
Django Better Cache Documentation

Release 0.7.0

Calvin Spealman

February 04, 2016

1	Table of Contents	3
1.1	bettercache template tags	3
1.2	CacheModel	4
1.3	CachedMethod	5
1.4	CachedFormMethod	5
1.5	Bettercache Middleware and Celery Task	6
1.6	Bettercache Proxy Server	7
1.7	API Documentation	8
1.8	Roadmap	8
1.9	Template Tag	8
1.10	Cache ORM	9
1.11	Middleware	9
1.12	Celery Task	9
1.13	Proxy Server	9
1.14	Cache Backend	9
2	Discussion	11
3	Contributing	13
4	Indices and tables	15

Better Cache originally provided a replacement `{% cache %}` tag, but as of version 0.7 includes a Cache ORM module and a suite of caching and proxy tools. Overall, the aim is to simplify and empower your use of caches with sane defaults and obvious behaviors.

Table of Contents

1.1 bettercache template tags

Currently, the only tag provided is a replacement for Django's builtin `{% cache %}` tag, which makes it easier to work with nested blocks.

1.1.1 cache

Better Cache provides a replacement for the default cache template tag library from Django. It is a better version of `{% cache %}`.

What is better about Better Cache's version of the cache tag?

- Nested cache fragments inherit the variables their parent fragments key on
- Parent cache fragments can be given additional keys by their child cache fragments

An example:

```
{% cache 500 "outer" x %}
  y = {{ y }}<br />
  {% cache 500 "inner" y %}
    x = {{ x }}<br />
  {% endcache %}
{% endcache %}
```

In the default `{% cache %}` tag from Django, the inner fragment will not be rerendered when `x` changes, because only the outer fragment uses that as a key variable. The outer fragment will not update with `y` changes, because only the inner fragment uses that.

With Better Cache, `x` and `y` affect both, so fragments will be re-rendered when any important variable changes.

Default Keys

Better Cache also allows a syntax of giving defaults to key variables:

```
{% cache 500 "test" x=10 %}
  ...
{% endcache %}
```

This allows the block to be rendered as if `x` had the value `10`, caching the result and reusing if later if `x` really does exist and have that value later.

Controlling inheritance

You don't always want the outer cache fragments to invalidate when variables only important to the inner fragment changes. In some cases, the inner fragment is allowed to get stale if it stays cached longer as part of the parent, so we want a way to disable the inheritance of the variables.

You can do this with the *local* modifier. All modifiers after the *local* will affect only this cache fragment, not its parent.

```
{% cache 500 "outer" x %}
  y = {{ y }}<br />
  {% cache 500 "inner" local y %}
    x = {{ x }}<br />
  {% endcache %}
{% endcache %}
```

1.2 CacheModel

To make the management of cached data easier, `bettercache` provides a structured model for data caching, without the developer constantly building up ad-hoc key strings. This should be a familiar interface, fashioned after Django's own database models.

```
class User(CacheModel):
    username = Key()
    email = Field()
    full_name = Field()

user = User(
    username = 'bob',
    email = 'bob@hotmail.com',
    full_name = 'Bob T Fredrick',
)
user.save()

...

user = User.get(username='bob')
user.email == 'bob@hotmail.com'
user.full_name == 'Bob T Fredrick'
```

`CacheModel` subclasses are a collection of `Key` and `Field` properties to populate with data to be stored in the cache. The creation of keys are automatic, based on the `CacheModel` class and the values given for all the `Key` fields for an instance.

The cache objects can save any fields with JSON-serializable values, but this does not include other instances of `CacheModel`. If you'd like to connect multiple cached entities, you can do so with the field type `Reference`.

```
class Workplace(CacheModel):
    name = Key()
    phone = Field()
    address = Field()
    employee_count = Field()

class User(CacheModel):
    username = Key()
    email = Field()
    full_name = Field()
```



```
workplace = Reference(Workplace)

mother = Reference('self')
father = Reference('self')
```

Reference fields are created with a single argument: either a `CacheModel` class which the field must reference, or `'self'` to reference instances of the same class as itself.

1.3 CachedMethod

One useful `CacheModel` is included with `bettercache`, named `bettercache.decorators.CachedMethod`. This class acts as a decorator for methods, and will cache the results of those methods using a defined set of attributes from the instance. For any instance of the class with the same values for this set of attributes, the method will use the cached value properly, but also use its own parameters.

This is a decorator-factory, and it takes one required parameter and one optional.

```
@CachedMethod('attributes to cache on', expires=SECONDS)
```

```
class Home(object):

    def __init__(self, address):
        self.address = address

    @CachedMethod('address')
    def geocode(self):
        return g.geocode(self.address)
```

1.4 CachedFormMethod

An included `CachedMethod` decorator subclass which knows how to cache methods on Django forms, such that given the same form results, the methods will be cached from previous forms with the same results. This caches based on the `cleaned_data` rather than pre-validation `data`, so if your cleaning normalizes the input the caching will be more efficient.

```
class FriendsLookup(forms.Form):

    username = forms.CharField(required=True)

    @CachedFormMethod(expires=60*15) # expire in 15 minutes
    def get_friends_list(self, include_pending=False):
        username = self.cleaned_data['username']
        friends = Friendship.objects.filter(
            from_user__username=username)
        if include_pending:
            friends = friends.filter(status__in=(PENDING, APPROVED))
        else:
            friends = friends.filter(status=APPROVED)

        return friends
```

1.4.1 API Reference

CacheModel A base class you can inherit and define structures to store in the cache, much like a Django Model storing data in the database.

CacheModel.Missing An exception raised when an object cannot be found in the cache.

CacheModel.save() Sends the serialized object to the cache for storage.

CacheModel.get(key1=x, key2=y) Looks for an instance of the cache model to load and return, by the keys given. All keys defined in the model without defaults must be given.

CacheModel.from_miss(kwargs)** When you define a `CacheModel` subclass, you can opt to implement the `from_miss()` method, which will be called on an instance of your class with the keys which couldn't be found in the database.

Your `from_miss()` method should initialize the instance, after which the object will be saved to the cache and returned back from the original `get()` call in the first place.

Key At least one of your fields must be defined as a `Key`, which will be combined with the class information to generate a unique key to identify the object in the cache.

Field In your `CacheModel`, you should define one or more `Field` properties. The values of these properties in your instance will all be serialized and sent to the cache when the object is saved.

Reference If a field needs to contain other `CacheModel` instances, you may use the special field type `Reference`, which will fetch the referenced instance from the cache at load time. If any referenced fields in a model are missing, the entire model is considered invalid and a `get()` will raise a `CacheModel.Missing` exception.

PickleField Special field type which uses the python `pickle` format, rather than `JSON`, for serialization. This should only be used in special cases, as `pickle` has a number of drawbacks and corner cases.

1.5 Bettercache Middleware and Celery Task

The `bettercache` middleware is intended as a replacement for `django's` caching middleware that does offline page updateing via `celery`.

1.5.1 Bettercache Regeneration Strategy

`Bettercache` has two cache timeouts a `postcheck` and `precheck`. The `postcheck` time should be shorter then the `precheck`. If it isn't `celery` will never be used to regenerate pages. * Before the `postcheck` time the page is simply served from cache. * Between the `postcheck` and the `precheck` times `Bettercache` will serve the cached page. Then it will queue a `celery` task to regenerate the page and recache the page to reset both `postcheck` and `precheck` timeouts. * After the `precheck` time a new page will be regenerated and served.

1.5.2 When will bettercache cache a page?

`Bettercache` will cache a page under the following conditions

- `request._cache_update_cache` is not `True`.
- The status code is 200, 203, 300, 301, 404, or 410.
- The setting `BETTERCACHE_ANONYMOUS_ONLY` is not `True` or the session hasn't been accessed.
- The request does not have any uncacheable headers. To change this override `has_uncacheable_headers`.

See the [task API docs](#) for more information.

1.5.3 Bettercache header manipulation

The bettercache middleware will change some of the request headers before it caches a page for the first time.

- If `BETTERCACHE_ANONYMOUS_ONLY` is not `True` bettercache will remove `Vary: Cookie` headers.
- The `Cache-Control` headers are modified so that - `max-age` and `pre-check` set to `BETTERCACHE_CACHE_MAXAGE` unless the request already had a `max-age` header in which case that will be honored. - `post-check` is set to `BETTERCACHE_EDGE_POSTCHECK_RATIO` * the `max-age`.
- The `Edge-Control` header is set with `cache-maxage` to `BETTERCACHE_EDGE_MAXAGE`.

1.5.4 Bettercache middleware settings

The following settings are currently aspirational but the changes should be coming soon.

- `BETTERCACHE_EXTERNAL_MAXAGE` - the default external the Cache-Control max-age/pre-check headers to
- `BETTERCACHE_EXTERNAL_POSTCHECK_RATIO` - the ratio of max_age to set the Cache-Control post-check header to
- `BETTERCACHE_LOCAL_MAXAGE` - the number of seconds to cache pages for locally
- `BETTERCACHE_LOCAL_POSTCHECK` - the number of seconds after which to attempt to regenerate a page locally

See the [middleware API docs](#) for more information.

1.5.5 Bettercache middleware TODO list

- Remove akamai headers and create hooks for additional header manipulation
- Allow views to set non-default cache local_maxage/postchecks?
- Switch to better settings module
- Switch to not caching django request objects but json body/header/additional info objects

1.6 Bettercache Proxy Server

The Bettercache proxy server is intended to work with a slower django application that's using the bettercache middleware. It is threadsafe so many proxy requests can be served without loading the application that is actually generating the pages. It will take care of serving pages from a cache populated by the bettercache middleware/celery task and sending tasks to celery to regenerate those pages when necessary.

1.6.1 Settings required for proxy server

In addition to the normal settings for the bettercache middleware, celery and django the following setting is also required for the proxy server.

- `BETTERCACHE_ORIGIN_HOST` - The server which proxy traffic should be directed at. The host name from the original request will be passed on.

1.7 API Documentation

1.7.1 decorators

1.7.2 handlers

1.7.3 middleware

1.7.4 objects

1.7.5 proxy

1.7.6 tasks

1.7.7 utils

1.7.8 views

1.8 Roadmap

The next releases of `bettercache` is planned to expand upon the `CacheModel` even further, handling cache misses and allow push updates of cached data, among other new treats.

- `from_model_APP_MODEL()` methods on `CacheModel` can be implemented to update the cached data when models are updated
- Secondary key-sets, to allow more than one lookup for the same cache data
- Included Celery tasks to async update the cached data
- Two part `from_miss` with a sync step that defers the second step to Celery
- Implemented nested invalidation of `CacheModels`
- Convert the replacement `{% cache %}` tag to generate `CacheModels`
- Add a `{% notcached %}` tag to nest inside `{% cache %}` blocks
- Add an `{% else %}` clause to cache blocks
- Defer rendering of cache blocks to celery
- Push deferred-rendered cache blocks back to pages

1.9 Template Tag

The `bettercache` `cache template tag` provides some automatic invalidation.

1.10 Cache ORM

Caching can be more than a string and random object. `bettercache.objects` provides an ORM interface to structure caching and manage keys for you, replacing a mix-mash of adhoc key generation and fragile object pickling with stable cache models and key management, via the *cachemodel*.

1.11 Middleware

Bettercache *middleware* serves as an improved version of the django caching middleware allowing better control of cache headers and easier to generate cache keys.

1.12 Celery Task

The bettercache *celery task* allows most pages to be updated offline in a post check fashion. This means a user never has to wait for a slow page when serving a cached one would be acceptable.

1.13 Proxy Server

The bettercache proxy server can serve pages cached by the bettercache middleware and deal with updating via the celery task.

1.14 Cache Backend

Currently not implemented this will be a django 1.3 compatible caching backend with stampede prevention and check and set support

Discussion

You can make suggestions and seek assistance on the mailing list:

<https://groups.google.com/forum/#!forum/bettercache>

Contributing

Fork and send pull requests, please!

<http://github.com/ironfroggy/django-better-cache/>

Indices and tables

- `genindex`
- `modindex`
- `search`