Django Shared Property Documentation

Release 0.8.0

Matthew Schinckel

Sep 21, 2023

CONTENTS:

CHAPTER

ONE

DJANGO SHARED PROPERTY

Properties that are both ORM expressions and python code.

- Free software: MIT license
- Documentation: https://django-shared-property.readthedocs.io.

1.1 Installation:

```
$ pip install django_shared_property
```

1.2 Philosophy:

I often find that I have annotations in Django querysets that are based on one or more other fields, and are used frequently. In some cases I have even been known to ensure these annotations are always available on the model using a custom queryset/manager.

But, unlike Python properties, these annotations are not "live". If, for example, you have the following:

```
class FullNameQueryset(models.query.QuerySet):
    def with_full_name(self):
        return self.annotate(
          full_name=Concat(models.F('first_name'), models.Value(' '), models.F('last_name
↪')),
        )


class Person(models.Model):
    first_name = models.TextField()
    last_name = models.TextField()

    objects = FullNameQueryset.as_manager()
```

Then, you can do a *person = Person.objects.with_full_name().get(pk=1)*, and then reference *person.full_name*.

But, if you modify *person.first_name*, you'd need to write it back to the database, and then reload it. Which may not be ideal, and at best requires two database operations.

django-shared-property allows you to have properties that automatically act as an annototation, allowing you to define the expression, and have Django use that operation within any database query. It can then be used in a filter (or further annotation), even across relationships. And finally, if any local changes are made to the object that would affect the value when stored in the database, then the property value will also update in Python.

1.3 Show me how!

Similar to a Python property, a django-shared-property requires a method that takes no arguments. It should, however, return a Django Expression. For example, following our annotation above:

```
from django.db import models
from django.db.models.functions import Concat
from django_shared_property.decorator import shared_property


class Person(models.Model):
    first_name = models.TextField()
    last_name = models.TextField()

    @shared_property
    def full_name(self):
        return Concat(models.F('first_name'), models.Value(' '), models.F('last_name'))
```

You then reference it just like any other field.

```
Person.objects.filter(full_name__contains='Bob')
```

1.4 What else can it do?

Shared properties can reference any number of fields on the model, and even other shared properties, just like with annotations. They can even reference fields from related models, using the familiar `models.F('relation__field')` lookup syntax. You can also use some Django expressions (such as Concat, Lower, Upper), where there is a clear relationship to a Python concept.

- Case
- When
- F
- Q (Specifically, within a When, but it could work elsewhere)
- Concat
- Value
- Lower and Upper (but only on Python objects that have these as attributes)
- ExpressionWrapper
- CombinedExpresson

- Coalesce (but see the note below)

Within the context of a Q expression, you can use `__isnull` and `__exact` lookups.

You can even refer to constants in your Python file, such as the different values of an Enum. The return value of your python object will then correctly return instances of the Enum.

If your chosen expression/function/value does not work, then it may be possible to implement it (see below).

Shared properties should be pure functions - they must not refer to `self` (indeed, this will cause an error), and should not refer to variables, as they will be executed at times other than when they are about to be decorated.

1.5 How does it work?

Because of the limit that the decorated function be a pure function, we are able to execute the callable, using the result as a Django expression.

The Django part is relatively straightforward. The expression returned by the method that is decorated as a shared_property is used in a context that looks a bit like an annotation - however there are a couple of things that need to be done to ensure that the expression has the correct data available to it to make sure it points at the correct tables. We indicate to Django that it should not be written back to the database by marking it as a `private` field.

Creating the Python property is a bit trickier. We still need the expression, but we build an Abstract Syntax Tree based on the expression. We then compile this into a callable object that we use as the property.

In a little more detail:

- Call the decorated function, returning the Expression
- Use Python's introspection tools to examine the expression (and it's "source expressions") and a recursive descent parser to build an AST equivalent to the expression. Specifically, the AST contains a function definition.
- Compile this AST into a code object
- `eval` this code object with the correct context to pull in any constants from outside the namespace.
- Extract the newly defined function, and use it for the callable in our property.

1.6 Advanced Use

Sometimes you want to define the callable yourself: there is an alternate syntax for that. This could be where the expression has not been defined, or it's possible to create a more efficient callable by hand:

```
class MyModel(models.Model):
    # other fields

    @shared_property(Case(
        When(models.Q(x__gte=2, x__lt=5), then=models.Value('B')),
        When(models.Q(x__lt=2), then=models.Value('A')),
        default=models.Value('C'),
        output_field=models.TextField(),
    )
    def state(self):
        if 2 <= self.x < 5:
            return 'B'
        elif x < 2:
```

(continues on next page)

(continued from previous page)

```
            return 'A'
        return 'C'

    @shared_property(Coalesce(
      CombinedExpression(F('expiry_date'), '<', Func(function='current_timestamp')),
      models.Value(True),
    ))
    def active(self):
        return self.expiry_date is None or self.expiry_date < timezone.now()
```

In this specific case, the code that is generated would be fairly similar (although it would not use the `a < b < c` idiom), however it shows how it is possible to to explicitly provide the python code. Please note that the onus of responsibility is on the developer to ensure that the expression and function are equivalent in this context.

The second example shows where a python comparison doesn't quite map to the SQL code: the `COALESCE(expiry_date < now(), true)` relies on SQL comparisons involving NULL to also return NULL, but in Python you cannot do this.

Also note that in this case only a single expression may be used as the argument to the decorator.

1.7 Registering Expressions

It is possible to register your own expressions. The structure is quite strict, and you'll need to reference the parser instance as well as the incoming expression. There's sometimes quite a bit of work to turn the Expression into (a) the correct Python, and then (b) the AST that is required.

```
from django_shared_property.parser import register


@register
def handle_foo(parser, expression):
    # This assumes a foo() function in python that matches a foo()
    # function in SQL, neither of which takes arguments.
    return Call(
        func=Name(id='foo', **parser.file),
        args=[],
        keywords=[],
        kwonlyargs=[],
        **parser.file,
    )


class Foo(Func):
    function = 'foo'


class MyModel(models.Model):
    # ...

    @shared_property
    def the_foo(self):
        return Foo()
```

This is a toy example - try looking in the `parser` module for other examples.

1.8 Limitations

Use of the django queryset methods defer/only prevent any shared properties from loading. However, because of the way the feature works, you would still be able to use this property - at least in the case where the referenced fields are local.

When you use a shared property that references a related model, and then try to filter on this, you cannot perform a count or exists query. See https://github.com/schinckel/django-shared-property/issues/2

1.9 Credits

Developed by Matthew Schinckel.

This package was created with Cookiecutter and the wboxx1/cookiecutter-pypackage-poetry project template.

CHAPTER

TWO

INSTALLATION

2.1 Stable release

To install Django Shared Property, run this command in your terminal:

```
$ pip install django_shared_property
```

This is the preferred method to install Django Shared Property, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for Django Shared Property can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/schinckel/django-shared-property
```

Or download the tarball:

```
$ curl  -OL https://github.com/schinckel/django-shared-property/tarball/main
```

Once you have a copy of the source, you can install into poetry virtual environment with:

```
$ poetry install
```

CHAPTER

THREE

USAGE

To use Django Shared Property in a project, you need to decorate a method that (a) takes no arguments other than self, (b) does not reference self in the method body, and (c) returns a single expression that is a Django ORM expression object:

```
from django_shared_property.decorator import shared_property

class Person(models.Model):
    first_name = models.TextField()
    last_name = models.TextField()
    preferred_name = models.TextField(null=True, blank=True)

    @shared_property
    def display_name(self):
        first_last = Concat(F('first_name'), Value(' '), F('last_name'))
        first_preferred_last = Concat(
            F('first_name'),
            Value(' ('), F('preferred_name'), Value(') '),
            F('last_name'),
        )
        return Case(
            When(preferred_name__isnull=True, then=first_last),
            When(preferred_name__exact=Value(''), then=first_last),
            default=first_preferred_last,
            output_field=models.TextField()
        )
```

This will result in two things being added to the Model.

1. The expression that is returned will be used as a computed field - this will be usable in any queryset filters, and will be available in a .values() iff it is referenced directly.
2. The expression will be turned into a python property and added onto the Model class directly - accessing this property will evaluate the python equivalent to the expression.

For instance:

```
>>> Person.objects.create(first_name='Bob', last_name='Dobalino')
>>> person = Person.objects.filter(display_name='Bob Dobalino').first()
>>> print(person.display_name)
Bob Dobalino
>>> person.preferred_name = 'Dobs'
```

(continues on next page)

(continued from previous page)

```
>>> print(person.display_name)
Bob (Dobs) Dobalino
```

Note that this evaluation is based on the current values of any referenced fields, as opposed to how an annotation of the expression would only be based on what is stored in the database.

CHAPTER

FOUR

DJANGO_SHARED_PROPERTY

4.1 django_shared_property package

4.1.1 Submodules

4.1.2 django_shared_property.apps module

4.1.3 django_shared_property.decorator module

4.1.4 django_shared_property.expressions module

4.1.5 django_shared_property.parser module

class django_shared_property.parser.**Parser**(*function*)

Bases: object

build_expression(*expression*)

handle_bool(*boolean*)

handle_case(*case*)

handle_coalesce(*expression*)

next(

itertools.chain(
(x for x in expression.get_source_expressions() where x is not None), (None,)

)

)

handle_combinedexpression(*expression*)

handle_concat(*concat*)

handle_concatpair(*pair*)

handle_exact(*exact*)

handle_expressionwrapper(*expression*)

handle_f(*f*)

handle_lower(*expression*)

handle_q(*q*)

handle_upper(*expression*)

handle_value(*value*)

handle_when(*when*, **others*)

class django_shared_property.parser.**register**(*func*)

Bases: object

4.1.6 Module contents

Top-level package for Django Shared Property.

CHAPTER

FIVE

CONTRIBUTING

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

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/schinckel/django-shared-property/issues.

If you are reporting a bug, please include:

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

5.1.2 Fix Bugs

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

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.1.4 Write Documentation

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

5.1.5 Submit Feedback

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

If you are proposing a feature:

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

5.2 Get Started!

Ready to contribute? Here's how to set up *Django Shared Property* for local development.

1. Fork the *Django Shared Property* repo on GitHub.
2. Clone your fork locally:

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

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

```
$ cd django-shared-property/
$ poetry install
$ poetry shell
```

4. Create a branch for local development:

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

   Now you can make your changes locally.

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

```
$ flake8 src/django_shared_property tests
$ tox
```

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

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

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

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

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

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/schinckel/django-shared-property/pull_requests

   and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ poetry run pytest tests/
```

5.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bump2version patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER

SIX

CREDITS

6.1 Development Lead

- Matthew Schinckel <matt@schinckel.net>

6.2 Contributors

- Jonathan <jonathan.clarke@fivesigma.co.uk>

CHAPTER

SEVEN

HISTORY

7.1 0.8.0 (2023-08-11)

- Support Python 3.11
- Run CI on GitHub Actions

7.2 0.7.0 (2023-08-10)

- Call convert_value on underlying expression if available.

7.3 0.5.4 (2022-09-22)

- Allow using a shared_property as a target of an OuterRef in a Subquery/Exists.

7.4 0.5.3 (2022-09-15)

- Really support abstract models - actually querying was not working.

7.5 0.5.2 (2022-09-09)

- Support using a shared_property on an abstract base model.

7.6 0.5.1 (2022-09-05)

- Bugfix for supporting chained lookups.

7.7 0.5.0 (2022-09-03)

- Support queries over joined models.

7.8 0.4.0 (2022-08-30)

- Support Django 4.1
- Handle writing values to shared_property fields (ie, refetch from db, etc).

7.9 0.3.0 (2022-03-18)

- Support Django 4.0
- Implement Coalesce and CombinedExpression
- Support pluggable handlers.

7.10 0.2.6 (2021-07-29)

- Remove unused dependency.

7.11 0.2.5 (2021-07-26)

- Fixed bug with complex queries.

7.12 0.2.4 (2021-07-23)

- Fix bug with output_field

7.13 0.2.3 (2021-07-23)

- Ensure tables are referenced.

7.14 0.2.2 (2021-07-22)

- Remove debugging statements

7.15 0.2.1 (2021-07-22)

- Fix to allow installing

7.16 0.2.0 (2021-07-22)

- Support Enum return values.
- Simplify decorator code.

7.17 0.1.0 (2020-09-15)

- First release on PyPI.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX