an aggregate,
- checks for the existence of related/other objects in the database,
- loads a field value from anywhere else in the database via a custom subquery,
- etc.

Since most, if not all, of these cases can be expressed using queryset annotations, this allows the use of *Annotatable properties* to implement a corresponding queryable property. If the getter of a property would require to perform a query anyways, one could simply reuse the annotation to implement the getter to achieve both features in a DRY manner. *django-queryable-properties* therefore offers a dedicated option that allows to implement annotation-based properties that use the annotater implementation to provide the getter value - this allows to implement a queryable property that has a functional getter and allows filtering and the use all annotation-based queryset features while only implementing the annotation.

Note: One should only use annotation-based properties whenever the getter would need to perform a query anyways. Whenever the getter could be implemented without performing extra queries, it should be implemented manually as the query-less implementation is likely more performant.

7.1 Implementation

To provide a realistic example, let's implement a property that provides the number of versions that is defined for an application, similar to the example in *Regarding aggregate annotations across relations*.

The decorator-based approach for an annotation-based property looks slightly different since the *queryable_property* decorator is normally used for the getter, but the goal of annotation-based properties is to avoid having to manually implement a getter. The `queryable_property` decorator therefore accepts an `annotation_based` argument for this use case - if it is set to `True`, the decorator expects the annotation function (that is usually decorated with `@<property_name>.annotater` - see *Implementation*) as the decorated function instead of the getter function.

```
from django.db.models import Count, Model, Value
from queryable_properties.properties import queryable_property


class ApplicationVersion(Model):
    ...
```

(continues on next page)

(continued from previous page)

```
    @queryable_property(annotation_based=True)
    @classmethod
    def version_count(cls):
        """Return the number of versions that exist for this application."""
        return Count('versions')
```

Note: The `classmethod` decorator is not required, but makes the function look more natural since it takes the model class as its first argument.

The class-based approach looks a lot like a regular annotatable property - it simply uses the `AnnotationGetterMixin` instead of the `AnnotationMixin`, which already implements `get_value` to be based on the annotation.

```
from django.db.models import Count, Value
from queryable_properties.properties import AnnotationGetterMixin, QueryableProperty


class VersionCountProperty(AnnotationGetterMixin, QueryableProperty):

    def get_annotation(self, cls):
        return Count('versions')
```

7.1.1 About the `AnnotationGetterMixin`

The *queryable_properties.properties.AnnotationGetterMixin* is the core part of the option to implement annotation-based properties. It is used explicitly in the class-based approach, but also automatically added to properties defined using the decorator-based approach whenever the `annotation_based` argument is set to `True`. This mixin is based on the `AnnotationMixin`, which means that all notes described in *The AnnotationMixin and custom filter implementations* apply here as well.

The main addition provided by the `AnnotationGetterMixin` is the provided implementation of the `get_value` method to implement the getter. This getter builds a `DISTINCT` queryset using the base manager (`_base_manager`) of the object the property is accessed on, filters it to only that object via its primary key, adds the annotation and retrieves only the annotation value via `values_list` and `get`. The getter may therefore raise `MultipleObjectsReturned` exceptions if somehow more than one row is returned or `DoesNotExist` exceptions if no row can be found (e.g. when accessing the property on an object that is not yet saved to the database).

Due to the performed queries, the getters of annotation-based properties can be a prime use case for a *Cached getter*. Because of this, the `AnnotationGetterMixin` also adds the `cached` argument to the initializer (`__init__`) of the classes that use it (which is only relevant for the class-based approach). This means that objects of the property class can be individually flagged as cached properties. The `VersionCountProperty` example above could therefore be used in the following ways:

```
class Application(Model):
    ...

    version_count = VersionCountProperty()
    # ... or ...
    version_count = VersionCountProperty(cached=False)
    # ... or ...
    version_count = VersionCountProperty(cached=True)
```

The default value for this `cached` argument is `None`, which is interpreted as "use the default value". This allows to retain the ability to set the `cached` flag as a class attribute as well, which then provides this default value.

CHAPTER

EIGHT

UPDATE QUERIES

Queryable properties offer the option to use the names of properties in batch updates (i.e. when using the `update` method of querysets). To achieve this, the `update` value for a queryable property will simply be translated into `update` values for actual model fields.

8.1 Implementation

Let's use the `version_str` of the `ApplicationVersion` model as an example once again.

To allow the usage of this queryable property in queryset updates using the decorator-based approach, the property's `updater` method must be used.

```
from queryable_properties.properties import queryable_property


class ApplicationVersion(models.Model):
    ...

    @queryable_property
    def version_str(self):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=self.major, minor=self.minor)

    @version_str.updater
    @classmethod
    def version_str(cls, value):
        # Don't implement any validation to keep the example simple.
        major, minor = value.split('.')
        return {'major': major, 'minor': minor}
```

Note: The `classmethod` decorator is not required, but makes the function look more natural since it takes the model class as its first argument.

Using the class-based approach, the same thing can be achieved by implementing the `get_update_kwargs` method of the property class. It is recommended to use the *queryable_properties.properties.UpdateMixin* for class-based queryable properties that are supposed to be used in queryset updates because it defines the actual stub for the `get_update_kwargs` method. However, using this mixin is not required - a queryable property can be used for queryset updates as long as the `get_update_kwargs` method is implemented correctly.

```
from queryable_properties.properties import QueryableProperty, UpdateMixin


class VersionStringProperty(UpdateMixin, QueryableProperty):

    def get_value(self, obj):
        """Return the combined version info as a string."""
        return '{major}.{minor}'.format(major=obj.major, minor=obj.minor)

    def get_update_kwargs(self, cls, value):
        # Don't implement any validation to keep the example simple.
        major, minor = value.split('.')
        return {'major': major, 'minor': minor}
```

In both cases, the function/method to implement takes 2 arguments:

cls
: The model class. Mainly useful to implement custom logic in inheritance scenarios.

value
: The value to update the database rows with.

Using either approach, the function/method is expected to return a `dict` object that contains the model field/value combinations that are actually required to perform the update correctly.

Note: The returned `dict` object may contain name/value pairs referring to other queryable properties on the same model, which will be resolved accordingly in the same manner.

8.2 Usage

With both implementations, the queryable property can be used in queryset updates like this:

```
ApplicationVersion.objects.update(version_str='1.1')
```

The specified value is then translated into actual field values by the implemented function/method and the real, underlying `update` call will take place with these values.

8.3 Limitations

8.3.1 Related models

Unlike filtering and annotation-based operations, updating can not be used for fields on related models. This is because updates are generally meant for records of the same type to be able to perform an `UPDATE` query on a single table (aside from inheritance scenarios, where Django takes care of updating multiple tables correctly). *django-queryable-properties* doesn't add any additional logic here and simply translates the given value according to the updater implementation and therefore doesn't allow updating fields on related models, either.

8.3.2 Expression-based update values

Using expression-based values (like an `F` objects or a conditional update) are generally not supported when updating via a queryable property. This is because the queryable property updater is simply a preprocessor for the `.update(...)` keyword arguments on the python side, while expression-based updates rely on other values in the query, which are only evaluated in SQL when the query actually runs.

However, *django-queryable-properties* doesn't technically prevent to use expressions as update values. This means that if an expression is used as an update value, it will be passed through to the method decorated with `updater` (decorator-based approach) or the `get_update_kwargs` implementation (class-based approach). Therefore it would technically be possible to process an expression in the updater's implementation as long the expression can be preprocessed in a sensible way before the query runs.

CHAPTER

NINE

COMMON PATTERNS

django-queryable-properties offers some fully implemented properties for common code patterns out of the box. All of them are class-based and parametrizable for their specific use case (while still supporting all *Common property arguments*) and are supposed to help remove boilerplate for recurring types of properties while making them usable in querysets at the same time.

9.1 Specialized properties

The properties in this category are designed for very specific use cases and are not based on annotations.

9.1.1 `ValueCheckProperty`: Checking a field for one or multiple specific values

Properties on model objects are often used to check if an attribute on a model instance contains a specific value (or one of multiple values). This is often done for fields with choices as it allows to implement the check for a certain choice value in one place instead of redefining it whenever the field should be checked for the value. However, the pattern is not limited to fields with choices.

Imagine that the `ApplicationVersion` example model would also contain a field that contains information about the type of release, e.g. if a certain version is an alpha, a beta, etc. It would be well-advised to use a field with choices for this value and to also define properties to check for the individual values to only define these checks once.

Without *django-queryable-properties*, the implementation could look similar to this:

```
from django.db import models
from django.utils.translation import ugettext_lazy as _


class ApplicationVersion(models.Model):
    ALPHA = 'a'
    BETA = 'b'
    STABLE = 's'
    RELEASE_TYPE_CHOICES = (
        (ALPHA, _('Alpha')),
        (BETA, _('Beta')),
        (STABLE, _('Stable')),
    )

    ...  # other fields
    release_type = models.CharField(max_length=1, choices=RELEASE_TYPE_CHOICES)
```

(continues on next page)

(continued from previous page)

```
    @property
    def is_alpha(self):
        return self.release_type == self.ALPHA

    @property
    def is_beta(self):
        return self.release_type == self.BETA

    @property
    def is_stable(self):
        return self.release_type == self.STABLE

    @property
    def is_unstable(self):
        return self.release_type in (self.ALPHA, self.BETA)
```

Instead of defining the properties like this, the property class *queryable_properties.properties.ValueCheckProperty* could be used:

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

from queryable_properties.managers import QueryablePropertiesManager
from queryable_properties.properties import ValueCheckProperty


class ApplicationVersion(models.Model):
    ALPHA = 'a'
    BETA = 'b'
    STABLE = 's'
    RELEASE_TYPE_CHOICES = (
        (ALPHA, _('Alpha')),
        (BETA, _('Beta')),
        (STABLE, _('Stable')),
    )

    ...  # other fields
    release_type = models.CharField(max_length=1, choices=RELEASE_TYPE_CHOICES)

    objects = QueryablePropertiesManager()

    is_alpha = ValueCheckProperty('release_type', ALPHA)
    is_beta = ValueCheckProperty('release_type', BETA)
    is_stable = ValueCheckProperty('release_type', STABLE)
    is_unstable = ValueCheckProperty('release_type', ALPHA, BETA)
```

Instances of this property class take the path of the attribute to check as their first parameter in addition to any number of parameters that represent the values to check for - if one of them matches when the property is accessed on a model instance, the property will return `True` (otherwise `False`).

Not only does this property class allow to achieve the same functionality with less code, but it offers even more functionality due to being a *queryable* property. The class implements both queryset filtering as well as annotating (based on Django's `Case`/`When` objects), so the properties can be used in querysets as well:

```
stable_versions = ApplicationVersion.objects.filter(is_stable=True)
non_alpha_versions = ApplicationVersion.objects.filter(is_alpha=False)
ApplicationVersion.objects.order_by('is_unstable')
```

For a quick overview, the `ValueCheckProperty` offers the following queryable property features:

Feature	Supported
Getter	Yes
Setter	No
Filtering	Yes
Annotation	Yes (Django 1.8 or higher)
Updating	No

Attribute paths

The attribute path specified as the first parameter can not only be a simple field name like in the example above, but also a more complex path to an attribute using dot-notation - basically the same way as for Python's `operator.attrgetter`. For queryset operations, the dots are then simply replaced by the lookup separator (`__`), so an attribute path `my.attr` becomes `my__attr` in queries.

This is especially useful to reach fields of related model instances via foreign keys, but it also allows to be more creative since the path simply needs to make sense both on the object-level as well as in queries. For example, a `DateField` may be defined as `date_field = models.DateField()`, which would allow a `ValueCheckProperty` to be set up with the path `date_field.year`. This works because the `date` object has an attribute `year` on the object-level and Django offers a `year` transform for querysets (so `date_field__year` does in fact work). However, this specific example requires at least Django 1.9 as older versions don't allow to combine transforms and lookups. In general, this means that the attribute path does not have to refer to an actual field, which also means that it may refer to another queryable property (which needs to support the `in` lookup to be able to filter correctly).

Unlike Python's `operator.attrgetter`, the property will also automatically catch some exceptions during getter access (if any of them occur, the property considers none of the configured values as matching):

- `AttributeError` s if an intermediate object is `None` (e.g. if a path is `a.b` and the `a` attribute already returns `None`, then the attribute error when accessing `b` will be caught). This is intended to make working with nullable fields easier. Any other kind of `AttributeError` will still be raised.
- Any `ObjectDoesNotExist` errors raised by Django, which are raised e.g. when accessing a reverse One-To-One relation with a missing value. This is intended to make working with these kinds of relations easier.

9.1.2 `RangeCheckProperty`: Checking if a value is contained in a range defined by two fields

A common pattern that uses a property is having a model with two attributes that define a lower and an upper limit and a property that checks if a certain value is contained in that range. These fields may be numerical fields (`IntegerField`, `DecimalField`, etc.) or something like date fields (`DateField`, `DateTimeField`, etc.) - basically anything that allows "greater than" and "lower than" comparisons.

As an example, the `ApplicationVersion` example model could contain two such date fields to express the period in which a certain app version is supported, which could look similar to this:

```
from django.db import models
from django.utils import timezone
```

(continues on next page)

(continued from previous page)

```
class ApplicationVersion(models.Model):
    ...  # other fields
    supported_from = models.DateTimeField()
    supported_until = models.DateTimeField()

    @property
    def is_supported(self):
        return self.supported_from <= timezone.now() <= self.supported_until
```

Instead of defining the properties like this, the property class *queryable_properties.properties.RangeCheckProperty* could be used:

```
from django.db import models
from django.utils import timezone

from queryable_properties.managers import QueryablePropertiesManager
from queryable_properties.properties import RangeCheckProperty


class ApplicationVersion(models.Model):
    ...  # other fields
    supported_from = models.DateTimeField()
    supported_until = models.DateTimeField()

    objects = QueryablePropertiesManager()

    is_supported = RangeCheckProperty('supported_from', 'supported_until', timezone.now)
```

Instances of this property class take the paths of the attributes for the lower and upper limits as their first and second arguments. Both values may also be more complex attribute paths in dot-notation - the same behavior as for the attribute path of `ValueCheckProperty` objects apply (refer to chapter *Attribute paths* above). If one of the limiting values is `None` or an exception is caught, the value is considered missing (see next sub- chapter). The third mandatory parameter for `RangeCheckProperty` objects is the value to check against, which may either be a static value or a callable that can be called without any argument and that returns the actual value (`timezone.now` in the example above), similar to the `default` option of Django's model fields.

Not only does this property class allow to achieve the same functionality with less code, but it offers even more functionality due to being a *queryable* property. The class implements both queryset filtering as well as annotating (based on Django's `Case`/`When` objects), so the properties can be used in querysets as well:

```
currently_supported = ApplicationVersion.objects.filter(is_supported=True)
not_supported = ApplicationVersion.objects.filter(is_supported=False)
ApplicationVersion.objects.order_by('is_supported')
```

For a quick overview, the `RangeCheckProperty` offers the following queryable property features:

Feature	Supported
Getter	Yes
Setter	No
Filtering	Yes
Annotation	Yes (Django 1.8 or higher)
Updating	No

Range configuration

`RangeCheckProperty` objects also allow further configuration to tweak the configured range via some optional parameters:

`include_boundaries`
: Determines if a value exactly equal to one of the limits is considered a part of the range (default: `True`).

`include_missing`
: Determines if a missing value for either boundary is considered part of the range (default: `False`).

`in_range`
: Determines if the property should return `True` if the value is contained in the configured range (this is the default) or if it should return `True` if the value is outside of the range.

It should be noted that the `include_boundaries` and `include_missing` parameters are applied first to define the range (which values are considered inside the range between the two values) and the `in_range` parameter is applied *afterwards* to potentially invert the result (in the case of `in_range=False`). This means that setting `include_missing=True` defines that missing values are part of the range and a value of `in_range=False` would then invert this range, meaning that missing values would **not** lead to a value of `True` since they are configured to be in the range while the property is set up to return `True` for values outside of the range. For a quick reference, all possible configuration combinations are listed in the following table:

include_boun	include_miss	in_range	returns True for
True	False	True	• Values in between boundaries (excl.) • The exact boundary values
True	True	True	• Values in between boundaries (excl.) • The exact boundary values • Missing values
False	False	True	• Values in between boundaries (excl.)
False	True	True	• Values in between boundaries (excl.) • Missing values
True	False	False	• Values outside of the boundaries (excl.) • Missing values
True	True	False	• Values outside of the boundaries (excl.)
False	False	False	• Values outside of the boundaries (excl.) • The exact boundary values • Missing values
False	True	False	• Values outside of the boundaries (excl.) • The exact boundary values

Note: The attribute paths passed to `RangeCheckProperty` may also refer to other queryable properties as long as these properties allow filtering with the `lt`/`lte` and `gt`/`gte` lookups (depending on the value of `include_boundaries`) and potentially the `isnull` lookup (depending on the value of `include_missing`).

9.1.3 `MappingProperty`: Mapping field values to other values

The property class *queryable_properties.properties.MappingProperty* streamlines a very simple pattern: mapping the values of an attribute (most likely a field) to different values. While there is nothing special about this on an object basis, it allows to introduce values into querysets that otherwise are not database values. The value mapping inside querysets is achieved using `CASE`/`WHEN` expressions based on Django's `Case`/`When` objects, which means that this property class can only be properly used in Django versions that provide these features (1.8+).

A common use case for this might be to set up a `MappingProperty` that simply works with a choice field and uses the choice definitions themselves as its mappings. This allows to introduce the (most likely translatable) choice verbose names into the query, which in turn allows to order the queryset by the *translated* verbose names, providing sensible ordering no matter what language an application is used in.

For the release type values in an example above, this could look like this:

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

from queryable_properties.managers import QueryablePropertiesManager
from queryable_properties.properties import MappingProperty


class ApplicationVersion(models.Model):
    ALPHA = 'a'
    BETA = 'b'
    STABLE = 's'
    RELEASE_TYPE_CHOICES = (
        (ALPHA, _('Alpha')),
        (BETA, _('Beta')),
        (STABLE, _('Stable')),
    )

    ...  # other fields
    release_type = models.CharField(max_length=1, choices=RELEASE_TYPE_CHOICES)

    objects = QueryablePropertiesManager()

    release_type_verbose_name = MappingProperty('release_type', models.CharField(),␣
↪RELEASE_TYPE_CHOICES)
```

In a view, one could then perform a query similar to the following to order the `ApplicationVersion` objects by their translated verbose name, which may lead to a different ordering depending on the user's language:

```
ApplicationVersion.objects.order_by('release_type_verbose_name')
```

This is, however, not the only way `MappingProperty` objects can be used - any attribute values may be translated into any other values that can be represented in database queries and then used in querysets.

`MappingProperty` objects may be initialized with up to four parameters:

`attribute_path` (required)
: An attribute path to the attribute whose values are to be mapped to other values - the same behavior as for the attribute path of `ValueCheckProperty` objects apply (refer to chapter *Attribute paths* above).

`output_field` (required)
: A field instance that is used to represent the translated values in queries.

`mappings` (required)
: Defines the actual mappings as an iterable of 2-tuples, where the first value is the expected attribute value and the second value is the translated value. This can be almost any type of iterable - it just needs to be able to be iterated multiple times as the whole iterable is evaluated any time the property is accessed on an object or in queries (generators are therefore not usable).

`default` (optional)
: Defines a default value, which defaults to `None`. Whenever an attribute value is encountered that has no mapping via the third parameter, this default value is returned instead.

Note: Whenever the mapping output values are actually accessed (by accessing the property on an object or by referencing it in a queryset), lazy values (like the translations in the example above) are evaluated. Property access or queryset references should therefore be performed as late as possible when dealing with lazy mapping values. For

queryset operations, the translated values are also automatically wrapped in Value objects.

Note: The attribute path passed to `MappingProperty` may also refer to another queryable property as long as this property allows filtering with the `exact` lookup.

For a quick overview, the `MappingProperty` offers the following queryable property features:

Feature	Supported
Getter	Yes
Setter	No
Filtering	Yes (Django 1.8 or higher)
Annotation	Yes (Django 1.8 or higher)
Updating	No

9.2 Annotation-based properties

The properties in this category are all *Annotation-based properties*, which means their getter implementation will also perform a database query. All of the listed properties therefore also take an additional `cached` argument in their initializer that allows to mark individual properties as having a *Cached getter*. This can improve performance since the query will only be executed on the first getter access at the cost of potentially not working with an up-to-date value.

9.2.1 `AnnotationProperty`: Static annotations

The property class *queryable_properties.properties.AnnotationProperty* represents the most simple common annotation-based property. It can be instanciated using any annotation and will use that annotation both in queries as well as to provide its getter value. This, however, means that the `AnnotationProperty` is only intended to be used with static/fixed annotations without any dynamic components as its objects are set up by passing the annotation to the initializer.

As an example, the `version_str` property from the annotation *Implementation* section could be reduced to (**not recommended**):

```
from django.db.models import Model, Value
from django.db.models.functions import Concat
from queryable_properties.properties import AnnotationProperty


class ApplicationVersion(Model):
    ...  # other fields/properties

    version_str = AnnotationProperty(Concat('major', Value('.'), 'minor'))
```

Note: This example is only supposed to demonstrate how to set up an `AnnotationProperty`. Implementing a `Concat` annotation like this is not recommended as even the getter will perform a query, even though concatenating field values on the object level could simply be done without involving the database.

For a quick overview, the `AnnotationProperty` offers the following queryable property features:

Feature	Supported
Getter	Yes
Setter	No
Filtering	Yes
Annotation	Yes
Updating	No

9.2.2 `AggregateProperty`: Simple aggregates

django-queryable-properties also comes with a property class for simple aggregates that simply takes an aggregate object and uses it for both queryset annotations as well as the getter. This is therefore not entirely different from the `AnnotationProperty` class shown above. The main difference between the two is that while `AnnotationProperty` uses `QuerySet.annotate` to query the getter value, `AggregateProperty` uses `QuerySet.aggregate`, which is slightly more efficient. Using `AggregateProperty` for aggregate annotations might also make code more clear/readable.

As an example, the `Application` model could receive a simple property that returns the number of versions like the one in the *Implementation* section of annotation-based properties. *`queryable_properties.properties.AggregateProperty`* allows to implement this in an even more condensed form:

```
from django.db.models import Count, Model
from queryable_properties.properties import AggregateProperty


class Application(Model):
    ...  # other fields/properties

    version_count = AggregateProperty(Count('versions'))
```

Note: Since this property deals with aggregates, the notes *Regarding aggregate annotations across relations* apply when using such properties across relations in querysets.

For a quick overview, the `AggregateProperty` offers the following queryable property features:

Feature	Supported
Getter	Yes
Setter	No
Filtering	Yes
Annotation	Yes
Updating	No

9.2.3 `RelatedExistenceCheckProperty`: Checking whether or not certain related objects exist

A common use case for properties is checking whether or not at least one related object exists. For example, both the `Application` as well the `Category` models could define a property that checks whether or not any corresponding applications versions exist in the database.

Without *django-queryable-properties*, the implementation could look similar to this:

```
from django.db import models


class Category(models.Model):
    ...  # other fields/properties

    @property
    def has_versions(self):
        return self.applications.filter(versions__isnull=False).exists()


class Application(models.Model):
    ...  # other fields/properties

    @property
    def has_versions(self):
        return self.versions.exists()
```

Instead of defining the properties like this, the property class *queryable_properties.properties.RelatedExistenceCheckProperty* could be used:

```
from django.db import models
from queryable_properties.properties import RelatedExistenceCheckProperty


class Category(models.Model):
    ...  # other fields/properties

    has_versions = RelatedExistenceCheckProperty('applications__versions')


class Application(models.Model):
    ...  # other fields/properties

    has_versions = RelatedExistenceCheckProperty('versions')
```

Instances of this property class take the query path to the related objects, which may also span multiple relations using the `__` separator, as their first parameter. Additionally, the optional `negated` parameter can be used to set up the property to check for the *non-existence* of related objects instead. In queries, the given query path is extended with the `__isnull` lookup, to determine whether or not related objects exist. The path may also lead to a nullable field, which would allow to check for the existence of related objects that have a value for a certain field.

Not only does this property class allow to achieve the same functionality with less code, but it offers even more functionality due to being a *queryable* property. The class implements both queryset filtering as well as annotating (based on Django's `Case`/`When` objects), so the properties can be used in querysets as well:

```
apps_with_versions = Application.objects.filter(has_versions=True)
apps_without_versions = Application.objects.filter(has_versions=False)
Category.objects.order_by('has_versions')
```

When being used in querysets like this, the filter condition is tested in a `__in` subquery (supported in all Django versions supported by *django-queryable-properties*), which is built using the base manager (`_base_manager`) of the property's associated model class. This avoids `JOIN` ing the related models in the main queryset and therefore avoids duplicate objects in the results whenever ...-to-many relations are involved.

Note: The query paths passed to `RelatedExistenceCheckProperty` may also refer to another queryable property as long as this property allows filtering with the `isnull` lookup.

Note: Since the property's getter also performs a query for the existence check, the use of the `RelatedExistenceCheckProperty` is only recommended whenever a query would have to be performed anyway. It is therefore not recommended to be used to check if local non-relation fields are filled or even if a simple forward `ForeignKey` or `OneToOneField` has a value (which could be tested by checking the `<fk_name>_id` attribute without performing a query). A `ValueCheckProperty` may be better suited to check the value of local fields instead.

For a quick overview, the `RelatedExistenceCheckProperty` offers the following queryable property features:

Feature	Supported
Getter	Yes
Setter	No
Filtering	Yes
Annotation	Yes (Django 1.8 or higher)
Updating	No

9.3 Subquery-based properties (Django 1.11 or higher)

The properties in this category are all based on custom subqueries, i.e. they utilize Django's `Subquery` objects. They are therefore *Annotation-based properties*, which means their getter implementation will also perform a database query. Due to the utilization of `Subquery` objects, these properties can only be used in conjunction with a Django version that supports custom subqueries, i.e. Django 1.11 or higher.

All subquery-based properties take a queryset that will be used to generate the custom subquery as their first argument. This queryset is always expected to be a regular queryset, i.e. **not** a `Subquery` object - the properties will build these objects on their own using the given queryset. The specified queryset can (and in most cases should) contain `OuterRef` objects to filter the subquery's rows based on the outer query. These `OuterRef` objects will always be based on the model the property is defined on - all fields of that model or related fields starting from that model can therefore be referenced.

Instead of specifying a queryset directly, the subquery-based properties can also take a callable without any arguments as their first parameter, which in turn must return the queryset. This is useful in cases where the model class for the subquery's queryset cannot be imported on the module level or is defined later in the same module.

9.3.1 SubqueryFieldProperty: Getting a field value from a subquery

The property class *queryable_properties.properties.SubqueryFieldProperty* allows to retrieve the value of any field from the specified subquery. The field does not have to be a static model field, but may also be an annotated field (which can even be used to work around the problem described in *Regarding aggregate annotations across relations*) or even a queryable property as long as it was selected as described in *Selecting annotations*.

Based on the `version_str` property for the `ApplicationVersion` shown in the *Implementation* documentation for annotatable properties, an example property could be implemented for the `Application` model that determines the highest version for each application via a subquery:

```
from django.db import models
from queryable_properties.properties import SubqueryFieldProperty


class Application(models.Model):
    ...  # other fields/properties

    highest_version = SubqueryFieldProperty(
        (ApplicationVersion.objects.select_properties('version_str')
                                   .filter(application=models.OuterRef('pk'))
                                   .order_by('-major', '-minor')),
        field_name='version_str',  # The field to extract the property value from
        output_field=models.CharField()  # Only required in cases where Django can't␣
↪determine the type on its own
    )
```

Note: Since the property can only return a single value per object, the subquery is limited to the first row (the specified queryset and field name is essentially transformed into `Subquery(queryset.values(field_name)[:1])`). If a subquery returns multiple rows, it should therefore be ordered in a way that puts the desired value into the first row.

For a quick overview, the `SubqueryFieldProperty` offers the following queryable property features:

Feature	Supported
Getter	Yes
Setter	No
Filtering	Yes
Annotation	Yes
Updating	No

9.3.2 SubqueryExistenceCheckProperty: Checking whether or not certain objects exist via a subquery

The property class *queryable_properties.properties.SubqueryExistenceCheckProperty* is similar to the *queryable_properties.properties.RelatedExistenceCheckProperty* mentioned above, but can be used to perform any kind of existence check via a subquery. The objects whose existence is to be determined does therefore not have to be related to the class the property is defined on via a `ForeignKey` or another relation field.

To perform this check, the given queryset is wrapped into an `Exists` object, which may also be negated using the property's `negated` argument.

For an example use case, certain applications may be so popular that they receive their own category with the same name as the application. To determine whether or not an application has its own category, a `SubqueryExistenceCheckProperty` could be used:

```
from django.db import models
from queryable_properties.properties import SubqueryExistenceCheckProperty


class Application(models.Model):
    ...  # other fields/properties

    has_own_category = SubqueryExistenceCheckProperty(Category.objects.
↪filter(name=models.OuterRef('name')))
```

For a quick overview, the `SubqueryExistenceCheckProperty` offers the following queryable property features:

Feature	Supported
Getter	Yes
Setter	No
Filtering	Yes
Annotation	Yes
Updating	No

CHAPTER

TEN

ADMIN INTEGRATION

django-queryable-properties comes with an integration in Django's admin, allowing to use queryable properties in various places in both `ModelAdmin` subclasses and inlines. To properly get queryable properties to work with certain features of admins/inlines, *django-queryable-properties* offers specialized base classes that can be used instead of Django's regular base classes:

- *queryable_properties.admin.QueryablePropertiesAdmin* in place of Django's ModelAdmin
- *queryable_properties.admin.QueryablePropertiesStackedInline* in place of Django's StackedInline
- *queryable_properties.admin.QueryablePropertiesTabularInline* in place of Django's TabularInline

For more complex inheritance scenarios, there is also the *queryable_properties.admin.QueryablePropertiesAdminMixin*, which can be added to both admin and inline classes to enable queryable properties functionality while using different admin/inline base classes.

The following table shows the admin/inline options that queryable properties may be referenced in and whether or not each feature requires the use of one of the specialized base classes mentioned above. Queryable properties may be refenced via name in either the listed admin/inline class attributes or in the result of their corresponding `get_*` methods (although there is a special case for `get_list_filter` as described in *Dynamically generating list filters* below).

Admin/inline option	Requires special class	Restrictions/Remarks
`fields/fieldsets`	No	• For properties with a getter or selected properties only • Properties must also be part of `readonly_fields`
`list_display`	No	• For properties with a getter or selected properties only
`list_display_links`	No	• For properties with a getter or selected properties only
`list_filter`	Yes	• For annotatable properties only • Properties must support the lookups used by their list filter class (which is automatically the case if no custom filtering is implemented) • When using the tuple form, the same list filter classes as for regular fields are used, but not all of Django's filter classes are supported as some of them may perform queries that are incompatible with queryable properties
`list_select_propertie`	Yes	• Custom attribute/method of the specialized admin classes listed above • Takes a sequence of queryable property names that will automatically be selected via `select_properties` (see *Selecting annotations*). • For annotatable properties only
`ordering`	Yes	• For annotatable properties only
`readonly_fields`	No	• For properties with a getter or selected properties only
`sortable_by`	No	• For annotatable properties only
`search_fields`	No	• Requires Django 2.1 or higher • Properties must support the lookup used by their respective entry in `search_fields`

10.1 Dynamically generating list filters

Whenever the list filters are to be determined dynamically by overriding `get_list_filter`, proper handling of queryable property items may be disabled as this is also implemented by overriding `get_list_filter`. Therefore, it is important either invoke the queryable property processing by either generating the base filters using a `super` call:

```
from queryable_properties.admin import QueryablePropertiesAdmin


class MyAdmin(QueryablePropertiesAdmin):

    def get_list_filter(self, request):
        list_filter = super(MyAdmin, self).get_list_filter(request)
        # ... process the list filter sequence ...
        # Note: queryable property entries have been replaced with custom callables at␣
↪this point.
        return list_filter
```

... or by utilizing the admin method *queryable_properties.admin.QueryablePropertiesAdminMixin.process_queryable_property_filters()* to postprocess a custom generated filter sequence:

```
from queryable_properties.admin import QueryablePropertiesAdmin


class MyAdmin(QueryablePropertiesAdmin):

    def get_list_filter(self, request):
        list_filter = []
        # ... generate the list filter sequence ...
        # Utilize process_queryable_property_filters to handle queryable property␣
↪filters correctly.
        return self.process_queryable_property_filters(list_filter)
```

CHAPTER

ELEVEN

API

11.1 Module queryable_properties.properties

class queryable_properties.properties.**AggregateProperty**(*aggregate*, ***kwargs*)

A property that is based on an aggregate that is used to provide both queryset annotations as well as getter values.

get_value(*obj*)

Getter method for the queryable property, which will be called when the property is read-accessed.

Parameters
obj (*django.db.models.Model*) – The object on which the property was accessed.

Returns
The getter value.

class queryable_properties.properties.**AnnotationProperty**(*annotation*, ***kwargs*)

A property that is based on a static annotation that is even used to provide getter values.

get_annotation(*cls*)

Construct an annotation representing this property that can be added to querysets of the model associated with this property.

Parameters
cls (*type*) – The model class of which a queryset should be annotated.

Returns
An annotation object.

class queryable_properties.properties.**RelatedExistenceCheckProperty**(*relation_path*, *negated=False*, ***kwargs*)

A property that checks whether related objects to the one that uses the property exist in the database and returns a corresponding boolean value.

Supports queryset filtering and CASE/WHEN-based annotating.

get_value(*obj*)

Getter method for the queryable property, which will be called when the property is read-accessed.

Parameters
obj (*django.db.models.Model*) – The object on which the property was accessed.

Returns
The getter value.

class queryable_properties.properties.**QueryableProperty**(*verbose_name=None*)

Base class for all queryable property definitions, which provide methods for single object as well as queryset interaction.

cached = False

Determines if the result of the getter is cached, like Python's/Django's `cached_property`.

filter_requires_annotation = False

Determines if using the property to filter requires annotating first.

setter_cache_behavior(*obj*, *value*, *return_value*)

Determines what happens if the setter of a cached property is used.

property short_description

Return the verbose name of this property as its short description, which is required for the admin integration.

Returns
The verbose name of this property.

Return type
str

get_value(*obj*)

Getter method for the queryable property, which will be called when the property is read-accessed.

Parameters
obj (*django.db.models.Model*) – The object on which the property was accessed.

Returns
The getter value.

get_filter(*cls*, *lookup*, *value*)

Generate a `django.db.models.Q` object that emulates filtering a queryset using this property.

Parameters

- **cls** (*type*) – The model class of which a queryset should be filtered.
- **lookup** (*str*) – The lookup to use for the filter (e.g. 'exact', 'lt', etc.)
- **value** – The value passed to the filter condition.

Returns
A Q object to filter using this property.

Return type
django.db.models.Q

class queryable_properties.properties.**queryable_property**(*getter=None*, *cached=None*, *annotation_based=False*, ***kwargs*)

A queryable property that is intended to be used as a decorator.

get_value = None

get_filter = None

getter(*method*, *cached=None*)

Decorator for a function or method that is used as the getter of this queryable property. May be used as a parameter-less decorator (`@getter`) or as a decorator with keyword arguments (`@getter(cached=True)`).

Parameters

- **method** (*function*) – The method to decorate.
- **cached** (*bool | None*) – If `True`, values returned by the decorated getter method will be cached. A value of None means no change.

Returns
A cloned queryable property.

Return type
queryable_property

setter(*method*, *cache_behavior=None*)

Decorator for a function or method that is used as the setter of this queryable property. May be used as a parameter-less decorator (`@setter`) or as a decorator with keyword arguments (`@setter(cache_behavior=DO_NOTHING)`).

Parameters

- **method** (*function*) – The method to decorate.
- **cache_behavior** (*function | None*) – A function that defines how the setter interacts with cached values. A value of None means no change.

Returns
A cloned queryable property.

Return type
queryable_property

filter(*method*, *requires_annotation=None*, *lookups=None*, *boolean=False*, *remaining_lookups_via_parent=None*)

Decorator for a function or method that is used to generate a filter for querysets to emulate filtering by this queryable property. May be used as a parameter-less decorator (`@filter`) or as a decorator with keyword arguments (`@filter(requires_annotation=False)`). May be used to define a one-for-all filter function or a filter function that will be called for certain lookups only using the `lookups` argument.

Parameters

- **method** (*function | classmethod | staticmethod*) – The method to decorate.
- **requires_annotation** (*bool | None*) – `True` if filtering using this queryable property requires its annotation to be applied first; otherwise `False`. None if this information should not be changed.
- **lookups** (*collections.Iterable[str] | None*) – If given, the decorated function or method will be used for the specified lookup(s) only. Automatically adds the *LookupFilterMixin* to this property if this is used.
- **boolean** (*bool*) – If `True`, the decorated function or method is expected to be a simple boolean filter, which doesn't take the `lookup` and `value` parameters and should always return a `Q` object representing the positive (i.e. `True`) filter case. The decorator will automatically negate the condition if the filter was called with a `False` value.
- **remaining_lookups_via_parent** (*bool*) – `True` if lookup-based filters should fall back to the base class implementation for lookups without a registered filter function; otherwise `False`. None if this information should not be changed.

Returns
A cloned queryable property.

Return type
queryable_property

annotater(*method*)

Decorator for a function or method that is used to generate an annotation to represent this queryable property in querysets. The *AnnotationMixin* will automatically applied to this property when this decorator is used.

Parameters
method (*function | classmethod | staticmethod*) – The method to decorate.

Returns
A cloned queryable property.

Return type
queryable_property

updater(*method*)

Decorator for a function or method that is used to resolve an update keyword argument for this queryable property into the actual update keyword arguments.

Parameters
method (*function | classmethod | staticmethod*) – The method to decorate.

Returns
A cloned queryable property.

Return type
queryable_property

queryable_properties.properties.**CACHE_RETURN_VALUE**(*descriptor*, *obj*, *value*, *return_value*)

Setter cache behavior function that will update the cache for the cached queryable property on the object in question with the return value of the setter function/method.

Parameters

- **descriptor** (*queryable_properties.properties.base.QueryablePropertyDescriptor*) – The descriptor of the property whose setter was used.
- **obj** (*django.db.models.Model*) – The object the setter was used on.
- **value** – The value that was passed to the setter.
- **return_value** – The return value of the setter function/method.

queryable_properties.properties.**CACHE_VALUE**(*descriptor*, *obj*, *value*, *return_value*)

Setter cache behavior function that will update the cache for the cached queryable property on the object in question with the (raw) value that was passed to the setter.

Parameters

- **descriptor** (*queryable_properties.properties.base.QueryablePropertyDescriptor*) – The descriptor of the property whose setter was used.
- **obj** (*django.db.models.Model*) – The object the setter was used on.
- **value** – The value that was passed to the setter.
- **return_value** – The return value of the setter function/method.

queryable_properties.properties.**CLEAR_CACHE**(*descriptor*, *obj*, *value*, *return_value*)

Setter cache behavior function that will clear the cached value for a cached queryable property on objects after the setter was used.

Parameters

- **descriptor** (*queryable_properties.properties.base.QueryablePropertyDescriptor*) – The descriptor of the property whose setter was used.
- **obj** (*django.db.models.Model*) – The object the setter was used on.
- **value** – The value that was passed to the setter.
- **return_value** – The return value of the setter function/method.

queryable_properties.properties.**DO_NOTHING**(*descriptor*, *obj*, *value*, *return_value*)

Setter cache behavior function that will do nothing after the setter of a cached queryable property was used, retaining previously cached values.

Parameters

- **descriptor** (*queryable_properties.properties.base.QueryablePropertyDescriptor*) – The descriptor of the property whose setter was used.
- **obj** (*django.db.models.Model*) – The object the setter was used on.
- **value** – The value that was passed to the setter.
- **return_value** – The return value of the setter function/method.

class queryable_properties.properties.**AnnotationGetterMixin**(*cached=None*, **args*, ***kwargs*)

A mixin for queryable properties that support annotation and use their annotation even to provide the value for their getter (i.e. perform a query to retrieve the getter value).

get_queryset(*model*)

Construct a base queryset for the given model class that can be used to build queries in property code.

Parameters
model – The model class to build the queryset for.

get_queryset_for_object(*obj*)

Construct a base queryset that can be used to retrieve the getter value for the given object.

Parameters
obj (*django.db.models.Model*) – The object to build the queryset for.

Returns
A base queryset for the correct model that is already filtered for the given object.

Return type
django.db.models.QuerySet

class queryable_properties.properties.**AnnotationMixin**(**args*, ***kwargs*)

A mixin for queryable properties that allows to add an annotation to represent them to querysets.

property admin_order_field

Return the field name for the ordering in the admin, which is simply the property's name since it's annotatable.

Returns
The field name for ordering in the admin.

Return type
str

get_annotation(*cls*)

Construct an annotation representing this property that can be added to querysets of the model associated with this property.

Parameters
cls (*type*) – The model class of which a queryset should be annotated.

Returns
An annotation object.

queryable_properties.properties.**boolean_filter**(*method*)

Decorator for individual filter methods of classes that use the *LookupFilterMixin* to register the methods that are simple boolean filters (i.e. the filter can only be called with a `True` or `False` value). This automatically restricts the usable lookups to `exact`. Decorated methods should not expect the `lookup` and `value` parameters and should always return a `Q` object representing the positive (i.e. `True`) filter case. The decorator will automatically negate the condition if the filter was called with a `False` value.

Parameters
method (*function*) – The method to decorate.

Returns
The decorated method.

Return type
function

class queryable_properties.properties.**LookupFilterMixin**(**args*, ***kwargs*)

A mixin for queryable properties that allows to implement queryset filtering via individual methods for different lookups.

classmethod lookup_filter(**lookups*)

Decorator for individual filter methods of classes that use the *LookupFilterMixin* to register the decorated methods for the given lookups.

Parameters
lookups (*str*) – The lookups to register the decorated method for.

Returns
The actual internal decorator.

Return type
function

classmethod boolean_filter(*method*)

Decorator for individual filter methods of classes that use the *LookupFilterMixin* to register the methods that are simple boolean filters (i.e. the filter can only be called with a `True` or `False` value). This automatically restricts the usable lookups to `exact`. Decorated methods should not expect the `lookup` and `value` parameters and should always return a `Q` object representing the positive (i.e. `True`) filter case. The decorator will automatically negate the condition if the filter was called with a `False` value.

Parameters
method (*function*) – The method to decorate.

Returns
The decorated method.

Return type
function

queryable_properties.properties.**lookup_filter**(**lookups*)

Decorator for individual filter methods of classes that use the *LookupFilterMixin* to register the decorated methods for the given lookups.

Parameters
lookups (*str*) – The lookups to register the decorated method for.

Returns
The actual internal decorator.

Return type
function

class queryable_properties.properties.**SetterMixin**

A mixin for queryable properties that also define a setter.

set_value(*obj*, *value*)

Setter method for the queryable property, which will be called when the property is write-accessed.

Parameters

- **obj** (*django.db.models.Model*) – The object on which the property was accessed.
- **value** – The value to set.

class queryable_properties.properties.**UpdateMixin**

A mixin for queryable properties that allows to use themselves in update queries.

get_update_kwargs(*cls*, *value*)

Resolve an update keyword argument for this property into the actual keyword arguments to emulate an update using this property.

Parameters

- **cls** (*type*) – The model class of which an update query should be performed.
- **value** – The value passed to the update call for this property.

Returns
The actual keyword arguments to set in the update call instead of the given one.

Return type
dict

class queryable_properties.properties.**MappingProperty**(*attribute_path*, *output_field*, *mappings*, *default=None*, ***kwargs*)

A property that translates values of an attribute into other values using defined mappings.

get_value(*obj*)

Getter method for the queryable property, which will be called when the property is read-accessed.

Parameters
obj (*django.db.models.Model*) – The object on which the property was accessed.

Returns
The getter value.

get_annotation(*cls*)

Construct an annotation representing this property that can be added to querysets of the model associated with this property.

Parameters
cls (*type*) – The model class of which a queryset should be annotated.

Returns
An annotation object.

class queryable_properties.properties.**RangeCheckProperty**(*min_attribute_path*, *max_attribute_path*, *value*, *include_boundaries=True*, *in_range=True*, *include_missing=False*, ***kwargs*)

A property that checks if a static or dynamic value is contained in a range expressed by two field values and returns a corresponding boolean value.

Supports queryset filtering and `CASE`/`WHEN`-based annotating.

get_value(*obj*)

Getter method for the queryable property, which will be called when the property is read-accessed.

Parameters
obj (*django.db.models.Model*) – The object on which the property was accessed.

Returns
The getter value.

class queryable_properties.properties.**ValueCheckProperty**(*attribute_path*, **values*, ***kwargs*)

A property that checks if an attribute of a model instance or a related object contains a certain value or one of multiple specified values and returns a corresponding boolean value.

Supports queryset filtering and `CASE`/`WHEN`-based annotating.

get_value(*obj*)

Getter method for the queryable property, which will be called when the property is read-accessed.

Parameters
obj (*django.db.models.Model*) – The object on which the property was accessed.

Returns
The getter value.

class queryable_properties.properties.**SubqueryExistenceCheckProperty**(*queryset*, *negated=False*, ***kwargs*)

A property that checks whether or not certain objects exist in the database using a custom subquery.

class queryable_properties.properties.**SubqueryFieldProperty**(*queryset*, *field_name*, *output_field=None*, ***kwargs*)

A property that returns a field value contained in a subquery, extracting it from the first row of the subquery's result set.

11.2 Module queryable_properties.admin

class queryable_properties.admin.**QueryablePropertiesAdmin**(**args*, ***kwargs*)

Base class for admin classes which allows to use queryable properties in various admin features.

Intended to be used in place of Django's regular `ModelAdmin` class.

class queryable_properties.admin.**QueryablePropertiesAdminMixin**(**args*, ***kwargs*)

A mixin for admin classes including inlines that allows to use queryable properties in various admin features.

list_select_properties = ()

A sequence of queryable property names that should be selected.

get_list_select_properties(*request*)

Wrapper around the `list_select_properties` attribute that allows to dynamically create the list of queryable property names to select based on the given request.

Parameters
request (*django.http.HttpRequest*) – The request to the admin.

Returns
A sequence of queryable property names to select.

Return type
collections.Sequence[str]

process_queryable_property_filters(*list_filter*)

Process a sequence of list filters to create a new sequence in which queryable property references are replaced with custom callables that make them compatible with Django's filter workflow.

Parameters
list_filter (*collections.Sequence*) – The list filter sequence.

Returns
The processed list filter sequence.

Return type
list

class queryable_properties.admin.**QueryablePropertiesStackedInline**(**args*, ***kwargs*)

Base class for stacked inline classes which allows to use queryable properties in various admin features.

Intended to be used in place of Django's regular `StackedInline` class.

class queryable_properties.admin.**QueryablePropertiesTabularInline**(**args*, ***kwargs*)

Base class for tabular inline classes which allows to use queryable properties in various admin features.

Intended to be used in place of Django's regular `TabularInline` class.

11.3 Module queryable_properties.managers

class queryable_properties.managers.**QueryablePropertiesManager**(**args*, ***kwargs*)

A special manager class that allows to use queryable properties methods and returns *QueryablePropertiesQuerySet* instances.

classmethod get_for_model(*model*, *using=None*, *hints=None*)

Get a new manager with queryable properties functionality for the given model.

Parameters

- **model** – The model class for which the manager should be built.
- **using** (*str | None*) – An optional name of the database connection to use.
- **hints** (*dict | None*) – Optional hints for the db connection.

Returns
A new manager with queryable properties functionality.

Return type
QueryablePropertiesManager

class queryable_properties.managers.**QueryablePropertiesManagerMixin**(**args*, ***kwargs*)

A mixin for Django's `django.db.models.Manager` objects that allows to use queryable properties methods and returns *QueryablePropertiesQuerySet* instances.

classmethod apply_to(*manager*)

Copy the given manager and apply this mixin (and thus queryable properties functionality) to it, returning a new manager that allows to use queryable property interaction.

Parameters
manager (*Manager*) – The manager to apply this mixin to.

Returns
A copy of the given manager with queryable properties functionality.

Return type
QueryablePropertiesManager

select_properties(**names*)

Return a new queryset and add the annotations of the queryable properties with the specified names to this query. The annotation values will be cached in the properties of resulting model instances, regardless of the regular caching behavior of the queried properties.

Parameters
names – Names of queryable properties.

Returns
A copy of this queryset with the added annotations.

Return type
QuerySet

class queryable_properties.managers.**QueryablePropertiesQuerySet**(**args*, ***kwargs*)

A special queryset class that allows to use queryable properties in its filter conditions, annotations and update queries.

classmethod get_for_model(*model*)

Get a new queryset with queryable properties functionality for the given model. The queryset is built using the model's default manager.

Parameters
model – The model class for which the queryset should be built.

Returns
A new queryset with queryable properties functionality.

Return type
QueryablePropertiesQuerySet

class queryable_properties.managers.**QueryablePropertiesQuerySetMixin**(**args*, ***kwargs*)

A mixin for Django's `django.db.models.QuerySet` objects that allows to use queryable properties in filters, annotations and update queries.

classmethod apply_to(*queryset*)

Copy the given queryset and apply this mixin (and thus queryable properties functionality) to it, returning a new queryset that allows to use queryable property interaction.

Parameters
queryset (*QuerySet*) – The queryset to apply this mixin to.

Returns
A copy of the given queryset with queryable properties functionality.

Return type
QueryablePropertiesQuerySet

`select_properties`(**names*)

Add the annotations of the queryable properties with the specified names to this query. The annotation values will be cached in the properties of resulting model instances, regardless of the regular caching behavior of the queried properties.

Parameters
names – Names of queryable properties.

Returns
A copy of this queryset with the added annotations.

Return type
QuerySet

11.4 Module queryable_properties.utils

queryable_properties.utils.**get_queryable_property**(*model*, *name*)

Retrieve the *queryable_properties.properties.QueryableProperty* object with the given attribute name from the given model class or raise an error if no queryable property with that name exists on the model class.

Parameters

- **model** (*type*) – The model class to retrieve the property object from.
- **name** (*str*) – The name of the property to retrieve.

Returns
The queryable property.

Return type
queryable_properties.properties.QueryableProperty

queryable_properties.utils.**prefetch_queryable_properties**(*model_instances*, **property_paths*)

Populate the queryable property caches for a list of model instances based on the given property paths.

Parameters

- **model_instances** (*collections.Sequence*) – The model instances to prefetch the property values for. The instances may be objects of different models as long as the given property paths are valid for all of them.
- **property_paths** (*str*) – The paths to the properties whose values should be fetched, which are need to be annotatable. The paths may contain the lookup separator to fetch values of properties on related objects (make sure that the related objects are already prefetched to avoid additional queries).

queryable_properties.utils.**reset_queryable_property**(*obj*, *name*)

Reset the cached value of the queryable property with the given name on the given model instance. Read-accessing the property on this model instance at a later point will therefore execute the property's getter again.

Parameters

- **obj** (*django.db.models.Model*) – The model instance to reset the cached value on.
- **name** (*str*) – The name of the queryable property.

11.5 Module queryable_properties.exceptions

exception queryable_properties.exceptions.**QueryablePropertyDoesNotExist**

The requested queryable property does not exist.

exception queryable_properties.exceptions.**QueryablePropertyError**

Some kind of problem with a queryable property.

CHAPTER

TWELVE

CHANGELOG

12.1 master (unreleased)

12.2 1.9.2 (2024-03-06)

- Fixed an error that prevented querysets with queryable property features from being used as filter values in `__in` filters (i.e. implicit subqueries)

12.3 1.9.1 (2024-01-09)

- Fixed resolving of filter conditions of aggregate properties in cases where a property was accessed via relation

12.4 1.9.0 (2023-12-05)

- Added support for Django 5.0
- Added support for Python 3.12
- Added options to create querysets/managers with queryable property features on demand and without having to define a manager on the corresponding model
- Queryable properties can now be populated in raw queries by using the property name as SQL column name

12.5 1.8.5 (2023-11-13)

- Selected queryable properties are no longer aliased with a unique name in queries and use their regular name instead (also fixes errors that occurred when queries use themselves as subqueries recursively, e.g. in sliced prefetches)

12.6 1.8.4 (2023-04-05)

- Added support for Django 4.2
- Added support for Python 3.11

12.7 1.8.3 (2022-08-06)

- Added support for Django 4.1

12.8 1.8.2 (2022-06-08)

- Fixed queryset cloning in conjunction with positional arguments in Django versions below 1.9

12.9 1.8.1 (2022-03-05)

- Fixed erroneous transformations of querysets with queryable properties functionality into `.values()` querysets under rare circumstances in Django versions above 3.0
- Fixed the ability to pickle `.values()`/`.values_list()` querysets with queryable properties functionality in Django versions below 1.9
- Fixed the erroneous inclusion of values of queryable properties that are used for ordering without being explicitly selected in `.values()`/`.values_list()` querysets in Django versions below 1.8

12.10 1.8.0 (2021-12-07)

- Added support for Django 4.0
- Added new ready-to-use queryable property implementations for properties based on subqueries (`SubqueryFieldProperty`, `SubqueryExistenceCheckProperty`)
- `RelatedExistenceCheckProperty` objects can now be configured as negated to be able to check for the non-existence of related objects

12.11 1.7.1 (2021-11-01)

- Added support for Python 3.10
- Fixed duplicate selections of `GROUP BY` columns when multiple aggregate properties are selected, which also led to wrong property values, in Django versions below 1.8

12.12 1.7.0 (2021-07-05)

- Added the `prefetch_queryable_properties` utility function which allows to efficiently query property values for model instances that were already loaded from the database beforehand
- Extended the `LookupFilterMixin` to allow to define a filter function/method that handles all lookups that don't use an explicitly registered function/method
- Values for queryable properties with setters can now also be set using initializer keyword arguments of their respective models

12.13 1.6.1 (2021-04-19)

- Fixed the `AnnotationGetterMixin` and its subclasses to be able to work with nested properties correctly regardless of whether or not the model's base manager uses the queryable properties extensions
- Fixed the admin filter that displays all possible options to be able to work with nested properties correctly regardless of whether or not the model's default manager uses the queryable properties extensions

12.14 1.6.0 (2021-04-06)

- Added support for Django 3.2
- Queryable properties can now define a verbose name that can be used in UI representations
- Added a Django admin integration that allows to reference queryable properties like regular model fields in various admin options
- Fixed the construction of `GROUP BY` clauses when using annotations based on aggregate queryable properties in Django 1.8

12.15 1.5.0 (2020-12-30)

- Added an option to implement annotation-based properties that use their annotation to query their getter value from the database
- Added a new ready-to-use queryable property implementation for properties that check whether or not certain related objects exist (`RelatedExistenceCheckProperty`)
- Added a new ready-to-use queryable property implementation for properties that map field/attribute values to other values (`MappingProperty`)

12.16 1.4.1 (2020-10-21)

- String representations of queryable properties do now contain the full Python path instead of the Django model path (also fixes an error that occurred when building the string representation for a property on an abstract model that was defined outside of the installed apps)

12.17 1.4.0 (2020-10-17)

- `ValueCheckProperty` and `RangeCheckProperty` objects can now take more complex attribute paths instead of simple field/attribute names
- `RangeCheckProperty` objects now have an option that determines how to treat missing values to support ranges with optional boundaries
- Added a new ready-to-use queryable property implementation for properties based on simple aggregates (`AggregateProperty`)

12.18 1.3.1 (2020-08-04)

- Added support for Django 3.1
- Refactored decorator-based properties to be more maintainable and memory-efficient and documented a way to use them without actually decorating

12.19 1.3.0 (2020-05-22)

- Added an option to implement simplified custom boolean filters utilizing lookup-based filters
- Fixed the ability to use the `classmethod` or `staticmethod` decorators with lookup-based filter methods for decorator-based properties
- Fixed the queryable property resolution in `When` parts of conditional updates
- Fixed the ability to use conditional expressions directly in `.filter`/`.exclude` calls in Django 3.0

12.20 1.2.1 (2019-12-03)

- Added support for Django 3.0

12.21 1.2.0 (2019-10-21)

- Added a mixin that allows custom filters for queryable properties (both class- and decorator-based) to be implemented using multiple functions/methods for different lookups
- Added some ready-to-use queryable property implementations (`ValueCheckProperty`, `RangeCheckProperty`) to simplify common code patterns
- Added a standalone version of six to the package requirements

12.22 1.1.0 (2019-06-23)

- Queryable property filters (both annotation-based and custom) can now be used across relations when filtering querysets (i.e. a queryset can now be filtered by a queryable property on a related model)
- Queryset annotations can now refer to annotatable queryable properties defined on a related model
- Querysets can now be ordered by annotatable queryable properties defined on a related model
- Filters and annotations that reference annotatable queryable properties will not select the queryable property annotation anymore in Django versions below 1.8 (ordering by such a property will still lead to a selection in these versions)
- Fixed unnecessary selections of queryable property annotations in querysets that don't return model instances (i.e. queries with `.values()` or `.values_list()`)
- Fixed unnecessary fields in `GROUP BY` clauses in querysets that don't return model instances (i.e. queries with `.values()` or `.values_list()`) in Django versions below 1.8
- Fixed an infinite recursion when constructing the `HAVING` clause for annotation-based filters that are not an aggregate in Django 1.8

12.23 1.0.2 (2019-06-02)

- The `lookup` parameter of custom filter implementations of queryable properties will now receive the combined lookup string if multiple lookups/transforms are used at once instead of just the first lookup/transform
- Fixed the construction of `GROUP BY` clauses when annotating queryable properties based on aggregates
- Fixed the construction of `HAVING` clauses when annotating queryable properties based on aggregates in Django versions below 1.9
- Fixed the ability to pickle queries and querysets with queryable properties functionality in Django versions below 1.6

12.24 1.0.1 (2019-05-11)

- Added support for Django 2.2

12.25 1.0.0 (2018-12-31)

- Initial release

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

q

INDEX