django-fancy-cache Documentation

Release 0.3

Peter Bengtsson

Sep 27, 2017

Contents

Contents:

CHAPTER 1

Getting started

The minimal you need to do is install `django-fancy-cache` and add it to a view.

Installing is easy:

```
$ pip install django-fancy-cache
```

Now, let's assume you have a `views.py` that looks like this:

```
from django.shortcuts import render

def my_view(request):
    something_really_slow...
    return render(request, 'template.html')
```

What you add is this:

```
from django.shortcuts import render
from fancy_cache import cache_page

@cache_page(3600)
def my_view(request):
    something_really_slow...
    return render(request, 'template.html')
```

Getting fancy

The above stuff isn't particularly fancy. The next steps is to start *Getting fancy*.

CHAPTER 2

Getting fancy

Invalidation based on key prefix

That *Getting started* stuff is basically what you get from standard Django. Now let's get a bit more fancy. For starters you want to write some very specific code to effectively control the invalidation.

Suppose you have a view that represents a blog post. The blog post contains a list of comments. You want the page cached but as soon as a new comment is added, the page should be cleared from cache.

Step one, add the prefixing:

```
def latest_comment_prefix(request, pk):
    try:
        latest, = (
            Comment.objects.filter(post_id=request.pk)
            .order_by('-date')[:1]
        )
        return latest.date.strftime('%f')  # microsecond
    except ValueError:
        return 'no-comments'

@cache_page(3600, key_prefix=latest_comment_prefix)
def blog_post(request, pk):
    post = Post.objects.get(pk=pk)
    comments = Comment.objects.all().order_by('date')
    return render(request, 'post.html',
                  {'comments': comments, 'post': post})
```

That means that the page will be cached nicely based on the latest comment. If a new comment is added, it's going to have a different timestamp and thus the blog post view cache will be refreshed.

However, this means that if no new comments are added for a while, we have to do that look-up each and every time. Better cache that too. Here's an updated version:

```
from django.core.cache import cache

def latest_comment_prefix(request, pk):
    cache_key = 'latest-comment-for-%s' % pk
    timestamp = cache.get(cache_key)
    if timestamp is None:
        try:
            latest, = (
                Comment.objects
                .filter(post_id=request.pk)
                .order_by('-date')[:1]
            )
            timestamp = latest.date.strftime('%f')
        except ValueError:
            timestamp = 'no-comments'
        cache.set(cache_key, timestamp, 3600)
    return timestamp
```

So we're looking up what the latest comment is and caching it for the same amount of time as we cache the view. That means that the least effort you need to return fresh view is to do 1 cache look-up which is very fast.

But now you're committed to this timestamp value for one hour (aka. 3,600 seconds) which isn't cool. You want a fresh view immediately after a new comment is added. Waiting one hour isn't good enough. You have a choice here. You can either hook in some code in where new comments are created in another view or you can use signals. Like this:

```
# now in models.py

from django.db.models.signals import post_save
from django.dispatch import receiver
from django.core.cache import cache

@receiver(post_save, sender=Comment)
def invalidate_latest_comment_timestamp(sender, instance, **kwargs):
    pk = instance.post.pk
    cache_key = 'latest-comment-for-%s' % pk
    cache.delete(cache_key)
```

Now you get the ideal set up you can have. Almost. All it takes to render the blog post page is 1 quick cache look-up. And as soon as a new comment is posted, the page is immediately refreshed.

As soon as you've implemented something like this and you get confident that your `key_prefix` callable is good you can start increasing the timeout from one hour to one week or something like that.

Another realistic use case of this is if your logged-in users get different content but whatever it is, it can be cached. Here's an example of that:

```
def dashboard_prefix(request):
    # it depends on the user
    if not request.user.is_active:
        # no cache key applicable
        return None
    return str(request.user.pk)

@cache_page(60 * 60 * 24, key_prefix=dashboard_prefix)
def users_dashboard(request):
    if not request.user.is_active:
        raise FuckOff()
```

```
    stuff = get_interesting_stuff(request.user)
    return render(request, 'dashboard.html', {'stuff': stuff})
```

Remembering what's cached

Writing an advanced `key_prefix` callable gives you a lot of flexibility. You site is probably filled with lots of interesting edge cases and other pieces of data that affect what it means for a page to be fresh and when it needs to be invalidated.

A simpler approach is to let `django-fancy-cache` keep track of *all* its internal cache keys so that you can reverse that based on a URL.

Again, let's imagine you have a blog post that shows the latest comments. As soon as a new comment is added you want the blog post to refresh. The first thing you need to do is make all cached URLs remembered. Add this to your settings:

```
FANCY_REMEMBER_ALL_URLS = True
```

Now, every time `django-fancy-cache` wraps and caches a view it remembers what the URL was and what cache key it led to. Now, let's add a signal that uses this information to invalidate the cache when a new comment is added:

```
# in models.py

from django.db.models.signals import post_save
from django.dispatch import receiver
from django.core.urlresolvers import reverse
from fancy_cache.memory import find_urls

@receiver(post_save, sender=Comment)
def invalidate_latest_comment_timestamp(sender, instance, **kwargs):
    post_pk = instance.post.pk
    post_url = reverse('blog:post', args=(post_pk,))
    list(find_urls([post_url], purge=True))
```

Note: Since find_urls() returns a generator, the purging won't happen unless you exhaust the generator. E.g. looping over it or turning it into a list.

Voila! As soon as a new comment is added to a post, all cached URLs with that URL are purged from the cache.

The page is now aggressively cached meaning you're ready for the next Hacker News stampeding herd, and as soon as a new comment it added the page is refreshed automatically so people don't get stale content.

Do some last minute fixes to the output

Suppose that you want to cache the view rendering but only apply certain cool optimizations to the rendered output just before the rendered output is put into the cache. Or perhaps you have some piece of code that you can't easily weave into your view code.

What you can do, is some "last minute" changes to the response with a callable function like this:

```
from django.shortcuts import render
from fancy_cache import cache_page

def css_stats(response, request):
```

```
    no_styles = response.content.count('</style>')
    response.content += (
        '\n<!--\n'
        'This page has %s style tags\n'
        % no_styles
    )
    return response

@cache_page(3600, post_process_response=css_stats)
def my_view(request):
    something_really_slow...
    return render(request, 'template.html')
```

The example is a bit silly but it demonstrates what you can do with the `post_process_response` parameter.

Post process the response on every request

Suppose you have one of those sites where the content is identical to all anonymous users and or all logged in users. I.e. the content looks the same not matter if you're logged in as Tom, Dick or Harry. Just as long as you're logged in.

Additionally, your page is one of those that says "You're logged in as Tom" somewhere in the upper right hand corner. Then, the only thing that needs to be different is the name of the logged in user. What you can do is this. First change your template to look something like this:

```
# in dashboard.html

<html>
  <header>
  You're logged in as *LOGGED_IN_NAME*
  </header>

  <section>
    {% for thing in stuff %}
    ...
    {% endfor %}
  </section>
</html>
```

Now, make the view cache the whole content but replace the currently logged in name each time. Something like this will work:

```
def logged_in_love(response, request):
    html = response.content
    html = html.replace(
        '*LOGGED_IN_NAME*',
        request.user.first_name
    )
    response.write = html
    return response

@cache_page(60 * 60, post_process_response_always=logged_in_love)
def users_dashboard(request):
    if not request.user.is_active:
        raise FuckOff()
    stuff = get_interesting_stuff(request.user)
    return render(request, 'dashboard.html', {'stuff': stuff})
```

An alternative solution to this is to use client-side programming and render the HTML without any name and then use one snappy and simple piece of AJAX/localStorage code that updates the header accordingly.

Not all query string parameters matter

By default, the `cache_page` decorator automatically creates a cache key based on the URL and the query string. That means that `/blog/?page=2` is going to be different from `/blog/?page=3`. Makes sense. But if you enter a URL like `/blog/?page=2&other=junk` it's a new URL that means it can't use the cached content used for `/blog/?page=2` and that's not good. It means the server has to generate a new page for every piece of junk query string that is requested. Perhaps you confidently know that the only query string parameter that means something is `page` and its value.

What you can do is supply `only_get_keys` to hint that only certain GET variables matter. It looks like this:

```
from django.shortcuts import render
from fancy_cache import cache_page

@cache_page(3600, only_get_keys=['page'])
def my_view(request):
    something_really_slow...
    return render(request, 'template.html')
```

Now, if someone requests `/blog/?page=2&other=junk` they will get the exact same cached response if they request `/blog/?page=2&different=junk` or `/blog/?page=2&other=crap`. It'll all act as if the URL was `/blog/?page=2`.

Note: the order of keys does *not* matter.

Stats of hits and misses

See *Stats - hits and misses*.

CHAPTER 3

Stats - hits and misses

In *Getting fancy* we went through various ways of using `django-fancy-cache` that all have a functional difference.

What you can do is enable stats so you can get an insight into what `django-fancy-cache` does for you.

Setting up

The first step is to switch on a setting. Put this in your `settings`:

```
FANCY_REMEMBER_ALL_URLS = True
FANCY_REMEMBER_STATS_ALL_URLS = True
```

The other thing you need to do is to add `fancy_cache` to `INSTALLED_APPS` like this:

```
INSTALLED_APPS += ('fancy_cache',)
```

The first one is to tell `django-fancy-cache` to remember every URL it caches and the second one is to keep track of how many hits and misses you have per URL.

This will enable stats collections on all uses of the `cache_page` decorator. Alternatively you can do it explicitely on just one view. Like this:

```
from django.shortcuts import render
from fancy_cache import cache_page

@cache_page(3600,
            remember_all_urls=True,
            remember_stats_all_urls=True)
def my_view(request):
    something_really_slow...
    return render(request, 'template.html')
```

Now, run your view a couple of times and then you can use the management command to get an output of this:

```
$ ./manage.py fancy-urls
/page5.html                                              HITS 62    MISSES 5
/page4.html                                              HITS 4     MISSES 1
```

Another way add `django-fancy-cache` to your root urls.py like this:

```
urlpatterns = patterns(
    '',
    ...your other stuff...,
    url(r'fancy-cache', include('fancy_cache.urls')),
)
```

Now you can visit `http://localhost:8000/fancy-cache`. It'll give you a very basic table with all cache keys, their hits and misses.

CHAPTER 4

Changelog

v0.4.0 (2013-03-12)

The function `fancy_cache.memory.find_urls()` can now be called without any arguments since means it matches *all* URLs.

If you enable `FANCY_REMEMBER_STATS_ALL_URLS` and have some really big URLs (e.g. long query strings) you could get `MemcachedKeyLengthError` errors potentially.

v0.3.2 (2013-02-13)

Small improvement to the example app, documentation and a bunch of commented out debugging removed.

v0.3.1 (2013-02-11)

Docs added.

v0.3 (2013-02-11)

First time a changelog is started.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index