Chapter 15: Caching

A fundamental trade-off in dynamic Web sites 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 simply 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 "upstream" caches, such as Squid (http://www.squid-cache.org/) 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.

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 CACHE_BACKEND setting in your settings file. Here's an explanation of all available values for CACHE_BACKEND.

Memcached

By far the fastest, most efficient type of cache available to Django, Memcached is an entirely memory-based cache framework originally developed to handle high loads at LiveJournal.com and subsequently open-sourced by Danga Interactive. It's used by sites such as Facebook and Wikipedia to reduce database access and dramatically increase site performance.

Memcached is available for free at http://danga.com/memcached/ . It runs as a daemon and is allotted a specified amount of RAM. All it does is provide an fast interface for adding, retrieving and deleting arbitrary 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 the Memcached Python bindings, which are not bundled with Django directly. Two versions of this are available. Choose and install *one* of the following modules:

- The fastest available option is a module called cmemcache, available at http://gijsbert.org/cmemcache/ .
- If you can't install cmemcache, you can install python-memcached, available atftp://ftp.tummy.com/pub/python-memcached/ . If that URL is no longer valid, just go to the Memcached Web site (http://www.danga.com/memcached/) and get the Python bindings from the "Client APIs" section.

To use Memcached with Django, set CACHE_BACKEND to memcached://ip:port/, where ip is the IP address of the Memcached daemon and port is the port on which Memcached is running.

In this example, Memcached is running on localhost (127.0.0.1) port 11211:

```
CACHE_BACKEND = 'memcached://127.0.0.1:11211/'
```

One excellent feature of Memcached is its ability to share 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 CACHE_BACKEND, separated by semicolons.

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:

```
CACHE_BACKEND =
'memcached://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):

```
CACHE_BACKEND =
'memcached://172.19.26.240:11211;172.19.26.242:11212;17
2.19.26.244:11213/'
```

A final point about Memcached is that memory-based caching has one 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.

Database Caching

To use a database table as your cache backend, first create a cache table in your database by running this command:

```
python manage.py createcachetable [cache_table_name]
```

...where [cache_table_name] is the name of the database table to create. (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.) This command creates a single table in your database that is in the proper format that Django's database-cache system expects.

Once you've created that database table, set your CACHE_BACKEND setting to "db://tablename", wheretablename is the name of the database table. In this example, the cache table's name is my_cache_table:

```
CACHE_BACKEND = 'db://my_cache_table'
```

The database caching backend uses the same database as specified in your settings file. You can't use a different database backend for your cache table.

Database caching works best if you've got a fast, well-indexed database server.

Filesystem Caching

To store cached items on a filesystem, use the "file://" cache type for CACHE_BACKEND. For example, to store cached data in /var/tmp/django_cache, use this setting:

```
CACHE_BACKEND = 'file:///var/tmp/django_cache'
```

Note that there are three forward slashes toward the beginning of that example. The first two are forfile://, and the third is the first character of the directory path, /var/tmp/django_cache. If you're on Windows, put the drive letter after the file://, like this:

```
file://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 exists and is readable and writable 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.

Each cache value will be stored as a separate file whose contents are the cache data saved in a serialized ("pickled") format, using Python's pickle module. Each file's name is the cache key, escaped for safe filesystem use.

Local-Memory Caching

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 multi-process and thread-safe. To use it, set CACHE_BACKEND to "locmem:///". For example:

```
CACHE_BACKEND = 'locmem:///'
```

Note that each process will have its own private cache instance, which means no cross-process caching is possible. This obviously 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 CACHE_BACKEND like so:

```
CACHE_BACKEND = 'dummy:///'
```

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 a Python import path as the scheme portion (the part before the initial colon) of the CACHE_BACKEND URI, like so:

```
CACHE_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 easy to use.

CACHE_BACKEND Arguments

Each cache backend may take arguments. They're given in query-string style on the CACHE_BACKEND 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).
- max_entries: For the locmem, filesystem and database backends, the maximum number of entries allowed in the cache before old values are deleted. This argument defaults to 300.
- cull_percentage: The percentage of entries that are culled when max_entries is reached. The actual ratio is 1/cull_percentage, so set cull_percentage=2 to cull half of the entries when max_entries is reached.

A value of 0 for cull_percentage means that the entire cache will be dumped when max_entries is reached. This makes culling *much* faster at the expense of more cache misses.

In this example, timeout is set to 60:

```
CACHE_BACKEND =
"memcached://127.0.0.1:11211/?timeout=60"
```

In this example, timeout is 30 and max_entries is 400:

```
CACHE_BACKEND = "locmem:///?timeout=30&max_entries=400"
```

Invalid arguments are silently ignored, as are invalid values of known arguments.

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_CLASSES setting, as in this example:

```
MIDDLEWARE_CLASSES = (

    '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_CLASSES below if you'd like the full story.

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

- CACHE_MIDDLEWARE_SECONDS – The 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.

The cache middleware caches every page that doesn't have GET or POST parameters. Optionally, if theCACHE_MIDDLEWARE_ANONYMOUS_ONLY setting is True, only anonymous requests (i.e., not those made by a logged-in user) will be cached. This is a simple and effective way of disabling caching for any user-specific pages (include Django's admin interface). Note that if you use CACHE_MIDDLEWARE_ANONYMOUS_ONLY, you should make sure you've activated AuthenticationMiddleware.

Additionally, the cache middleware automatically sets a few headers in each HttpResponse:

- Sets the Last-Modified header to the current date/time when a fresh (uncached) version of the page is requested.
- 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 theCACHE_MIDDLEWARE_SECONDS setting.

See Chapter 17 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 indjango.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 below for more on these decorators.

The Per-View Cache

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. It's easy to use:

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


def my_view(request):

    # ...
```

```
my_view = cache_page(my_view, 60 * 15)
```

Or, using Python 2.4's decorator syntax:

```
@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 themy_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 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 = ('',

    (r'^foo/(\d{1,2})/$', 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.

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_pagealters 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.

Doing so is easy: simply wrap the view function with cache_page when you refer to it in the URLconf. Here's the old URLconf from earlier:

```
urlpatterns = ('',

    (r'^foo/(\d{1,2})/$', my_view),

)
```

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

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


urlpatterns = ('',
    (r'^foo/(\d{1,2})/$', cache_page(my_view, 60 *
15)),
)
```

If you take this approach, don't forget to import cache_page within your URLconf.

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. 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 additional arguments 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 %}
```

It's perfectly fine to specify more than one argument to identify the fragment. Simply pass as many arguments to {% cache %} as you need.

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 just reuse that value.

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 simple, 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.)

The cache module, django.core.cache, has a cache object that's automatically created from theCACHE_BACKEND setting:

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

The basic interface is set(key, value, timeout_seconds) and get(key):

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

>>> cache.get('my_key')

'hello, world!'
```

The timeout_seconds argument is optional and defaults to the timeout argument in the CACHE_BACKENDsetting (explained above).

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
```

We advise against storing the literal value None in the cache, because you won't be able to distinguish between your stored None value and a cache miss signified by a return value of None.

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'
```

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.

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}
```

Finally, you can delete keys explicitly with delete(). This is an easy way of clearing the cache for a particular object:

```
>>> cache.delete('a')
```

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 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.

Upstream Caches

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

Here are a few examples of upstream caches:

- 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.
- Your Django Web site 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.

Upstream 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, say you operate a Web e-mail system, and the contents of the "inbox" page obviously 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 his 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 upstream 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 path (e.g.,"/stories/2005/jun/23/bank_robbed/"). 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 vary_on_headers view decorator, like so:

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


# Python 2.3 syntax.

def my_view(request):

    # ...

my_view = vary_on_headers(my_view, 'User-Agent')


# Python 2.4+ decorator syntax.

@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['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 upstream 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 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.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.

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: his or her own browser cache (a private cache) and his or her 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_controlview 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.

There are a few other ways to control cache parameters. For example, HTTP allows applications to do the following:

- Define the maximum time a page should be cached.
- Specify whether a cache should always check for newer versions, only delivering the cached content when there are no changes. (Some caches might deliver cached content even if the server page changed, simply because the cache copy isn't yet expired.)

In Django, use the cache_control view decorator to specify these cache parameters. In this example,cache_control tells caches to revalidate the cache on every access and to store cached versions for, at most, 3,600 seconds:

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


@cache_control(must_revalidate=True, max_age=3600)
def my_view(request):
    # ...
```

Any valid Cache-Control HTTP directive is valid in cache_control(). Here’s a full list:

- public=True
- private=True
- no_cache=True
- no_transform=True
- must_revalidate=True
- proxy_revalidate=True
- max_age=num_seconds
- s_maxage=num_seconds

(Note that the caching middleware already sets the cache header’s max-age with the value of theCACHE_MIDDLEWARE_SETTINGS setting. If you use a custom max_age in a cache_control decorator, the decorator will take precedence, and the header values will be merged correctly.)

If you want to use headers to disable caching altogether, django.views.decorators.cache.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):
    # ...
```

Other Optimizations

Django comes with a few other pieces of middleware that can help optimize your apps’ performance:

- django.middleware.http.ConditionalGetMiddleware adds support for modern browsers to conditionally GET responses based on the ETag and Last-Modified headers.

- django.middleware.gzip.GZipMiddleware compresses responses for all moderns browsers, saving bandwidth and transfer time.

Order of MIDDLEWARE_CLASSES

If you use caching middleware, it’s important to put each half in the right place within theMIDDLEWARE_CLASSES 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 thatUpdateCacheMiddleware 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. TheFetchFromCacheMiddleware also needs to run after other middleware updates the Vary header, soFetchFromCacheMiddleware must be *after* any item that does so.