autocache Documentation

Release 1

Nathaniel Tucker

November 26, 2015

Contents

Table of Contents:

CHAPTER 1

Introduction

CacheMagic addresses the most common scenarios for caching and cache invalidation: *temporal caching*, *instance caching*, *related objects caching*, and *thundering herd* protection.

1.1 Temporal Caching

This is like memoizing an expensive operation. Usually something that doesn't need to be super-fresh, and aggregates many objects - making invalidation difficult or impossible.:

```
@cached
def my_expensive_call(arg1, arg2):
    return do_other_things(arg1) * arg2
```

1.2 Instance Caching

This is the practice of caching individual model instances. CacheMagic provides a *CacheController* that you can attach to models to cause automatic caching and invalidations.

```
class Model(django.models.Model):
    cache = cachemagic.CacheController()
    field = django.models.TextField()

Model.objects.get(pk=27)     # hits the database
Model.cache.get(27)          # Tries cache first
```

1.3 Related Objects Caching

Having fetched an instance of a model, a frequent database operation is to find all the instances of another model that are related to your instance via foreign keys. You can attach a *RelatedCacheController* to your model to enable automatic caching and invalidation of these relations.

```
instance = Model.cache.get(pk=27)
related_things = instance.things_set.all()  # hits the database
related_things = instance.cache.things_set  # Tries cache first
```

The RelatedCacheController will automatically detect and cache objects related to the model it resides on by ForeignKeys, ManyToManyFields, and OneToOneFields.

1.4 Thundering Herd Protection

Any caching will be useless when a cache key expires and thousands of requests try to recompute the value at the same time. CacheMagic provides a cache backend for redis that prevents this problem by designating only one client to recompute the value while others simply read the existing cache value.

```
CACHES['default'] = {
    'BACKEND': 'cachemagic.cache.RedisHerdCache',
    'LOCATION': ':'.join([REDIS_HOST, str(REDIS_PORT), '0']),
    'OPTIONS': {
        'PASSWORD': REDIS_PASSWORD,
    },
}
```

CHAPTER 2

Examples

The example model defines a person with a name.

```
class Person(models.Model):
    name = models.CharField(max_length=64)

    cache = RelatedCacheController()

    def __unicode__(self):
        return self.name

class Book(models.Model):
    title = models.CharField(max_length=64)
    author = models.ForeignKey(Person)
```

CHAPTER 3

Temporal Caching

Cached is a simple decorator that can be applied to any function to cache results. By default it uses the arguments as a key, but both the key and timeout can be customized.:

```
from django.db import models
from cachemagic.decorators import cached

@cached
def do_expensive_operation(thing, other):
    return [other(item) for item in MyModel.objects.where(a=thing)]
```

3.1 Using in model methods

In most cases you should use an object's primary key as the cache key instead of serializing the entire object.:

```
from django.db import models
from cachemagic.decorators import cached

class Model(models.Model):
    field1 = IntegerField()
    field2 = TextField()

    @cached(key=lambda self: self.pk, timeout=180)
    def get_my_related_things(self):
        return [other(item) for item in self.related_things.select_related()]
```

CHAPTER 4

Instance Caching

The CacheController works by listening for the post_save and post_delete signals that the model it is attached to will emit when you alter an instance. This allows it to automatically keep cached instances up to date!

```
from django.db import models
from cachemagic.controller import CacheController

class Model(models.Model):
    field1 = IntegerField()
    field2 = TextField()

    cache = CacheController()
```

4.1 Reading From Cache

You can use the cache controller like a very simple manager: currently only the `.get(pk)` operation is supported. This will try to get and return the model instance from cache. In the event that the key does not have a cache entry, the value is read from the database using the model's default manager. The result is placed into the cache before being returned to the caller.

```
obj = Model.cache.get(pk=933)
```

Just like objects.get(), cache.get() may raise a Model.DoesNotExist exception. A DoesNotExist marker is placed in cache when an instance is deleted or an attempt to fetch a non-existent row is made, preventing subsequent requests against the cache from hitting DB or returning stale data.

4.2 Instance Cache Keys

The default CacheController creates keys based on your model's app, name and primary key, separated by colons: `app_label:model_name:primary_key`. This should present you with a unique key for each object.

Note: This can be problematic if your model uses a primary key that can contain whitespace and you are using memcached as your cache backend. One possible solution is to provide a key generation function that hashes the key (see example below). You can also use a cache backend like Django NewCache that automatically hashes the key.

4.2.1 Overriding Cache Key Generation

You can subclass CacheController and override the make_key function to customize your cache keys.

CacheController.make_key(*self, pk*) Called to generate all cache keys for this controller. You can access the model class that this controller is attached to through `self.model`.

Examples

```
import hashlib

class HashCacheController(CacheController):
    """ Hashes the cache key. This creates keys that are difficult to type
        by hand, but can avoid problems related to key content and length.
    """
    def make_key(self, pk):
        key = super(HashCacheController, self).make_key(pk)
        return hashlib.sha256(key).hexdigest()

class ModelVersionCacheController(CacheController):
    """ Versions each cache key with the model's CACHE_VERSION attribute.
        Updating the model's version when altering it's schema will
        effectively invalidate all cached instances.
    """
    def make_key(self, pk):
        model_version = getattr(self.model, 'CACHE_VERSION', 0)
        key = ':'.join([super(HashCacheController, self), model_version])
        return key
```

4.3 Cache Timeouts

The default cache timeout is one hour. You can specify a number of seconds to timeout as the `timeout` parameter in the CacheController constructor. :

```
cache = CacheController(timeout=(60 * 60 * 24 * 7)) # timeout in one week
```

4.3.1 Overriding the default timeout

If you find yourself frequently overriding the default timeout, you can subclass the CacheController and set a `DEFAULT_TIMEOUT` attribute:

```
class LongCacheController(CacheController):
    # timeouts longer than 30 days are treated as absolute timestamps by
    # memcached; that makes 30 days the largest naive value we can use.
    DEFAULT_TIMEOUT = 60 * 60 * 24 * 30
```

4.3.2 Multicache

Starting in Django 1.3 you could define multiple cache backends. If you want to tie the instance cache for a model to a backend other than 'default', you can pass the name of the backend you want to use into the controller constructor as the keyword argument `backend`.

4.4 Caveats

CacheMagic relies on the post_save and post_delete signals to keep your cache up to date. Performing operations that alter the database state without sending these signals will result in your cache becoming out of sync with your database.

Note: Do not use queryset.update() with models that have a CacheController attached! Your cache will **not** be updated.

CHAPTER 5

Related Object Caching

5.1 Introduction

Given a model instance, one frequent database query is to get instances of another model that are related. This is commonly accomplished with the use of a Foreign Key.

We will be using the following models as examples throughout this document:

```
from django.db import models
from cachemagic.controllers import RelatedCacheController

class Person(models.Model):
    name = models.CharField(max_length=200)

    cache = RelatedCacheController()

class Book(models.Model):
    author = models.ForeignKey(Person, related_name='books')
    title = models.CharField(max_length=200)
    published_date = models.DateTimeField()

    class Meta:
        default_ordering = ('-published_date')
```

Suppose you have an authorship view, displaying all of the books that a given author has published. The view would typically look something like this:

```
def authorship(request, author_id):
    try:
        author = Person.objects.get(pk=author_id)
    except Person.DoesNotExist:
        raise Http404("No such person")
    books = author.books.all()
    return render(
        request,
        {'author': author, 'books': books},
        'authorship.html'
    )
```

This pattern will invoke two database queries: one to fetch a Person, and one to fetch the books with a foreign key relationship to the author. We can use the cachemagic features to try the cache first.

```
author = Person.objects.get(pk=author_id)    # database query
author = Person.cache.get(pk=author_id)      # cached query

books = author.books.all()                   # database query
books = author.cache.books                   # cached query
```

5.2 Reading From Cache

Given an instance of an object with a RelatedCacheController, all of the attributes on the instance to fetch related objects are mirrored on the controller. If the instance has a `.thing_set` and a RelatedCacheManager assigned to `cache`, then `instance.cache.thing_set` will return the same values as `list(instance.thing_set.all())`.

Note: Related object caches return lists of instances, not querysets. This means that you don't need to put the .all() on the end, but also that you can not apply django queryset operations like `.filter()` or `.select_related()` on the result.

5.3 Cache Keys

A cache key for the instance is obtained by calling the same `make_key(pk)` function described in *Instance Cache Keys*. The key for the related objects is the instance key, appended with the related name of the collection.

```
author = Person.objects.get(pk=1)    # get an instance of a Person in the sample_app
author.cache.books                   # cache key is sample_app:Person:1:books
```

5.4 Cache Timeouts and Multicache

The RelatedCacheController accepts the same *timeout* and *backend* arguments as CacheController.

```
cache = RelatedCacheController(
        timeout=(60 * 60 * 24 * 7),      # timeout in one week
        backend='my_app_cache'),         # use the cache backend named
                                         # 'my_app_cache' in settings.py
    )
```

CHAPTER 6

Thundering Herd Protection

When your cache keys expire, there is a time window before the new value is recomputed where many clients will not be able to retrieve any result. This will cause a huge load to database backends. More can be found here: http://en.wikipedia.org/wiki/Thundering_herd_problem

To eliminate the problem, CacheMagic provides a redis backend that stores extra metadata about expiry time. This allows one client to realize the key will expire soon and tell the others continue using the old value until it is recomputed.

6.1 Usage

To setup the protection, simply use the RedisHerdCache backend provided. Here is an example configuration:

```
CACHES['default'] = {
    'BACKEND': 'cachemagic.cache.RedisHerdCache',
    'LOCATION': ':'.join([REDIS_HOST, str(REDIS_PORT), '0']),
    'OPTIONS': {
        'PASSWORD': REDIS_PASSWORD,
    },
    'VERSION': 0,
}
```