Johnny Cache Documentation

Release 1.6.1a

Jason Moiron, Jeremy Self

Nov 03, 2017

Contents

Johnny Cache is a caching framework for django applications. It works with the django caching abstraction, but was developed specifically with the use of memcached in mind. Its main feature is a patch on Django's ORM that automatically caches all reads in a consistent manner. It works with Django 1.1 thru 1.4 and python 2.4 thru 2.7.

You can install johnny with pip:

```
pip install johnny-cache
```

You can fork johnny-cache from its git repository:

```
git clone https://github.com/jmoiron/johnny-cache.git
```

or, if you prefer, from its hg mirror:

```
hg clone http://bitbucket.org/jmoiron/johnny-cache
```

Please use github's issue tracker to report bugs. Contact the authors at @jmoiron and @finder83.

CHAPTER 1

Usage

A typical `settings.py` file for Django 1.3 or 1.4 configured for `johnny-cache`:

```
# add johnny's middleware
MIDDLEWARE_CLASSES = (
    'johnny.middleware.LocalStoreClearMiddleware',
    'johnny.middleware.QueryCacheMiddleware',
    # ...
)
# some johnny settings
CACHES = {
    'default' : dict(
        BACKEND = 'johnny.backends.memcached.MemcachedCache',
        LOCATION = ['127.0.0.1:11211'],
        JOHNNY_CACHE = True,
    )
}
JOHNNY_MIDDLEWARE_KEY_PREFIX='jc_myproj'
```

For a full inspection of options for earlier versions of Django please see the queryset cache docs.

The `MIDDLEWARE_CLASSES` setting enables two middlewares: the outer one clears a thread-local dict-like cache located at `johnny.cache.local` at the end of every request, and should really be the outer most middleware in your stack. The second one enables the main feature of Johnny: the queryset cache.

The `CACHES` configuration includes a custom backend, which allows cache times of "0" to be interpreted as "forever", and marks the `default` cache backend as the one Johnny will use.

Finally, the project's name is worked into the Johnny key prefix so that if other projects are run using the same cache pool, Johnny won't confuse the cache for one project with the cache for another.

With these settings, all of your ORM queries are now cached. You should read the queryset cache documentation closely to see if you are doing anything that might require manual invalidation.

Johnny does not define any views, urls, or models, so we can skip adding it to `INSTALLED_APPS`.

Note: Since Johnny is enabled by the inclusion of middleware, it will not be enabled by default in scripts, management commands, asynchronous workers, or the django shell. See the queryset cache documentation for instructions on how

to enable it in these cases.

1.1 New in this version

- Django 1.4 support
- Redis backend (requires `django-redis-cache`)
- Master/Slave support
- Cache whitelist
- New celery task utilities

1.2 Version Numbering

Because Johnny tracks Django's release schedule with its own releases, and is itself a mature project, the version number has been bumped from 0.3 to 1.4 to coincide with the highest version of Django with support. In the future, Johnny's version will track the major and minor version numbers of Django, but will have independent dot releases for bugfixes, maintenance, and backwards compatible feature enhancements.

1.3 Deprecation Policy

As of the release of Django 1.4, Django 1.1 and 1.2 are now officially unsupported projects. In addition, in an effort to clean up code in preparation for eventual Python 3.3 support, Django 1.4 drops support for Python 2.4 and Django 1.5 will drop support for Python 2.5.

Johnny 1.4 will maintain support for Django 1.1+ and Python 2.4 thru 2.7, as previous releases have had no official deprecation policies. Future versions will:

- Adopt Django's Python version support & deprecation policy (including py3k adoption)
- Support the 3 most recent versions of Django

If Django development goals are met, this means that Johnny 1.5 will support Django 1.3-1.5 and Python 2.6+, with experimental Python 3.3 support. This also means that, while future versions of Johnny will be compatible with older versions of Django, they might not be compatible with all of the supported versions of *python* for these old versions.

1.4 In Depth Documentation

1.4.1 The QuerySet Cache

QuerySet caching is the automatic caching of all database reads. Conceptually, it works very similar to the built-in write-invalidate queryset caching that is present in your RDBMS.

When a read (`SELECT`) is made from one or more tables, that read is cached. When a write (`INSERT`, `UPDATE`, etc) is made against that table, the read cache built up for that table is invalidated. The way that Johnny achieves this is through the use of generational keys:

- every table in your application gets a "key" associated with it that corresponds to the current *generation* of data
- reads are cached with keys based off of the generations of the tables being read from

- when a write is performed, the generation key for all involved tables is modified

When the generation key is modified, any cache keys previously built against prior generations are no longer recoverable, since the old generation key is now lost. This means on an LRU cache (like memcached, which maintains an LRU per slab), you can cache reads forever and the old generations will naturally expire out faster than any "live" cache data.

The QuerySet Cache supports Django versions 1.1, 1.2, 1.3, and 1.4

class `johnny.cache.`**`QueryCacheBackend`**(*cache_backend=None*, *keyhandler=None*, *keygen=None*)
This class is the engine behind the query cache. It reads the queries going through the django Query and returns from the cache using the generation keys, or on a miss from the database and caches the results. Each time a model is updated the table keys for that model are re-created, invalidating all cached querysets for that model.

There are different QueryCacheBackend's for different versions of django; call `johnny.cache.get_backend` to automatically get the proper class.

`johnny.cache.`**`get_backend`**(***kwargs*)
Get's a QueryCacheBackend object for the given options and current version of django. If no arguments are given, and a QCB has been created previously, `get_backend` returns that. Otherwise, `get_backend` will return the default backend.

NOTE: The usage of `get_backend` has changed in Johnny 0.3. The old version returned a class object, but the new version works more as a factory that gives you a properly configured QuerySetCache *object* for your Django version and current settings.

The main goals of the QuerySet Cache are:

- To cache querysets forever
- To be as simple as possible but still work
- To not increase the conceptual load on the developer

Invalidation

Because queries are cached forever, it's absolutely essential that stale data is never accessible in the cache. Since keys are never actually deleted, but merely made inaccessible by the progression of a table's generation, "invalidation" in this context is the modification of a table's generation key.

The query keys themselves are based on as many uniquely identifying aspects of a query that we could think of. This includes the sql itself, the params, the ordering clause, the database name (1.2 only), and of course the generations of all of the tables involved. The following would be two queries, not one:

```
MyModel.objects.all().order_by('-created_at')
MyModel.objects.all().order_by('created_at')
```

Avoiding the database at all costs was not a goal, so different ordering clauses on the same dataset are considered different queries. Since invalidation happens at the table level, *any* table having been modified makes the cached query inaccessible:

```
# cached, depends on `publisher` table
p = Publisher.objects.get(id=5)
# cached, depends on `book` and `publisher` table
Book.objects.all().select_related('publisher')
p.name = "Doubleday"
# write on `publisher` table, modifies publisher generation
p.save()
# the following are cache misses
```

```
Publisher.objects.get(id=5)
Book.objects.all().select_related('publisher')
```

Because invalidation is greedy in this way, it makes sense to test Johnny against your site to see if this type of caching is beneficial.

Transactions

Transactions represent an interesting problem to caches like Johnny. Because the generation keys are invalidated on write, and a transaction commit does not go down the same code path as our invalidation, there are a number of scenarios involving transactions that could cause problems.

The most obvious one is write and a read within a transaction that gets rolled back. The write invalidates the cache key, the read puts new data into the cache, but that new data never actually sees the light of day in the database. There are numerous other concurrency related issues with invalidating keys within transactions regardless of whether or not a rollback is performed, because the generational key change is in memcached and thus not protected by the transaction itself.

Because of this, when you enable Johnny, the `django.db.transaction` module is patched in various places to place new hooks around transaction rollback and committal. When you are in what django terms a "managed transaction", ie a transaction that *you* are managing manually, Johnny automatically writes any cache keys to the LocalStore instead. On commit, these keys are pushed to the global cache; on rollback, they are discarded.

Using with TransactionMiddleware (Django 1.2 and earlier)

Django ships with a middleware called `django.middleware.transaction.TransactionMiddleware`, which wraps all requests within a transaction and then rollsback when exceptions are thrown from within the view. Johnny only pushes transactional data to the cache on commit, but the TransactionMiddleware will leave transactions uncommitted if they are not dirty (if no writes have been performed during the request). This means that if you have views that don't write anything, and also use the TransactionMiddleware, you'll never populate the cache with the querysets used in those views.

This problem is described in django ticket #9964, and has been fixed as of Django 1.3. If you are using a Django version earlier than 1.3 and need to use the TransactionMiddleware, Johnny includes a middleware called `johnny.middleware.CommittingTransactionMiddleware`, which is the same as the built in version, but always commits transactions on success. Depending on your database, there are still ways to have SELECT statements modify data, but for the majority of people committing after every request, even when no UPDATE or INSERTs have been done is likely harmless and will make Johnny function much more smoothly.

Savepoints

Johnny supports savepoints, and although it has some comprehensive testing for savepoints, it is not entirely certain that they behave the same way across the two backends that support them. Savepoints are tested in single and multi-db environments, from both inside and outside the transactions.

Currently, of the backends shipped with Django only the PostgreSQL and Oracle backends support savepoints (MySQL's InnoDB engine supports savepoints, but the Django MySQL backend doesn't). If you use savepoints and are encountering invalidation issues, please report a bug and see the *Manual Invalidation* section for possible workarounds.

Multiple Databases

Johnny supports multiple databases in a variety of configurations. If using Johnny to cache results from a slave database, one should use the `DATABASES .. JOHNNY_CACHE_KEY` setting (see below) to ensure that slave databases use the same database key as the master database.

Please note that, in a master/slave database configuration, all of the typical problems still exist. For example, if you update a table and then initiate a read on the slave before the change has had time to propagate, stale data may be returned and cached in Johnny. As such, be certain that read queries prone to this issue are executed on a database that is guaranteed to be up to date.

Usage

To enable the QuerySet Cache, enable the middleware `johnny.middleware.QueryCacheMiddleware`.

Manual Invalidation

To manually invalidate a table or a model, use `johnny.cache.invalidate`:

`johnny.cache.`**`invalidate`**`(`*tables*, ***kwargs*`)`
Invalidate the current generation for one or more tables. The arguments can be either strings representing database table names or models. Pass in kwarg `using` to set the database.

Using with scripts, management commands, asynchronous workers and the shell

Since the QuerySet Cache is enabled via middleware, queries made from outside of Django's request-response loop will neither be cached nor used to invalidate the cache. This can lead to stale data in the cache.

You can enable and disable the QuerySet Cache by using the convenience functions:

`johnny.cache.`**`enable`**`()`
Enable johnny-cache, for use in scripts, management commands, async workers, or other code outside the django request flow.

`johnny.cache.`**`disable`**`()`
Disable johnny-cache. This will disable johnny-cache for the whole process, and if writes happen during the time where johnny is disabled, tables will not be invalidated properly. Use Carefully.

To make sure Johnny is always active in management commands, you can enable it the project's `__init__.py` file:

```
django-admin.py createproject myproject
```

Now, in `myproject/__init__.py`:

```
from johnny.cache import enable
enable()
```

This works because `django.core.management.setup_environ` always imports the project module before executing the management command.

Settings

The following settings are available for the QuerySet Cache:

- CACHES .. JOHNNY_CACHE
- DATABASES .. JOHNNY_CACHE_KEY
- DISABLE_QUERYSET_CACHE
- JOHNNY_MIDDLEWARE_KEY_PREFIX
- JOHNNY_MIDDLEWARE_SECONDS
- JOHNNY_TABLE_WHITELIST
- MAN_IN_BLACKLIST (JOHNNY_TABLE_BLACKLIST)

CACHES .. JOHNNY_CACHE is the preferred way of designating a cache as the one used by Johnny. Generally, it will look something like this:

```
CACHES = {
    # ...
    'johnny' : {
        'BACKEND': '...',
        'JOHNNY_CACHE': True,
    }
}
```

Johnny *needs* to run on one shared cache pool, so although the behavior is defined, a warning will be printed if JOHNNY_CACHE is found to be True in multiple cache definitions. If JOHNNY_CACHE is not present, Johnny will fall back to the deprecated JOHNNY_CACHE_BACKEND setting if set, and then to the default cache.

DATABASES .. JOHNNY_CACHE_KEY allows you to override the default key used for the given database. This is useful, for example, in master/slave database setups where writes are never issued to the slave, so Johnny would otherwise not invalidate a query cached for that slave when a write occurs on the master. For example, if you have a simple database setup with one master and one slave, you could set both databases to use the database key default when constructing cache keys like so:

```
DATABASES = {
    # ...
    'default': {
        # ...
        'JOHNNY_CACHE_KEY': 'default',
    },
    'slave': {
        # ...
        'JOHNNY_CACHE_KEY': 'default',
    },
}
```

DISABLE_QUERYSET_CACHE will disable the QuerySet cache even if the middleware is installed. This is mostly to make it easy for non-production environments to disable the queryset cache without re-creating the entire middleware stack and then removing the QuerySet cache middleware.

JOHNNY_MIDDLEWARE_KEY_PREFIX, default "jc", is to set the prefix for Johnny cache. It's *very important* that if you are running multiple apps in the same memcached pool that you use this setting on each app so that tables with the same name in each app (like Django's built in contrib apps) don't clobber each other in the cache.

JOHNNY_MIDDLEWARE_SECONDS, default 0, is the period that Johnny will cache both its generational keys *and* its query cache results. Since the design goal of Johnny was to be able to maintain a consistent cache at all times, the default behavior is to cache everything *forever*. If you are not using one of Johnny's custom backends, the default value of 0 will work differently on different backends and might cause Johnny to never cache anything.

JOHNNY_TABLE_WHITELIST, default "[]", is a user defined tuple that contains table names for exclusive inclusion in the cache. If you provide this setting, the MAN_IN_BLACKLIST (and JOHNNY_TABLE_BLACKLIST) settings are ignored.

MAN_IN_BLACKLIST is a user defined tuple that contains table names to exclude from the QuerySet Cache. If you have no sense of humor, or want your settings file to be understandable, you can use the alias JOHNNY_TABLE_BLACKLIST. We just couldn't resist.

Deprecated

- JOHNNY_CACHE_BACKEND

JOHNNY_CACHE_BACKEND is a cache backend URI similar to what is used by Django by default, but only used for Johnny. In Django 1.2 and earlier, it was impossible to define multiple cache backends for Django's core caching framework, and this was used to allow separation between the cache that is used by Johnny and the caching backend for the rest of your app.

In Django 1.3, this can also take the name of a configured cache, but it is recommended to use the JOHNNY_CACHE cache setting instead.

Signals

The QuerySet Cache defines the following signals:

- johnny.cache.signals.qc_hit, fired after a cache hit
- johnny.cache.signals.qc_miss, fired after a cache miss
- johnny.cache.signals.qc_skip, fired when a query misses cache due to table black/whitelisting

Backwards Compatability Warning: prior to johnny-cache 1.4.1, the qc_miss signal was fired whenever a read query was not found in the cache for any reason, even due to table blacklisting. This made it difficult to use these signals to collect cache overall performance statistics, because blacklisted tables (which would never go to cache even for the table keys) were being recorded as misses.

Although this is a semantic change to the meaning of the qc_miss signal, the original behavior was considered buggy and thus it's been changed from 1.4.0 to 1.4.1. If the original behavior is desired, please connect both the qc_miss and qc_hit signals to the same handler.

The sender of these signals is always the QueryCacheBackend itself.

Customization

There are many aspects of the behavior of the QuerySet Cache that are pluggable, but no easy settings-style hooks are yet provided for them. More ability to control the way Johnny functions is planned for future releases.

1.4.2 The LocalStore Cache

It is a thread-local dict-like object that is cleared at the end of each request by an associated middleware. This can be useful for global data that just be kept, referred to, or even modified throughout the lifetime of a request, like messaging, media registration, or cached datasets.

By default, the LocalStore cache is an instantiated copy of the johnny.localstore.LocalStore class located in johnny.cache.local. The usefulness of the class comes from the middleware that clears it at the end of each request. Being a module-level object, it is a singleton.

Johnny relies on `johnny.cache.local` for its transaction and savepoint support, so it is a good idea to enable the middleware to clear it per request as not doing so can gradually leak memory as this object has no built-in eviction.

class `johnny.localstore.`**`LocalStore`**(***d*)

A thread-local OpenStruct that can be used as a local cache. An instance is located at `johnny.cache.local`, and is cleared on every request by the `LocalStoreClearMiddleware`. It can be a thread-safe way to handle global contexts.

1.4.3 Backends

Johnny provides a number of backends, all of which are subclassed versions of django builtins that cache "forever" when passed a 0 timeout. These are essentially the same as mmalone's inspiring `django-caching` application's cache backend monkey-patch.

The way Django interprets cache backend URIs has changed during its history. Please consult the specific cache documentation for your version of Django for the exact usage you should use.

Example usage:

```
CACHES = {
    'default' : {
        'BACKEND': 'johnny.backends.memcached.PyLibMCCache',
        'LOCATION': '...',
    }
}
```

Important Note: The `locmem` and `filebased` caches are NOT recommended for setups in which there is more than one server using Johnny; invalidation will break with potentially disasterous results if the cache Johnny uses is not shared amongst all machines writing to the database.

memcached

locmem

Infinite caching locmem class. Caches forever when passed timeout of 0.

This actually doesn't cache "forever", just for a very long time. On 32 bit systems, it will cache for 68 years, quite a bit longer than any computer will last. On a 64 bit machine, your cache will expire about 285 billion years after the Sun goes red-giant and destroys Earth.

class `johnny.backends.locmem.`**`LocMemCache`**(*name*, *params*)

filebased

Infinite file-based caching. Caches forever when passed timeout of 0.

class `johnny.backends.filebased.`**`FileBasedCache`**(*dir*, *params*)

redis

Redis cache classes that forcebly limits the timeout of the redis cache backend to 30 days to make sure the cache doesn't fill up when johnny always caches queries. Redis doesn't have an automatic cache invalidation other than timeouts.

This module depends on the `django-redis-cache` app from PyPI.

class `johnny.backends.redis.`**`RedisCache`**(*server*, *params*)

Python Module Index

j

Index