Cache-dependencies Documentation

Release

Ivan Zakrevsky

Oct 24, 2017

Contents:

Cache-dependencies (former Cache-tagging) allows you easily invalidate all cache records tagged with a given tag(s). Supports Django.

Tags are a way to categorize cache records. When you save a cache, you can set a list of tags to apply for this record. Then you will be able to invalidate all cache records tagged with a given tag (or tags).

Cache tagging allows to manage cached values and easily link them to Model signals.

- Home Page: https://bitbucket.org/emacsway/cache-dependencies
- Docs: https://cache-dependencies.readthedocs.io/
- Browse source code (canonical repo): https://bitbucket.org/emacsway/cache-dependencies/src
- GitHub mirror: https://github.com/emacsway/cache-dependencies
- Get source code (canonical repo): `hg clone https://bitbucket.org/emacsway/cache-dependencies`
- Get source code (mirror): `git clone https://github.com/emacsway/cache-dependencies.git`
- PyPI: https://pypi.python.org/pypi/cache-dependencies

LICENSE:

- License is BSD

Table of Contents

CHAPTER 1

Usage with Django

project settings.py:

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

project urls.py:

```
from django_cache_dependencies import autodiscover
autodiscover()
```

application example:

```
# Default backend
from django_cache_dependencies import caches
cache = caches['default']

value = cache.get('cache_name')
if value is None:
    value = get_value_func()
    cache.set('cache_name', value, tags=(
        'blog.post',
        'categories.category.pk:{0}'.format(obj.category_id),
    ))
```

manual invalidation:

```
from django_cache_dependencies import caches
cache = caches['default']

# ...
cache.invalidate_tags('tag1', 'tag2', 'tag3')
# or
```

```
tag_list = ['tag1', 'tag2', 'tag3', ]
cache.invalidate_tags(*tag_list)
```

Ancestors automatically receive tags from their descendants. You do not have to worry about how to pass the tags from fragment's caches to the composite (parent) cache. It is done automatically:

```
val1 = cache.get('name1')

if val1 is None:
    val2 = cache.get('name2')

    if val2 is None:
        val2 = get_val2()
        cache.set('name2', val2, ('tag2', ), 120)

    val1 = get_val1() + val2
    cache.set('name1', val1, ('tag1', ), 120)

cache.invalidate_tags('tag2')
assert cache.get('name2') is None
assert cache.get('name1') is None  # cache with name 'name1' was invalidated
                                   # by tag 'tag2' of descendant.
```

You can turn off this logic:

```
# turn off for cache instance
cache.ignore_descendants = True

# turn off for get template
cache.get('cachename', abort=True)

# abort cache creating
val = cache.get('cachename')
if val is None:
    try:
        val = get_val()
    except Exception:
        cache.abort('cachename')
```

appname.caches.py file:

```
# Variant 1. Using registry.register().
# Each item from list creates model's post_save and pre_delete signal.
# Func takes changed model and returns list of tags.
# When the signal is called, it gets varied tags and deletes all caches with this␣
↪tags.
# Inside the handler function available all local variables from signal.
# Or only object. Of your choice.

from django_cache_dependencies import registry, caches
from models import Post
from news import Article

cache_handlers = [
    #(model, func, [cache_alias, ]),
    (Post, lambda *a, **kw: ("blog.post.pk:{0}".format(kw['instance'].pk), ), 'my_
↪cache_alias'),
    (Article, lambda obj: (
```

```
        "news.alticle.pk:{0}".format(obj.pk),
        "categories.category.pk:{0}.blog.type.pk:{1}".format(obj.category_id, obj.
↪type_id),  # Composite tag
        "news.alticle"
    )),
]
registry.register(cache_handlers)


# Variant 2. Low-lewel. Using signals for invalidation.

from django_cache_dependencies import registry
from models import Post
from django.db.models.signals import post_save, post_delete

def invalidation_callback(sender, instance, **kwars):
    cache.invalidate_tags(
        'tag1', 'tag2', 'blog.post.pk:{1}'.format(instance.pk)
    )

post_save.connect(invalidation_callback, sender=Post)
pre_delete.connect(invalidation_callback, sender=Post)
```

template:

```
{% load cache_tagging_tags %}
{% cache_tagging 'cache_name' 'categories.category.pk:15' 'blog.post' tags=tag_list_
↪from_view timeout=3600 %}
    ...
    {% cache_add_tags 'new_tag1' %}
    ...
    {% cache_add_tags 'new_tag2' 'new_tag3' %}
    ...
    {% if do_not_cache_condition %}
        {% cache_tagging_prevent %}
    {% endif %}
{% end_cache_tagging %}
{% comment %}
    {% cache_tagging cache_name [tag1]  [tag2] ... [tags=tag_list] [timeout=3600] %}
    {% cache_add_tags tag_or_list_of_tags %}
    If context has attribute "request", then templatetag {% cache_tagging %}
    adds to request a new attribute "cache_tagging" (instance of set() object) with␣
↪all tags.
    If request already has attribute "cache_tagging", and it's instance of set()␣
↪object,
    then templatetag {% cache_tagging %} adds all tags to this object.
    You can use together templatetag {% cache_tagging %} and decorator @cache_page().
    In this case, when @cache_page() decorator will save response,
    it will also adds all tags from request.cache_tagging to cache.
    You need not worry about it.

    If need, you can prevent caching by templatetag {% cache_tagging_prevent %}.
    In this case also will be prevented @cache_page() decorator, if it's used,
    and context has attribute "request".
{% endcomment %}
```

Support for django-phased:

```
{% comment %}
    Support for django-phased https://github.com/codysoyland/django-phased
    See documentation for more details https://django-phased.readthedocs.io/
{% endcomment %}
{% load cache_tagging_tags %}
{% load phased_tags %}
{% cache_tagging 'cache_name' 'categories.category.pk:15' 'blog.post' tags=tag_list_
↪from_view timeout=3600 phased=1 %}
    ... Cached fragment here ...
    {% phased with comment_count object %}
        {# Non-cached fragment here. #}
        There are {{ comment_count }} comments for "{{ object }}".
    {% endphased %}
{% end_cache_tagging %}
```

nocache support:

```
{% load cache_tagging_tags %}
{% cache_tagging 'cache_name' 'categories.category.pk:15' 'blog.post' tags=tag_list_
↪from_view timeout=3600 nocache=1 %}
    ... Cached fragment here ...
    {% nocache %}
        """
        Non-cached fragment here. Just python code.
        Why nocache, if exists django-phased?
        Because template engine agnostic. We can use not only Django templates.
        Of course, for only Django template engine, django-phased is the best option.
        """
        if request.user.is_authenticated():
            echo('Hi, ', filters.escape(request.user.username), '!')
            echo(render_to_string('user_menu.html', context))
        else:
            echo(render_to_string('login_menu.html', context))
    {% endnocache %}
{% end_cache_tagging %}
```

view decorator:

```
from django_cache_dependencies.decorators import cache_page

# See also useful decorator to bind view's args and kwargs to request
# https://bitbucket.org/emacsway/django-ext/src/d8b55d86680e/django_ext/middleware/
↪view_args_to_request.py

@cache_page(3600, tags=lambda request: ('blog.post', ) + get_tags_for_
↪request(request))
def cached_view(request):
    result = get_result()
    return HttpResponse(result)
```

How about transaction and multithreading (multiprocessing)?:

```
from django.db import transaction
from django_cache_dependencies import cache
from django_cache_dependencies import cache_transaction

with cache.transaction, transaction.commit_on_success():
    # ... some code
```

```
    # Changes a some data
    cache.invalidate_tags('tag1', 'tag2', 'tag3')
    # ... some long code
    # Another concurrent process/thread can obtain old data at this time,
    # after changes but before commit, and create cache with old data,
    # if isolation level is not "Read uncommitted".
    # Otherwise, if isolation level is "Read uncommitted", and transaction will␣
↪rollback,
    # the concurrent and current process/thread can creates cache with dirty data.
```

Transaction handler as decorator:

```
from django.db import transaction
from django_cache_dependencies import cache
from django_cache_dependencies.decorators import cache_transaction

@cache.transaction
@transaction.commit_on_success():
def some_view(request):
    # ... some code
    cache.invalidate_tags('tag1', 'tag2', 'tag3')
    # ... some long code
    # Another concurrent process/thread can obtain old data at this time,
    # after changes but before commit, and create cache with old data,
    # if isolation level is not "Read uncommitted".
    # Otherwise, if isolation level is "Read uncommitted", and transaction will␣
↪rollback,
    # the concurrent and current process/thread can creates cache with dirty data.
    #
    # We can even invalidate cache before data changes,
    # by signals django.db.models.signals.pre_save()
    # or django.db.models.signals.pre_delete(), and don't worry.
```

Transaction handler as middleware:

```
MIDDLEWARE_CLASSES = [
    # ...
    "django_cache_dependencies.middleware.TransactionMiddleware",  # Should be before
    "django.middleware.transaction.TransactionMiddleware",
    # ...
]
```

Forked from https://github.com/Harut/django-cachecontrol

See also articles:

- “About problems of cache invalidation. Cache tagging.“
- “ . .“

CHAPTER 2

Indices and tables

- genindex
- modindex
- search