Cache Machine

Release 0.9

July 29, 2015

Contents

Cache Machine provides automatic caching and invalidation for Django models through the ORM. The code is hosted on github.

For an overview of new features and backwards-incompatible changes which may affect you, please see the release-notes.

CHAPTER 1

Settings

Before we start, you'll have to update your `settings.py` to use one of the caching backends provided by Cache Machine. Prior to Django 1.6, Django's built-in caching backends did not allow for infinite cache timeouts, which are critical for doing invalidation (see below). Cache Machine extends the `locmem` and `memcached` backends provided by Django to enable indefinite caching when a timeout of `caching.base.FOREVER` is passed. If you were already using one of these backends, you can probably go on using them just as you were.

With Django 1.4 or higher, you should use the `CACHES` setting:

```
CACHES = {
    'default': {
        'BACKEND': 'caching.backends.memcached.MemcachedCache',
        'LOCATION': [
            'server-1:11211',
            'server-2:11211',
        ],
        'KEY_PREFIX': 'weee:',
    },
}
```

Note that we have to specify the class, not the module, for the `BACKEND` property, and that the `KEY_PREFIX` is optional. The `LOCATION` may be a string, instead of a list, if you only have one server.

If you require the default cache backend to be a different type of cache backend or want Cache Machine to use specific cache server options simply define a separate `cache_machine` entry for the `CACHES` setting, e.g.:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': 'server-1:11211',
    },
    'cache_machine': {
        'BACKEND': 'caching.backends.memcached.MemcachedCache',
        'LOCATION': [
            'server-1:11211',
            'server-2:11211',
        ],
        'KEY_PREFIX': 'weee:',
    },
}
```

Note: Cache Machine also supports the other memcache backend support by Django >= 1.4 based on pylibmc: `caching.backends.memcached.PyLibMCCache`.

1.1 COUNT queries

Calls to `QuerySet.count()` can be cached, but they cannot be reliably invalidated. Cache Machine would have to do a full select to figure out the object keys, which is probably much more data than you want to pull. I recommend a short cache timeout; long enough to avoid repetitive queries, but short enough that stale counts won't be a big deal.

```
CACHE_COUNT_TIMEOUT = 60  # seconds, not too long.
```

By default, calls to `QuerySet.count()` are not cached. They are only cached if `CACHE_COUNT_TIMEOUT` is set to a value other than `caching.base.NO_CACHE`.

1.2 Empty querysets

By default cache machine will not cache empty querysets. To cache them:

```
CACHE_EMPTY_QUERYSETS = True
```

1.3 Object creation

By default Cache Machine does not invalidate queries when a new object is created, because it can be expensive to maintain a flush list of all the queries associated with a given table and cause significant disruption on high-volume sites when *all* the queries for a particular model are invalidated at once. If these are not issues for your site and immediate inclusion of created objects in previously cached queries is desired, you can enable this feature as follows:

```
CACHE_INVALIDATE_ON_CREATE = 'whole-model'
```

CHAPTER 2

Cache Manager

To enable caching for a model, add the `CachingManager` to that class and inherit from the `CachingMixin`. If you want related lookups (foreign keys) to hit the cache, `CachingManager` must be the default manager. If you have multiple managers that should be cached, return a *`CachingQuerySet`* from the other manager's `get_queryset` method instead of subclassing `CachingManager`, since that would hook up the post_save and post_delete signals multiple times.

Here's what a minimal cached model looks like:

```
from django.db import models

from caching.base import CachingManager, CachingMixin

class Zomg(CachingMixin, models.Model):
    val = models.IntegerField()

    objects = CachingManager()
```

Whenever you run a query, `CachingQuerySet` will try to find that query in the cache. Queries are keyed by `{prefix}:{sql}`. If it's there, we return the cached result set and everyone is happy. If the query isn't in the cache, the normal codepath to run a database query is executed. As the objects in the result set are iterated over, they are added to a list that will get cached once iteration is done.

Note: Nothing will be cached if the QuerySet is not iterated through completely.

Caching is supported for normal `QuerySets` and for `django.db.models.Manager.raw()`. At this time, caching has not been implemented for `QuerySet.values` or `QuerySet.values_list`.

To support easy cache invalidation, we use "flush lists" to mark the cached queries an object belongs to. That way, all queries where an object was found will be invalidated when that object changes. Flush lists map an object key to a list of query keys.

When an object is saved or deleted, all query keys in its flush list will be deleted. In addition, the flush lists of its foreign key relations will be cleared. To avoid stale foreign key relations, any cached objects will be flushed when the object their foreign key points to is invalidated.

During cache invalidation, we explicitly set a None value instead of just deleting so we don't have any race condtions where:

- Thread 1 -> Cache miss, get object from DB
- Thread 2 -> Object saved, deleted from cache
- Thread 1 -> Store (stale) object fetched from DB in cache

The foundations of this module were derived from Mike Malone's django-caching.

2.1 Changing the timeout of a CachingQuerySet instance

By default, the timeout for a `CachingQuerySet` instance will be the timeout of the underlying cache being used by Cache Machine. To change the timeout of a `CachingQuerySet` instance, you can assign a different value to the `timeout` attribute which represents the number of seconds to cache for

For example:

```
def get_objects(name):
    qs = CachedClass.objects.filter(name=name)
    qs.timeout = 5  # seconds
    return qs
```

To disable caching for a particular `CachingQuerySet` instance, set the `timeout` attribute to `caching.base.NO_CACHE`.

CHAPTER 3

Manual Caching

Some things can be cached better outside of the ORM, so Cache Machine provides the function `caching.base.cached()` for caching arbitrary objects. Using this function gives you more control over what gets cached, and for how long, while abstracting a few repetitive elements.

CHAPTER 4

Template Caching

Cache Machine includes a Jinja2 extension to cache template fragments based on a queryset or cache-aware object. These fragments will get invalidated on using the same rules as `CachingQuerySets`.

First, add it to your template environment:

```
env = jinja2.Environment(extensions=['caching.ext.cache'])
```

Now wrap all your queryset looping with the `cache` tag.

```
{% cache objects %}  {# objects is a CachingQuerySet #}
  {% for obj in objects %}
    ...
  {% endfor %}
{% endcache %}
```

...and for caching by single objects:

```
{% cache object %}
  ...expensive processing...
{% endcache %}
```

The tag can take an optional timeout.

```
{% cache objects, 500 %}
```

If someone wants to write a template tag for Django templates, I'd love to add it.

CHAPTER 5

Redis Support

Cache Machine support storing flush lists in Redis rather than memcached, which is more efficient because Redis can manipulate the lists on the server side rather than having to tranfer the entire list back and forth for each modification.

To enable Redis support for Cache Machine, add the following to your settings file, replacing `localhost` with the hostname of your Redis server:

```
CACHE_MACHINE_USE_REDIS = True
REDIS_BACKEND = 'redis://localhost:6379'
```

Note: When using Redis, memcached is still used for caching model objects, i.e., only the flush lists are stored in Redis. You still need to configure `CACHES` the way you would normally for Cache Machine.

CHAPTER 6

Classes That May Interest You

class caching.base.**CachingQuerySet**

Overrides the default QuerySet to fetch objects from cache before hitting the database.

C