Django’s cache framework

A fundamental trade-off in dynamic websites is, well, they’re dynamic. Each time a user requests a page, the web server makes all sorts of calculations – from database queries to template rendering to business logic – to create the page that your site’s visitor sees. This is a lot more expensive, from a processing-overhead perspective, than your standard read-a-file-off-the-filesystem server arrangement.

For most web applications, this overhead isn’t a big deal. Most web applications aren’t `washingtonpost.com` or `slashdot.org` ; they’re small- to medium-sized sites with so-so traffic. But for medium- to high-traffic sites, it’s essential to cut as much overhead as possible.

That’s where caching comes in.

To cache something is to save the result of an expensive calculation so that you don’t have to perform the calculation next time. Here’s some pseudocode explaining how this would work for a dynamically generated web page:

```
given a URL, try finding that page in the cache
if the page is in the cache:
    return the cached page
else:
    generate the page
    save the generated page in the cache (for next time)
    return the generated page
```

Django comes with a robust cache system that lets you save dynamic pages so they don’t have to be calculated for each request. For convenience, Django offers different levels of cache granularity: You can cache the output of specific views, you can cache only the pieces that are difficult to produce, or you can cache your entire site.

Django also works well with “downstream” caches, such as Squid and browser-based caches. These are the types of caches that you don’t directly control but to which you can provide hints (via HTTP headers) about which parts of your site should be cached, and how.

See also: The Cache Framework design philosophy explains a few of the design decisions of the framework.

Setting up the cache

The cache system requires a small amount of setup. Namely, you have to tell it where your cached data should live – whether in a database, on the filesystem or directly in memory. This is an important decision that affects your cache’s performance; yes, some cache types are faster than others.

Your cache preference goes in the `CACHES` setting in your settings file. Here’s an explanation of all available values for `CACHES`.

Memcached

Memcached is an entirely memory-based cache server, originally developed to handle high loads at LiveJournal.com and subsequently open-sourced by Danga Interactive. It is used by sites such as Facebook and Wikipedia to reduce database access and dramatically increase site performance.

Memcached runs as a daemon and is allotted a specified amount of RAM. All it does is provide a fast interface for adding, retrieving and deleting data in the cache. All data is stored directly in memory, so there’s no overhead of database or filesystem usage.

After installing Memcached itself, you’ll need to install a Memcached binding. There are several Python Memcached bindings available; the two supported by Django are pylibmc and pymemcache.

To use Memcached with Django:

- Set `BACKEND` to `django.core.cache.backends.memcached.PyMemcacheCache` or `django.core.cache.backends.memcached.PyLibMCCache` (depending on your chosen memcached binding)
- Set `LOCATION` to `ip:port` values, where `ip` is the IP address of the Memcached daemon and `port` is the port on which Memcached is running, or to a `unix:path` value, where `path` is the path to a Memcached Unix socket file.

In this example, Memcached is running on localhost (127.0.0.1) port 11211, using the `pymemcache` binding:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.memcached.PyMemcacheCache",
        "LOCATION": "127.0.0.1:11211",
    }
}
```

In this example, Memcached is available through a local Unix socket file `/tmp/memcached.sock` using the `pymemcache` binding:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.memcached.PyMemcacheCache",
        "LOCATION": "unix:/tmp/memcached.sock",
    }
}
```

One excellent feature of Memcached is its ability to share a cache over multiple servers. This means you can run Memcached daemons on multiple machines, and the program will treat the group of machines as a *single* cache, without the need to duplicate cache values on each machine. To take advantage of this feature, include all server addresses in `LOCATION`, either as a semicolon or comma delimited string, or as a list.

In this example, the cache is shared over Memcached instances running on IP address 172.19.26.240 and 172.19.26.242, both on port 11211:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.memcached.PyMemcacheCache",
        "LOCATION": [
            "172.19.26.240:11211",
            "172.19.26.242:11211",
        ],
    }
}
```

In the following example, the cache is shared over Memcached instances running on the IP addresses 172.19.26.240 (port 11211), 172.19.26.242 (port 11212), and 172.19.26.244 (port 11213):

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.memcached.PyMemcacheCache",
        "LOCATION": [
            "172.19.26.240:11211",
            "172.19.26.242:11212",
            "172.19.26.244:11213",
        ],
    }
}
```

By default, the `PyMemcacheCache` backend sets the following options (you can override them in your `OPTIONS`):

```
"OPTIONS": {
    "allow_unicode_keys": True,
    "default_noreply": False,
    "serde": pymemcache.serde.pickle_serde,
}
```

A final point about Memcached is that memory-based caching has a disadvantage: because the cached data is stored in memory, the data will be lost if your server crashes. Clearly, memory isn't intended for permanent data storage, so don't rely on memory-based caching as your only data storage. Without a doubt, *none* of the Django caching backends should be used for permanent storage – they're all intended to be solutions for caching, not storage – but we point this out here because memory-based caching is particularly temporary.

Redis

Redis is an in-memory database that can be used for caching. To begin you'll need a Redis server running either locally or on a remote machine.

After setting up the Redis server, you'll need to install Python bindings for Redis. redis-py is the binding supported natively by Django. Installing the hiredis-py package is also recommended.

To use Redis as your cache backend with Django:

- Set `BACKEND` to `django.core.cache.backends.redis.RedisCache` .
- Set `LOCATION` to the URL pointing to your Redis instance, using the appropriate scheme. See the `redis-py` docs for details on the available schemes.

For example, if Redis is running on localhost (127.0.0.1) port 6379:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.redis.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379",
    }
}
```

Often Redis servers are protected with authentication. In order to supply a username and password, add them in the `LOCATION` along with the URL:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.redis.RedisCache",
        "LOCATION": "redis://username:password@127.0.0.1:6379",
    }
}
```

If you have multiple Redis servers set up in the replication mode, you can specify the servers either as a semicolon or comma delimited string, or as a list. While using multiple servers, write operations are performed on the first server (leader). Read operations are performed on the other servers (replicas) chosen at random:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.redis.RedisCache",
        "LOCATION": [
            "redis://127.0.0.1:6379",  # leader
            "redis://127.0.0.1:6378",  # read-replica 1
            "redis://127.0.0.1:6377",  # read-replica 2
        ],
    }
}
```

Database caching

Django can store its cached data in your database. This works best if you've got a fast, well-indexed database server.

To use a database table as your cache backend:

- Set `BACKEND` to `django.core.cache.backends.db.DatabaseCache`
- Set `LOCATION` to `tablename`, the name of the database table. This name can be whatever you want, as long as it's a valid table name that's not already being used in your database.

In this example, the cache table's name is `my_cache_table`:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.db.DatabaseCache",
        "LOCATION": "my_cache_table",
    }
}
```

Unlike other cache backends, the database cache does not support automatic culling of expired entries at the database level. Instead, expired cache entries are culled each time `add()`, `set()`, or `touch()` is called.

Creating the cache table

Before using the database cache, you must create the cache table with this command:

```
python manage.py createcachetable
```

This creates a table in your database that is in the proper format that Django's database-cache system expects. The name of the table is taken from `LOCATION`.

If you are using multiple database caches, `createcachetable` creates one table for each cache.

If you are using multiple databases, `createcachetable` observes the `allow_migrate()` method of your database routers (see below).

Like `migrate`, `createcachetable` won't touch an existing table. It will only create missing tables.

To print the SQL that would be run, rather than run it, use the `createcachetable --dry-run` option.

Multiple databases

If you use database caching with multiple databases, you'll also need to set up routing instructions for your database cache table. For the purposes of routing, the database cache table appears as a model named `CacheEntry`, in an application named `django_cache`. This model won't appear in the models cache, but the model details can be used for routing purposes.

For example, the following router would direct all cache read operations to `cache_replica`, and all write operations to `cache_primary`. The cache table will only be synchronized onto `cache_primary`:

```
class CacheRouter:
    """A router to control all database cache operations"""

    def db_for_read(self, model, **hints):
        "All cache read operations go to the replica"
        if model._meta.app_label == "django_cache":
            return "cache_replica"
        return None

    def db_for_write(self, model, **hints):
        "All cache write operations go to primary"
        if model._meta.app_label == "django_cache":
            return "cache_primary"
        return None

    def allow_migrate(self, db, app_label, model_name=None, **hints):
        "Only install the cache model on primary"
        if app_label == "django_cache":
            return db == "cache_primary"
        return None
```

If you don't specify routing directions for the database cache model, the cache backend will use the `default` database.

And if you don't use the database cache backend, you don't need to worry about providing routing instructions for the database cache model.

Filesystem caching

The file-based backend serializes and stores each cache value as a separate file. To use this backend set `BACKEND` to `"django.core.cache.backends.filebased.FileBasedCache"` and `LOCATION` to a suitable directory. For example, to store cached data in `/var/tmp/django_cache`, use this setting:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.filebased.FileBasedCache",
        "LOCATION": "/var/tmp/django_cache",
    }
}
```

If you're on Windows, put the drive letter at the beginning of the path, like this:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.filebased.FileBasedCache",
        "LOCATION": "c:/foo/bar",
    }
}
```

The directory path should be absolute – that is, it should start at the root of your filesystem. It doesn't matter whether you put a slash at the end of the setting.

Make sure the directory pointed-to by this setting either exists and is readable and writable, or that it can be created by the system user under which your web server runs. Continuing the above example, if your server runs as the user `apache`, make sure the directory `/var/tmp/django_cache` exists and is readable and writable by the user `apache`, or that it can be created by the user `apache`.

Warning: When the cache `LOCATION` is contained within `MEDIA_ROOT`, `STATIC_ROOT`, or `STATICFILES_FINDERS`, sensitive data may be exposed.

An attacker who gains access to the cache file can not only falsify HTML content, which your site will trust, but also remotely execute arbitrary code, as the data is serialized using `pickle`.

Warning: Filesystem caching may become slow when storing a large number of files. If you run into this problem, consider using a different caching mechanism. You can also subclass FileBasedCache and improve the culling strategy.

Local-memory caching

This is the default cache if another is not specified in your settings file. If you want the speed advantages of in-memory caching but don't have the capability of running Memcached, consider the local-memory cache backend. This cache is per-process (see below) and thread-safe. To use it, set `BACKEND` to `"django.core.cache.backends.locmem.LocMemCache"`. For example:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.locmem.LocMemCache",
        "LOCATION": "unique-snowflake",
    }
}
```

The cache `LOCATION` is used to identify individual memory stores. If you only have one `locmem` cache, you can omit the `LOCATION`; however, if you have more than one local memory cache, you will need to assign a name to at least one of them in order to keep them separate.

The cache uses a least-recently-used (LRU) culling strategy.

Note that each process will have its own private cache instance, which means no cross-process caching is possible. This also means the local memory cache isn't particularly memory-efficient, so it's probably not a good choice for production environments. It's nice for development.

Dummy caching (for development)

Finally, Django comes with a "dummy" cache that doesn't actually cache – it just implements the cache interface without doing anything.

This is useful if you have a production site that uses heavy-duty caching in various places but a development/test environment where you don't want to cache and don't want to have to change your code to special-case the latter. To activate dummy caching, set `BACKEND` like so:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.dummy.DummyCache",
    }
}
```

Using a custom cache backend

While Django includes support for a number of cache backends out-of-the-box, sometimes you might want to use a customized cache backend. To use an external cache backend with Django, use the Python import path as the `BACKEND` of the `CACHES` setting, like so:

```
CACHES = {
    "default": {
        "BACKEND": "path.to.backend",
    }
}
```

If you're building your own backend, you can use the standard cache backends as reference implementations. You'll find the code in the django/core/cache/backends/ directory of the Django source.

Note: Without a really compelling reason, such as a host that doesn't support them, you should stick to the cache backends included with Django. They've been well-tested and are well-documented.

Cache arguments

Each cache backend can be given additional arguments to control caching behavior. These arguments are provided as additional keys in the `CACHES` setting. Valid arguments are as follows:

- `TIMEOUT`: The default timeout, in seconds, to use for the cache. This argument defaults to `300` seconds (5 minutes). You can set `TIMEOUT` to `None` so that, by default, cache keys never expire. A value of `0` causes keys to immediately expire (effectively "don't cache").
- `OPTIONS`: Any options that should be passed to the cache backend. The list of valid options will vary with each backend, and cache backends backed by a third-party library will pass their options directly to the underlying cache library.

  Cache backends that implement their own culling strategy (i.e., the `locmem`, `filesystem` and `database` backends) will honor the following options:

  - `MAX_ENTRIES`: The maximum number of entries allowed in the cache before old values are deleted. This argument defaults to `300`.
  - `CULL_FREQUENCY`: The fraction of entries that are culled when `MAX_ENTRIES` is reached. The actual ratio is `1 / CULL_FREQUENCY`, so set `CULL_FREQUENCY` to `2` to cull half the entries when `MAX_ENTRIES` is reached. This argument should be an integer and defaults to `3`.

    A value of `0` for `CULL_FREQUENCY` means that the entire cache will be dumped when `MAX_ENTRIES` is reached. On some backends (`database` in particular) this makes culling *much* faster at the expense of more cache misses.

The Memcached and Redis backends pass the contents of OPTIONS as keyword arguments to the client constructors, allowing for more advanced control of client behavior. For example usage, see below.

- KEY_PREFIX: A string that will be automatically included (prepended by default) to all cache keys used by the Django server.

  See the cache documentation for more information.
- VERSION: The default version number for cache keys generated by the Django server.

  See the cache documentation for more information.
- KEY_FUNCTION A string containing a dotted path to a function that defines how to compose a prefix, version and key into a final cache key.

  See the cache documentation for more information.

In this example, a filesystem backend is being configured with a timeout of 60 seconds, and a maximum capacity of 1000 items:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.filebased.FileBasedCache",
        "LOCATION": "/var/tmp/django_cache",
        "TIMEOUT": 60,
        "OPTIONS": {"MAX_ENTRIES": 1000},
    }
}
```

Here's an example configuration for a `pylibmc` based backend that enables the binary protocol, SASL authentication, and the `ketama` behavior mode:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.memcached.PyLibMCCache",
        "LOCATION": "127.0.0.1:11211",
        "OPTIONS": {
            "binary": True,
            "username": "user",
            "password": "pass",
            "behaviors": {
                "ketama": True,
            },
        },
    }
}
```

Here's an example configuration for a `pymemcache` based backend that enables client pooling (which may improve performance by keeping clients connected), treats memcache/network errors as cache misses, and sets the `TCP_NODELAY` flag on the connection's socket:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.memcached.PyMemcacheCache",
        "LOCATION": "127.0.0.1:11211",
        "OPTIONS": {
            "no_delay": True,
            "ignore_exc": True,
            "max_pool_size": 4,
            "use_pooling": True,
        },
    }
}
```

Here's an example configuration for a `redis` based backend that selects database `10` (by default Redis ships with 16 logical databases), and sets a custom connection pool class (`redis.ConnectionPool` is used by default):

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.redis.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379",
        "OPTIONS": {
            "db": "10",
            "pool_class": "redis.BlockingConnectionPool",
        },
    }
}
```

The per-site cache

Once the cache is set up, the simplest way to use caching is to cache your entire site. You'll need to add 'django.middleware.cache.UpdateCacheMiddleware' and 'django.middleware.cache.FetchFromCacheMiddleware' to your MIDDLEWARE setting, as in this example:

```
MIDDLEWARE = [
    "django.middleware.cache.UpdateCacheMiddleware",
    "django.middleware.common.CommonMiddleware",
    "django.middleware.cache.FetchFromCacheMiddleware",
]
```

Note: No, that's not a typo: the "update" middleware must be first in the list, and the "fetch" middleware must be last. The details are a bit obscure, but see Order of MIDDLEWARE below if you'd like the full story.

Then, add the following required settings to your Django settings file:

- CACHE_MIDDLEWARE_ALIAS – The cache alias to use for storage.
- CACHE_MIDDLEWARE_SECONDS – The integer number of seconds each page should be cached.
- CACHE_MIDDLEWARE_KEY_PREFIX – If the cache is shared across multiple sites using the same Django installation, set this to the name of the site, or some other string that is unique to this Django instance, to prevent key collisions. Use an empty string if you don't care.

FetchFromCacheMiddleware caches GET and HEAD responses with status 200, where the request and response headers allow. Responses to requests for the same URL with different query parameters are considered to be unique pages and are cached separately. This middleware expects that a HEAD request is answered with the same response headers as the corresponding GET request; in which case it can return a cached GET response for HEAD request.

Additionally, UpdateCacheMiddleware automatically sets a few headers in each HttpResponse which affect downstream caches:

- Sets the Expires header to the current date/time plus the defined CACHE_MIDDLEWARE_SECONDS.
- Sets the Cache-Control header to give a max age for the page – again, from the CACHE_MIDDLEWARE_SECONDS setting.

See Middleware for more on middleware.

If a view sets its own cache expiry time (i.e. it has a max-age section in its Cache-Control header) then the page will be cached until the expiry time, rather than CACHE_MIDDLEWARE_SECONDS. Using the decorators in django.views.decorators.cache you can easily set a view's expiry time (using the cache_control() decorator) or disable caching for a view (using the never_cache() decorator). See the using other headers section for more on these decorators.

If USE_I18N is set to True then the generated cache key will include the name of the active language – see also How Django discovers language preference). This allows you to easily cache multilingual sites without having to create the cache key yourself.

Cache keys also include the current time zone when USE_TZ is set to True .

The per-view cache

```
django.views.decorators.cache.cache_page(timeout, *, cache=None, key_prefix=None)
```

A more granular way to use the caching framework is by caching the output of individual views. django.views.decorators.cache defines a cache_page decorator that will automatically cache the view's response for you:

```
from django.views.decorators.cache import cache_page


@cache_page(60 * 15)
def my_view(request): ...
```

cache_page takes a single argument: the cache timeout, in seconds. In the above example, the result of the my_view() view will be cached for 15 minutes. (Note that we've written it as 60 * 15 for the purpose of readability. 60 * 15 will be evaluated to 900 – that is, 15 minutes multiplied by 60 seconds per minute.)

The cache timeout set by cache_page takes precedence over the max-age directive from the Cache-Control header.

The per-view cache, like the per-site cache, is keyed off of the URL. If multiple URLs point at the same view, each URL will be cached separately. Continuing the my_view example, if your URLconf looks like this:

```
urlpatterns = [
    path("foo/<int:code>/", my_view),
]
```

then requests to `/foo/1/` and `/foo/23/` will be cached separately, as you may expect. But once a particular URL (e.g., `/foo/23/`) has been requested, subsequent requests to that URL will use the cache.

`cache_page` can also take an optional keyword argument, `cache`, which directs the decorator to use a specific cache (from your `CACHES` setting) when caching view results. By default, the `default` cache will be used, but you can specify any cache you want:

```
@cache_page(60 * 15, cache="special_cache")
def my_view(request): ...
```

You can also override the cache prefix on a per-view basis. `cache_page` takes an optional keyword argument, `key_prefix`, which works in the same way as the `CACHE_MIDDLEWARE_KEY_PREFIX` setting for the middleware. It can be used like this:

```
@cache_page(60 * 15, key_prefix="site1")
def my_view(request): ...
```

The `key_prefix` and `cache` arguments may be specified together. The `key_prefix` argument and the `KEY_PREFIX` specified under `CACHES` will be concatenated.

Additionally, `cache_page` automatically sets `Cache-Control` and `Expires` headers in the response which affect downstream caches.

Specifying per-view cache in the URLconf

The examples in the previous section have hard-coded the fact that the view is cached, because `cache_page` alters the `my_view` function in place. This approach couples your view to the cache system, which is not ideal for several reasons. For instance, you might want to reuse the view functions on another, cache-less site, or you might want to distribute the views to people who might want to use them without being cached. The solution to these problems is to specify the per-view cache in the URLconf rather than next to the view functions themselves.

You can do so by wrapping the view function with `cache_page` when you refer to it in the URLconf. Here's the old URLconf from earlier:

```
urlpatterns = [
    path("foo/<int:code>/", my_view),
]
```

Here's the same thing, with `my_view` wrapped in `cache_page`:

```
from django.views.decorators.cache import cache_page

urlpatterns = [
    path("foo/<int:code>/", cache_page(60 * 15)(my_view)),
]
```

Template fragment caching

If you're after even more control, you can also cache template fragments using the `cache` template tag. To give your template access to this tag, put `{% load cache %}` near the top of your template.

The `{% cache %}` template tag caches the contents of the block for a given amount of time. It takes at least two arguments: the cache timeout, in seconds, and the name to give the cache fragment. The fragment is cached forever if timeout is `None`. The name will be taken as is, do not use a variable. For example:

```
{% load cache %}
{% cache 500 sidebar %}
    .. sidebar ..
{% endcache %}
```

Sometimes you might want to cache multiple copies of a fragment depending on some dynamic data that appears inside the fragment. For example, you might want a separate cached copy of the sidebar used in the previous example for every user of your site. Do this by passing one or more additional arguments, which may be variables with or without filters, to the `{% cache %}` template tag to uniquely identify the cache fragment:

```
{% load cache %}
{% cache 500 sidebar request.user.username %}
    .. sidebar for logged in user ..
{% endcache %}
```

If USE_I18N is set to `True` the per-site middleware cache will respect the active language. For the `cache` template tag you could use one of the translation-specific variables available in templates to achieve the same result:

```
{% load i18n %}
{% load cache %}

{% get_current_language as LANGUAGE_CODE %}

{% cache 600 welcome LANGUAGE_CODE %}
    {% translate "Welcome to example.com" %}
{% endcache %}
```

The cache timeout can be a template variable, as long as the template variable resolves to an integer value. For example, if the template variable `my_timeout` is set to the value `600`, then the following two examples are equivalent:

```
{% cache 600 sidebar %} ... {% endcache %}
{% cache my_timeout sidebar %} ... {% endcache %}
```

This feature is useful in avoiding repetition in templates. You can set the timeout in a variable, in one place, and reuse that value.

By default, the cache tag will try to use the cache called "template_fragments". If no such cache exists, it will fall back to using the default cache. You may select an alternate cache backend to use with the `using` keyword argument, which must be the last argument to the tag.

```
{% cache 300 local-thing ...  using="localcache" %}
```

It is considered an error to specify a cache name that is not configured.

```
django.core.cache.utils.make_template_fragment_key(fragment_name, vary_on=None)
```

If you want to obtain the cache key used for a cached fragment, you can use `make_template_fragment_key`. `fragment_name` is the same as second argument to the `cache` template tag; `vary_on` is a list of all additional arguments passed to the tag. This function can be useful for invalidating or overwriting a cached item, for example:

```
>>> from django.core.cache import cache
>>> from django.core.cache.utils import make_template_fragment_key
# cache key for {% cache 500 sidebar username %}
>>> key = make_template_fragment_key("sidebar", [username])
>>> cache.delete(key)  # invalidates cached template fragment
True
```

The low-level cache API

Sometimes, caching an entire rendered page doesn't gain you very much and is, in fact, inconvenient overkill.

Perhaps, for instance, your site includes a view whose results depend on several expensive queries, the results of which change at different intervals. In this case, it would not be ideal to use the full-page caching that the per-site or per-view cache strategies offer, because you wouldn't want to cache the entire result (since some of the data changes often), but you'd still want to cache the results that rarely change.

For cases like this, Django exposes a low-level cache API. You can use this API to store objects in the cache with any level of granularity you like. You can cache any Python object that can be pickled safely: strings, dictionaries, lists of model objects, and so forth. (Most common Python objects can be pickled; refer to the Python documentation for more information about pickling.)

Accessing the cache

```
django.core.cache.caches
```

You can access the caches configured in the CACHES setting through a dict-like object: `django.core.cache.caches`. Repeated requests for the same alias in the same thread will return the same object.

```
>>> from django.core.cache import caches
>>> cache1 = caches["myalias"]
>>> cache2 = caches["myalias"]
>>> cache1 is cache2
True
```

If the named key does not exist, `InvalidCacheBackendError` will be raised.

To provide thread-safety, a different instance of the cache backend will be returned for each thread.

```
django.core.cache.cache
```

As a shortcut, the default cache is available as `django.core.cache.cache` :

```
>>> from django.core.cache import cache
```

This object is equivalent to `caches['default']` .

Basic usage

The basic interface is:

```
cache.set(key, value, timeout=DEFAULT_TIMEOUT, version=None)
```

```
>>> cache.set("my_key", "hello, world!", 30)
```

```
cache.get(key, default=None, version=None)
```

```
>>> cache.get("my_key")
'hello, world!'
```

`key` should be a `str` , and `value` can be any picklable Python object.

The `timeout` argument is optional and defaults to the `timeout` argument of the appropriate backend in the CACHES setting (explained above). It's the number of seconds the value should be stored in the cache. Passing in `None` for `timeout` will cache the value forever. A `timeout` of `0` won't cache the value.

If the object doesn't exist in the cache, `cache.get()` returns `None` :

```
>>> # Wait 30 seconds for 'my_key' to expire...
>>> cache.get("my_key")
None
```

If you need to determine whether the object exists in the cache and you have stored a literal value `None` , use a sentinel object as the default:

```
>>> sentinel = object()
>>> cache.get("my_key", sentinel) is sentinel
False
>>> # Wait 30 seconds for 'my_key' to expire...
>>> cache.get("my_key", sentinel) is sentinel
True
```

`cache.get()` can take a `default` argument. This specifies which value to return if the object doesn't exist in the cache:

```
>>> cache.get("my_key", "has expired")
'has expired'
```

```
cache.add(key, value, timeout=DEFAULT_TIMEOUT, version=None)
```

To add a key only if it doesn't already exist, use the `add()` method. It takes the same parameters as `set()` , but it will not attempt to update the cache if the key specified is already present:

```
>>> cache.set("add_key", "Initial value")
>>> cache.add("add_key", "New value")
>>> cache.get("add_key")
'Initial value'
```

If you need to know whether `add()` stored a value in the cache, you can check the return value. It will return `True` if the value was stored, `False` otherwise.

```
cache.get_or_set(key, default, timeout=DEFAULT_TIMEOUT, version=None)
```

If you want to get a key's value or set a value if the key isn't in the cache, there is the `get_or_set()` method. It takes the same parameters as `get()` but the default is set as the new cache value for that key, rather than returned:

```
>>> cache.get("my_new_key")  # returns None
>>> cache.get_or_set("my_new_key", "my new value", 100)
'my new value'
```

You can also pass any callable as a *default* value:

```
>>> import datetime
>>> cache.get_or_set("some-timestamp-key", datetime.datetime.now)
datetime.datetime(2014, 12, 11, 0, 15, 49, 457920)
```

```
cache.get_many(keys, version=None)
```

There's also a `get_many()` interface that only hits the cache once. `get_many()` returns a dictionary with all the keys you asked for that actually exist in the cache (and haven't expired):

```
>>> cache.set("a", 1)
>>> cache.set("b", 2)
>>> cache.set("c", 3)
>>> cache.get_many(["a", "b", "c"])
{'a': 1, 'b': 2, 'c': 3}
```

```
cache.set_many(dict, timeout)
```

To set multiple values more efficiently, use `set_many()` to pass a dictionary of key-value pairs:

```
>>> cache.set_many({"a": 1, "b": 2, "c": 3})
>>> cache.get_many(["a", "b", "c"])
{'a': 1, 'b': 2, 'c': 3}
```

Like `cache.set()` , `set_many()` takes an optional `timeout` parameter.

On supported backends (memcached), `set_many()` returns a list of keys that failed to be inserted.

```
cache.delete(key, version=None)
```

You can delete keys explicitly with `delete()` to clear the cache for a particular object:

```
>>> cache.delete("a")
True
```

`delete()` returns `True` if the key was successfully deleted, `False` otherwise.

```
cache.delete_many(keys, version=None)
```

If you want to clear a bunch of keys at once, `delete_many()` can take a list of keys to be cleared:

```
>>> cache.delete_many(["a", "b", "c"])
```

```
cache.clear()
```

Finally, if you want to delete all the keys in the cache, use `cache.clear()` . Be careful with this; `clear()` will remove *everything* from the cache, not just the keys set by your application:

```
>>> cache.clear()
```

```
cache.touch(key, timeout=DEFAULT_TIMEOUT, version=None)
```

`cache.touch()` sets a new expiration for a key. For example, to update a key to expire 10 seconds from now:

```
>>> cache.touch("a", 10)
True
```

Like other methods, the `timeout` argument is optional and defaults to the `TIMEOUT` option of the appropriate backend in the CACHES setting.

`touch()` returns `True` if the key was successfully touched, `False` otherwise.

```
cache.incr(key, delta=1, version=None)
```

```
cache.decr(key, delta=1, version=None)
```

You can also increment or decrement a key that already exists using the `incr()` or `decr()` methods, respectively. By default, the existing cache value will be incremented or decremented by 1. Other increment/decrement values can be specified by providing an argument to the increment/decrement call. A ValueError will be raised if you attempt to increment or decrement a nonexistent cache key:

```
>>> cache.set("num", 1)
>>> cache.incr("num")
2
>>> cache.incr("num", 10)
12
>>> cache.decr("num")
11
>>> cache.decr("num", 5)
6
```

Note: `incr()` / `decr()` methods are not guaranteed to be atomic. On those backends that support atomic increment/decrement (most notably, the memcached backend), increment and decrement operations will be atomic. However, if the backend doesn't natively provide an increment/decrement operation, it will be implemented using a two-step retrieve/update.

```
cache.close()
```

You can close the connection to your cache with `close()` if implemented by the cache backend.

```
>>> cache.close()
```

Note: For caches that don't implement `close` methods it is a no-op.

Note: The async variants of base methods are prefixed with `a`, e.g. `cache.aadd()` or `cache.adelete_many()`. See Asynchronous support for more details.

Cache key prefixing

If you are sharing a cache instance between servers, or between your production and development environments, it's possible for data cached by one server to be used by another server. If the format of cached data is different between servers, this can lead to some very hard to diagnose problems.

To prevent this, Django provides the ability to prefix all cache keys used by a server. When a particular cache key is saved or retrieved, Django will automatically prefix the cache key with the value of the `KEY_PREFIX` cache setting.

By ensuring each Django instance has a different `KEY_PREFIX`, you can ensure that there will be no collisions in cache values.

Cache versioning

When you change running code that uses cached values, you may need to purge any existing cached values. The easiest way to do this is to flush the entire cache, but this can lead to the loss of cache values that are still valid and useful.

Django provides a better way to target individual cache values. Django's cache framework has a system-wide version identifier, specified using the `VERSION` cache setting. The value of this setting is automatically combined with the cache prefix and the user-provided cache key to obtain the final cache key.

By default, any key request will automatically include the site default cache key version. However, the primitive cache functions all include a `version` argument, so you can specify a particular cache key version to set or get. For example:

```
>>> # Set version 2 of a cache key
>>> cache.set("my_key", "hello world!", version=2)
>>> # Get the default version (assuming version=1)
>>> cache.get("my_key")
None
>>> # Get version 2 of the same key
>>> cache.get("my_key", version=2)
'hello world!'
```

The version of a specific key can be incremented and decremented using the `incr_version()` and `decr_version()` methods. This enables specific keys to be bumped to a new version, leaving other keys unaffected. Continuing our previous example:

```
>>> # Increment the version of 'my_key'
>>> cache.incr_version("my_key")
>>> # The default version still isn't available
>>> cache.get("my_key")
None
# Version 2 isn't available, either
>>> cache.get("my_key", version=2)
None
>>> # But version 3 *is* available
>>> cache.get("my_key", version=3)
'hello world!'
```

Cache key transformation

As described in the previous two sections, the cache key provided by a user is not used verbatim – it is combined with the cache prefix and key version to provide a final cache key. By default, the three parts are joined using colons to produce a final string:

```
def make_key(key, key_prefix, version):
    return "%s:%s:%s" % (key_prefix, version, key)
```

If you want to combine the parts in different ways, or apply other processing to the final key (e.g., taking a hash digest of the key parts), you can provide a custom key function.

The `KEY_FUNCTION` cache setting specifies a dotted-path to a function matching the prototype of `make_key()` above. If provided, this custom key function will be used instead of the default key combining function.

Cache key warnings

Memcached, the most commonly-used production cache backend, does not allow cache keys longer than 250 characters or containing whitespace or control characters, and using such keys will cause an exception. To encourage cache-portable code and minimize unpleasant surprises, the other built-in cache backends issue a warning (`django.core.cache.backends.base.CacheKeyWarning`) if a key is used that would cause an error on memcached.

If you are using a production backend that can accept a wider range of keys (a custom backend, or one of the non-memcached built-in backends), and want to use this wider range without warnings, you can silence `CacheKeyWarning` with this code in the `management` module of one of your `INSTALLED_APPS`:

```
import warnings

from django.core.cache import CacheKeyWarning

warnings.simplefilter("ignore", CacheKeyWarning)
```

If you want to instead provide custom key validation logic for one of the built-in backends, you can subclass it, override just the `validate_key` method, and follow the instructions for using a custom cache backend. For instance, to do this for the `locmem` backend, put this code in a module:

```
from django.core.cache.backends.locmem import LocMemCache


class CustomLocMemCache(LocMemCache):
    def validate_key(self, key):
        """Custom validation, raising exceptions or warnings as needed."""
        ...
```

...and use the dotted Python path to this class in the `BACKEND` portion of your `CACHES` setting.

Asynchronous support

Django has developing support for asynchronous cache backends, but does not yet support asynchronous caching. It will be coming in a future release.

`django.core.cache.backends.base.BaseCache` has async variants of all base methods. By convention, the asynchronous versions of all methods are prefixed with `a`. By default, the arguments for both variants are the same:

```
>>> await cache.aset("num", 1)
>>> await cache.ahas_key("num")
True
```

Downstream caches

So far, this document has focused on caching your *own* data. But another type of caching is relevant to web development, too: caching performed by "downstream" caches. These are systems that cache pages for users even before the request reaches your website.

Here are a few examples of downstream caches:

- When using HTTP, your ISP may cache certain pages, so if you requested a page from `http://example.com/`, your ISP would send you the page without having to access example.com directly. The maintainers of example.com have no knowledge of this caching; the ISP sits between example.com and your web browser, handling all of the caching transparently. Such caching is not possible under HTTPS as it would constitute a man-in-the-middle attack.
- Your Django website may sit behind a *proxy cache*, such as Squid Web Proxy Cache (http://www.squid-cache.org/), that caches pages for performance. In this case, each request first would be handled by the proxy, and it would be passed to your application only if needed.
- Your web browser caches pages, too. If a web page sends out the appropriate headers, your browser will use the local cached copy for subsequent requests to that page, without even contacting the web page again to see whether it has changed.

Downstream caching is a nice efficiency boost, but there's a danger to it: Many web pages' contents differ based on authentication and a host of other variables, and cache systems that blindly save pages based purely on URLs could expose incorrect or sensitive data to subsequent visitors to those pages.

For example, if you operate a web email system, then the contents of the "inbox" page depend on which user is logged in. If an ISP blindly cached your site, then the first user who logged in through that ISP would have their user-specific inbox page cached for subsequent visitors to the site. That's not cool.

Fortunately, HTTP provides a solution to this problem. A number of HTTP headers exist to instruct downstream caches to differ their cache contents depending on designated variables, and to tell caching mechanisms not to cache particular pages. We'll look at some of these headers in the sections that follow.

Using `Vary` headers

The `Vary` header defines which request headers a cache mechanism should take into account when building its cache key. For example, if the contents of a web page depend on a user's language preference, the page is said to "vary on language."

By default, Django's cache system creates its cache keys using the requested fully-qualified URL – e.g., `"https://www.example.com/stories/2005/?order_by=author"`. This means every request to that URL will use the same cached version, regardless of user-agent differences such as cookies or language preferences. However, if this page produces different content based on some difference in request headers – such as a cookie, or a language, or a user-agent – you'll need to use the `Vary` header to tell caching mechanisms that the page output depends on those things.

To do this in Django, use the convenient `django.views.decorators.vary.vary_on_headers()` view decorator, like so:

```
from django.views.decorators.vary import vary_on_headers


@vary_on_headers("User-Agent")
def my_view(request): ...
```

In this case, a caching mechanism (such as Django's own cache middleware) will cache a separate version of the page for each unique user-agent.

The advantage to using the `vary_on_headers` decorator rather than manually setting the `Vary` header (using something like `response.headers['Vary'] = 'user-agent'`) is that the decorator *adds* to the `Vary` header (which may already exist), rather than setting it from scratch and potentially overriding anything that was already in there.

You can pass multiple headers to `vary_on_headers()`:

```
@vary_on_headers("User-Agent", "Cookie")
def my_view(request): ...
```

This tells downstream caches to vary on *both*, which means each combination of user-agent and cookie will get its own cache value. For example, a request with the user-agent `Mozilla` and the cookie value `foo=bar` will be considered different from a request with the user-agent `Mozilla` and the cookie value `foo=ham`.

Because varying on cookie is so common, there's a `django.views.decorators.vary.vary_on_cookie()` decorator. These two views are equivalent:

```
@vary_on_cookie
def my_view(request): ...


@vary_on_headers("Cookie")
def my_view(request): ...
```

The headers you pass to `vary_on_headers` are not case sensitive; `"User-Agent"` is the same thing as `"user-agent"`.

You can also use a helper function, `django.utils.cache.patch_vary_headers()`, directly. This function sets, or adds to, the `Vary header`. For example:

```
from django.shortcuts import render
from django.utils.cache import patch_vary_headers


def my_view(request):
    ...
    response = render(request, "template_name", context)
    patch_vary_headers(response, ["Cookie"])
    return response
```

`patch_vary_headers` takes an `HttpResponse` instance as its first argument and a list/tuple of case-insensitive header names as its second argument.

For more on Vary headers, see the **official Vary spec**.

Controlling cache: Using other headers

Other problems with caching are the privacy of data and the question of where data should be stored in a cascade of caches.

A user usually faces two kinds of caches: their own browser cache (a private cache) and their provider's cache (a public cache). A public cache is used by multiple users and controlled by someone else. This poses problems with sensitive data–you don't want, say, your bank account number stored in a public cache. So web applications need a way to tell caches which data is private and which is public.

The solution is to indicate a page's cache should be "private." To do this in Django, use the `cache_control()` view decorator. Example:

```
from django.views.decorators.cache import cache_control


@cache_control(private=True)
def my_view(request): ...
```

This decorator takes care of sending out the appropriate HTTP header behind the scenes.

Note that the cache control settings "private" and "public" are mutually exclusive. The decorator ensures that the "public" directive is removed if "private" should be set (and vice versa). An example use of the two directives would be a blog site that offers both private and public entries. Public entries may be cached on any shared cache. The following code uses `patch_cache_control()`, the manual way to modify the cache control header (it is internally called by the `cache_control()` decorator):

```
from django.views.decorators.cache import patch_cache_control
from django.views.decorators.vary import vary_on_cookie


@vary_on_cookie
def list_blog_entries_view(request):
    if request.user.is_anonymous:
        response = render_only_public_entries()
        patch_cache_control(response, public=True)
    else:
        response = render_private_and_public_entries(request.user)
        patch_cache_control(response, private=True)

    return response
```

You can control downstream caches in other ways as well (see **RFC 9111** for details on HTTP caching). For example, even if you don't use Django's server-side cache framework, you can still tell clients to cache a view for a certain amount of time with the **max-age** directive:

```
from django.views.decorators.cache import cache_control


@cache_control(max_age=3600)
def my_view(request): ...
```

(If you *do* use the caching middleware, it already sets the `max-age` with the value of the `CACHE_MIDDLEWARE_SECONDS` setting. In that case, the custom `max_age` from the `cache_control()` decorator will take precedence, and the header values will be merged correctly.)

Any valid `Cache-Control` response directive is valid in `cache_control()`. Here are some more examples:

- `no_transform=True`
- `must_revalidate=True`
- `stale_while_revalidate=num_seconds`
- `no_cache=True`

The full list of known directives can be found in the IANA registry (note that not all of them apply to responses).

If you want to use headers to disable caching altogether, `never_cache()` is a view decorator that adds headers to ensure the response won't be cached by browsers or other caches. Example:

```
from django.views.decorators.cache import never_cache


@never_cache
def myview(request): ...
```

Order of `MIDDLEWARE`

If you use caching middleware, it's important to put each half in the right place within the `MIDDLEWARE` setting. That's because the cache middleware needs to know which headers by which to vary the cache storage. Middleware always adds something to the `Vary` response header when it can.

`UpdateCacheMiddleware` runs during the response phase, where middleware is run in reverse order, so an item at the top of the list runs *last* during the response phase. Thus, you need to make sure that `UpdateCacheMiddleware` appears *before* any other middleware that might add something to the `Vary` header. The following middleware modules do so:

- `SessionMiddleware` adds `Cookie`
- `GZipMiddleware` adds `Accept-Encoding`
- `LocaleMiddleware` adds `Accept-Language`

`FetchFromCacheMiddleware`, on the other hand, runs during the request phase, where middleware is applied first-to-last, so an item at the top of the list runs *first* during the request phase. The `FetchFromCacheMiddleware` also needs to run after other middleware updates the `Vary` header, so `FetchFromCacheMiddleware` must be *after* any item that does so.