

# How to implement Caching

- The per-site cache - Once the cache is set up, the simplest way to use caching is to cache your entire site.
- The per-view cache - A more granular way to use the caching framework is by caching the output of individual views.
- Template fragment caching – This gives you more control what to cache.

# Template Fragment Caching

This gives you more control what to cache.

{% load cache %} – This gives access to cache tag in template.

{% cache %} - This template tag caches the contents of the block for a given amount of time.

Syntax:-

```
{% cache timeout name variable using=" " %} ..... {% endcache name %}
```

Timeout - The cache timeout, in seconds. The fragment is cached forever if timeout is None. It can be a template variable, as long as the template variable resolves to an integer value.

Name - The name to give the cache fragment. The name will be taken as is, do not use a variable.

Variable – This can be a variable with or without filters. This will cache multiple copies of a fragment depending on some dynamic data that appears inside the fragment.

using - The cache tag will try to use the given cache. 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.

# Template Fragment Caching


Example:-

```
{% load cache %}  
{% cache 60 sidebar request.user.username using="localcache" %}  
    .. sidebar for logged in user ..  
{% endcache %}
```

# Database Caching

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

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



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.

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.

# Cache Arguments

Each cache backend can be given additional arguments to control caching behavior.

**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  $1 / \text{CULL\_FREQUENCY}$ , 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.

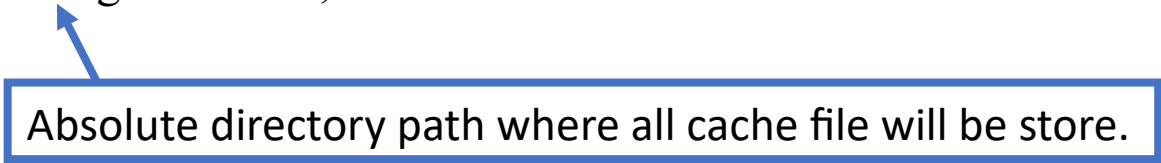
# Cache Arguments

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',  
        'LOCATION': 'c:\Djangocode\gs85\cache',  
        'TIMEOUT': 60,  
        'OPTIONS': {  
            'MAX_ENTRIES': 1000  
        }  
    }  
}
```

# Filesystem Caching

The file-based backend serializes and stores each cache value as a separate file.

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',  
        'LOCATION': 'c:\Djangocode\gs85\cache',  
    }  
}
```



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.

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

# Local Memory Caching

This is the default cache if another is not specified in your settings file. This cache is per-process and thread-safe.

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.

It's probably not a good choice for production environments. It's nice for development.

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



The cache LOCATION is used to identify individual memory stores.



# Dummy Caching

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.

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