django-queryable-properties Documentation

Release 1.4.1

Marcus Klöpfel

Oct 21, 2020

Contents

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

CHAPTER 1

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 (from bad to good):

- Performing special annotations only in the exact places that they are needed in while possibly even duplicating the code if there are multiple such places.
- Implementing functions/methods that perform the annotations to avoid duplicating code.
- Implementing a custom model manager/queryset class to allow the usage of these special annotations whenever dealing with a queryset.

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.

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.

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.

```python
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 2

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
3.1	3.8, 3.7, 3.6
3.0	3.8, 3.7, 3.6
2.2	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	3.6, 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

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

CHAPTER 3

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_property` and its methods as decorators:

```python
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):
```

(continues on next page)

(continued from previous page)

```
        # 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 anything 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
```

(continues on next page)

(continued from previous page)

```


class ApplicationVersion(models.Model):
    ...

    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 `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 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 Using the required manager/queryset

If we were to actually implement queryset-related logic in the examples above, the `ApplicationVersion` model would be missing one small detail to actually be able to use the queryable properties in querysets: the model must use a special queryset class, which can most easily be achieved by using a special manager:

```
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()`). If there's a need to use another special queryset class, `queryable_properties` also comes with a mixin to add its logic to other custom querysets: `queryable_properties.managers.QueryablePropertiesQuerySetMixin`. A manager class can then be generated from the queryset class using `CustomQuerySet.as_manager()` or `CustomManager.from_queryset(CustomQuerySet)`.

Using the special manager/queryset class may not only be important for models that define queryable properties. Since most features of queryable properties can also be used on related models in queryset operations, the manager is required whenever queryable property functionality should be offered, even if the corresponding 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.

CHAPTER 4

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 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 a utility function `reset_queryable_property` that comes with the `queryable_properties` package.

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 `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('.')
```

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:

```python
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)

    @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:]
```

(continues on next page)

(continued from previous page)

```
        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 5

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:

```python
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."""
```

(continues on next page)

(continued from previous page)

```
        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()
        # 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 `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 `LookupFilterMixin` (which implements `get_filter`) must be used in conjunction with the `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):

    def get_value(self, obj):
```

(continues on next page)

(continued from previous page)

```
        """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'`)

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):
```

(continues on next page)

(continued from previous page)

```
        """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 `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.
        # 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 6

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
```

(continues on next page)

(continued from previous page)

```
    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 `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 *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):
```

(continues on next page)

(continued from previous page)

```
        """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)

    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.

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.

In querysets that return model instances, this may have performance implications due to the additional columns that are queried, but the annotation values will be discarded when model instances are created. This is done because selected queryable properties behave differently (see below), and this behavior is meant to be consistent across all supported Django versions.

The selection of the queryable property annotations in these scenarios may also affect queries with `.distinct()` calls (since the `DISTINCT` clause also applies to the annotation) or `.values()`/`.values_list()` queries, which will return the annotation column in addition to the ones specified in `.values()`/`.values_list()`.

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 (3.6 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 *cached queryable properties*** 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.

6.4 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 7

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` argument for a queryable property will simply be translated into `update` values for actual model fields.

7.1 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.

```python
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 `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.

7.2 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.

7.3 Limitations

7.3.1 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.

7.3.2 Expression-based update values

Using expression-based values (like `F` objects or conditional updates) 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 8

Common Patterns

django-queryable-properties offers some fully implemented properties for common code patterns out of the box. They are parameterizable and are supposed to help remove boilerplate for recurring types of properties while making them usable in querysets at the same time.

8.1 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:

```python
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
```

(continues on next page)

(continued from previous page)

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

    @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 `ValueCheckProperty` of the *django-queryable-properties* package 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

8.1.1 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 `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):

- `AttributeErrors` 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.

8.2 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 `RangeCheckProperty` of the *django-queryable-properties* package 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

8.2.1 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_boundaries`	`include_missing`	`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`).

8.3 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 allows to define such properties in only one line of code. For example, the `Application` model could receive a simple property that returns the number of versions like this:

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


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

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

Since the getter also performs a query to retrieve the aggregated value, the `AggregateProperty` initializer also allows to mark the property as cached using an additional `cached` parameter (defaults to `False`). 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.

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

CHAPTER 9

API

9.1 Module `queryable_properties.properties`

class `queryable_properties.properties.`**`QueryableProperty`**

Base class for all queryable properties, which are basically simple descriptors with some added methods for queryset interaction.

`cached = False`

Determines if the result of the getter is cached, like 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.

`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*)

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

`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*)

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.

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**(*prop*, *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

- **prop** (QueryableProperty) – 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**(*prop*, *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

- **prop** (QueryableProperty) – 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**(*prop*, *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

- **prop** (QueryableProperty) – 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`**(*prop*, *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

- **`prop`** (`QueryableProperty`) – 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.`**`AggregateProperty`**(*aggregate*, *cached=False*)

`__init__`(*aggregate*, *cached=False*)

Initialize a new property that gets its value by retrieving an aggregated value from the database.

Parameters

- **`aggregate`** (*`django.db.models.Aggregate`*) – The aggregate to use to determine the value of this property.
- **`cached`** (*`bool`*) – Whether or not this property should use a cached getter. If the property is not cached, the getter will perform the corresponding aggregate query on every access.

`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*)

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.

`__init__`(*min_attribute_path*, *max_attribute_path*, *value*, *include_boundaries=True*, *in_range=True*, *include_missing=False*)

Initialize a new property that checks if a value is contained in a range expressed by two field values.

Parameters

- **`min_attribute_path`** (*`str`*) – The name of the attribute to get the lower boundary from. May also be a more complex path to a related attribute using dot-notation (like with `operator.attrgetter()`). If an intermediate value on the path is None, it will be treated as a missing value instead of raising an exception. The behavior is the same if an intermediate value raises an ObjectDoesNotExist error.
- **`max_attribute_path`** (*`str`*) – The name of the attribute to get the upper boundary from. The same behavior as for the lower boundary applies.
- **`value`** – The value which is tested against the boundary. May be a callable which can be called without any arguments, whose return value will then be used as the test value.

- **include_boundaries** (*bool*) – Whether or not the value is considered a part of the range if it is exactly equal to one of the boundaries.
- **in_range** (*bool*) – Configures whether the property should return *True* if the value is in range (*in_range=True*) or if it is out of the range (*in_range=False*). This also affects the impact of the *include_boundaries* and *include_missing* parameters.
- **include_missing** (*bool*) – Whether or not a missing value is considered a part of the range (see the description of *min_attribute_path*). Useful e.g. for nullable fields.

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

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.

__init__(*attribute_path*, **values*)

Initialize a new property that checks for certain field values.

Parameters

- **attribute_path** (*str*) – The name of the attribute to compare against. May also be a more complex path to a related attribute using dot-notation (like with `operator.attrgetter()`). If an intermediate value on the path is None, it will be treated as "no match" instead of raising an exception. The behavior is the same if an intermediate value raises an ObjectDoesNotExist error.
- **values** – The value(s) to check for.

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

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

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.`**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

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

9.2 Module `queryable_properties.managers`

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.

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.

`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

`managers.`**`QueryablePropertiesManager = <class 'django.db.models.manager.ManagerFromQueryable`**

9.3 Module `queryable_properties.utils`

`queryable_properties.utils.`**`MISSING_OBJECT = <object object>`**

Arbitrary object to represent that an object in an attribute chain is missing.

`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.`**`reset_queryable_property`**(*obj*, *name*)

Reset the cached value of the queryable property with the given name on the given model instance. Re-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.

9.4 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 10

Changelog

10.1 master (unreleased)

10.2 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)

10.3 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`)

10.4 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

10.5 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

10.6 1.2.1 (2019-12-03)

- Added support for Django 3.0

10.7 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

10.8 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

10.9 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

10.10 1.0.1 (2019-05-11)

- Added support for Django 2.2

10.11 1.0.0 (2018-12-31)

- Initial release

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index