django_dbcache_fields Documentation

Release 0.9.1

Joeri Bekker

Nov 01, 2017

Contents

1 About **3**

2 Installation **5**

3 Usage **7**

4 Example **9**

5 Contents **11**

5.1 Usage examples 11
5.2 Copyright 14
5.3 API Reference 14
5.4 Change history 15

Python Module Index **17**

Version 0.9.1

Docs https://django-dbcache-fields.readthedocs.io/

Download https://pypi.python.org/pypi/django_dbcache_fields

Source https://github.com/maykinmedia/django-dbcache-fields

Keywords django, database, cache, methods, decorator

CHAPTER 1

About

This library provides a decorator `dbcache` that caches the result of your Django `Model` methods in your database.

It adds a regular `Field` on your `Model` for each method that you decorate. This means you can use all ORM-functions like aggregation and migrations. You can use existing fields or let `dbcache` create the field for you.

You can also invalidate the cached value by creating a _dirty_ function or by indicating which other models affect the this cached value. By default, the cached value is only updated when the model is saved.

CHAPTER 2

Installation

You can install *django_dbcache_fields* either via the Python Package Index (PyPI) or from source.

To install using *pip*:

```
$ pip install -U django_dbcache_fields
```

CHAPTER 3

Usage

To use this with your project you need to follow these steps:

1. Install the django_dbcache_fields library:

```
$ pip install django_dbcache_fields
```

2. Add `django_dbcache_fields` to `INSTALLED_APPS` in your Django project's `settings.py`:

```
INSTALLED_APPS = (
    # ...,
    'django_dbcache_fields',
)
```

   Note that there is no dash in the module name, only underscores.

3. All done. You can now decorate methods in your `Model` with `@dbcache`.

CHAPTER 4

Example

Simple example to show what `dbcache` does:

```
from django.db import models
from django_dbcache_fields.decorators import dbcache

class Ingredient(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=4, decimal_places=2)

class Pizza(models.Model):
    name = models.CharField(max_length=100)
    ingredients = models.ManyToManyField(Ingredient)

    @dbcache(models.DecimalField(max_digits=6, decimal_places=2,
             blank=True, null=True), invalidated_by=['myapp.Ingredient'])
    def get_price(self):
        return self.ingredients.aggregate(total=Sum('price'))['total'] or Decimal()
```

Every call to `get_price` would normally perform a database query to calculate the total price of all ingredients. However, the `dbcache` decorator caused a new field to be added to the model: A `DecimalField` that can store the resulting value of the `get_price` function, so it doesn't need to perform the same query over and over again.

CHAPTER 5

Contents

5.1 Usage examples

Consider the following Django *Model* for a pizza:

```
from decimal import Decimal
from django.db import models
from django_dbcache_fields.decorators import dbcache

class Pizza(models.Model):
    TYPES = (
        ('regular', 'regular'),
        ('calzone', 'calzone'),
    )
    name = models.CharField(max_length=100)
    pizza_type = models.CharField(max_length=10, choices=TYPES)
    base_price = models.DecimalField(max_digits=4, decimal_places=2)

def get_total_price(self):
    supplement = Decimal()
    if self.pizza_type == 'calzone':
        supplement = Decimal(1)

    return self.base_price + supplement
```

The *get_total_price()* method calculated the final price simply by returning the *base_price* with a supplement for calzone pizza's.

It's not very exciting nor very computational complex but for the sake of simplicity, this is our use case.

Every call to *get_total_price()* performs the same calculation over and over again. Also, you can not simply order the list of pizza's by their total price (without annotations and conditional expressions) on database level.

5.1.1 Basic usage

Let's decorate the *get_total_price()* function with *dbcache*. To cache the total price, we need to indicate what type of Django *Field* will be used.

We assume the total price will never exceeds 6 digits, so we'll use a Django *DecimalField* with the appropriate parameters. The field should always be allowed to be *null*. For good measure it can also be *blank* in case you add the field in the Django admin.

```
class Pizza(models.Model):
    # ...
    @dbcache(models.DecimalField(max_digits=6, decimal_places=2,
             blank=True, null=True))
    def get_total_price(self):
        # ...
```

The total price will now be stored in the database. Under the hood, a new field was added to the *Pizza* model: *_get_total_price_cached*.

So, our *get_total_price()* function works as it normally would:

```
>>> pizza = Pizza.objects.create(
...     name='margarita', base_price=Decimal(10), pizza_type='calzone')
>>> pizza.get_total_price()
Decimal('11')
```

In addition, this resulting value is cached on the model field created by the *dbcache* decorator, which also allows us to perform ORM-operations with it.

```
>>> pizza._get_total_price_cached
Decimal('11')
>>> Pizza.objects.filter(_get_total_price_cached__gte=Decimal(10))
<QuerySet [<Pizza: Pizza object>]>
```

The cached field is updated everytime a new instance is created or when the instance is saved.

If the cached value is *None*, it's considered to be invalid. As a consequence, calling the *get_total_price()* method will perform its calculations as it normally would **and** will update the cached value in the database (using an update query, so it does not trigger a save signal).

5.1.2 More precise cache invalidation

In its simplest use case, *dbcache* updates the cached value everytime the instance is saved or when the decorated function is called and the cached value is *None*.

Changing the name of a *Pizza* however, will not cause any change to the total price. For this use case, we can pass a function to the *dbcache* decorator to indicate whether a change to the instance caused a price change.

Such a *dirty* function can be a very simple function or lambda, and should accept 2 arguments: The *instance* and the *field_name* of the cached field.

```
def is_pizza_price_changed(instance, field_name):
    return instance._original_base_price != instance.base_price

class Pizza(models.Model):
    def __init__(self, *args, **kwargs)
        super(Pizza, self).__init__(*args, **kwargs)
        # Store the original base price on the instance.
```

```
        self._original_base_price = self.base_price
    # ...
    @dbcache(models.DecimalField(max_digits=6, decimal_places=2,
             blank=True, null=True), dirty_func=is_pizza_price_changed)
    def get_total_price(self):
        # ...
```

The function *is_pizza_price_changed(...)* is passed to the *dirty_func* parameter of the *dbcache* decorator. This causes the following behaviour:

```
>>> pizza.name = 'hawaii'
>>> pizza.save()  # The cached field will not be updated
>>> pizza.get_total_price()
Decimal('11')

>>> pizza.base_price = Decimal(5)
>>> pizza.save()  # The cached field will be updated
>>> pizza.get_total_price()
Decimal('6')

>>> pizza.pizza_type = 'regular'
>>> pizza.save()  # The cached field will not be updated
>>> pizza.get_total_price()
Decimal('11')
```

Note that in the last example, the total price is **not** correct. The cached value was not invalidated due to an incomplete *dirty* function. The *dirty* function should have taken the *pizza_type* into account as well since it can affect the total price.

5.1.3 Methods that depend on other models

Consider this slightly altered version of our *Pizza* model. The *pizza_type* is no longer a choice field but instead a related model: *PizzaType*.

```
from django.db import models
from django_dbcache_fields.decorators import dbcache

class PizzaType(models.Model):
    name = models.CharField(max_length=100)
    supplement = models.DecimalField(max_digits=4, decimal_places=2)

class Pizza(models.Model):
    name = models.CharField(max_length=100)
    base_price = models.DecimalField(max_digits=4, decimal_places=2)
    pizza_type = models.ForeignKey(PizzaType)

    @dbcache(models.DecimalField(max_digits=6, decimal_places=2,
            blank=True, null=True), invalidated_by=['myapp.PizzaType'])
    def get_total_price(self):
        return self.base_price + self.pizza_type.supplement
```

The function *PizzaType* is passed to the *invalidated_by* parameter of the *dbcache* decorator. Any update to a *PizzaType* will cause all cached *get_total_price* values to be invalidated.

On the next call of *get_total_price()*, the invalidated cached value will be updated for this *Pizza* instance. Any save on the instance, would cause the same update.

Caveat

It's worth noting that the value of the *dbcache* generated field can always be *None*. Be careful when using ORM-functions that rely on a filled value.

Also, a *QuerySet.update()* does not trigger cached field invalidation. In the above example *PizzaType.objects.update(supplement=Decimal())* will result in incorrect total prices for pizza's.

5.2 Copyright

django-dbcache-fields User Manual

by Joeri Bekker

Copyright © 2017, Maykin Media

All rights reserved. This material may be copied or distributed only subject to the terms and conditions set forth in the Creative Commons Attribution-ShareAlike 4.0 International license.

You may share and adapt the material, even for commercial purposes, but you must give the original author credit. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same license or a license compatible to this one.

Note: While the django_dbcache_fields *documentation* is offered under the Creative Commons *Attribution-ShareAlike 4.0 International* license the django_dbcache_fields *software* is offered under the BSD License (3 Clause)

5.3 API Reference

Release 0.9

Date Nov 01, 2017

5.3.1 `django_dbcache_fields.decorators`

5.3.2 `django_dbcache_fields.receivers`

5.3.3 `django_dbcache_fields.utils`

class `django_dbcache_fields.utils.`**`Register`**
Central register to keep track of all *dbcache* decorated methods.

`get`(*class_path*)
Returns a *list* of information about *dbcache* decorated methods.

Parameters **`class_path`** – The *Model* class path.

Returns

A *list* of *dict* with the following keys:

- decorated_method
- field

- field_name
- dirty_func
- invalidated_by

get_related_models(*model*)
Returns a *dict* of models related to the *dbcache* decorated method. Typically used to see which models and fields should be invalidated when a related model is changed

Parameters **model** – The model name in the form *{app label}.{model name}*.

Returns A *dict* where each key is the affected model class path. The value is a *list* of field names that are affected.

django_dbcache_fields.utils.**get_class_path**(*instance*)
Converts an instance or *class* to a class path string.

Parameters **instance** – An instance or class.

Returns The stringified class path.

django_dbcache_fields.utils.**get_model_name**(*instance*)
Converts a *Model* instance or *class* to a model name.

Parameters **instance** – A *Model* instance or *class*.

Returns The stringified model name in the form *{app label}.{model name}*.

5.4 Change history

5.4.1 0.9.2

Unreleased

5.4.2 0.9.1

November 1, 2017

- Added documentation.
- Added more tests.
- Expose *dbcache* via *django_dbcache_fields.decorators*.

5.4.3 0.9.0

October 30, 2017

- Initial public release on PyPI.

Python Module Index

d

django_dbcache_fields.utils, 14

Index

D

django_dbcache_fields.utils (module), 14

G

get() (django_dbcache_fields.utils.Register method), 14
get_class_path() (in module django_dbcache_fields.utils), 15
get_model_name() (in module django_dbcache_fields.utils), 15
get_related_models() (django_dbcache_fields.utils.Register method), 15

R

Register (class in django_dbcache_fields.utils), 14