

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.

The per-view cache

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. If multiple URLs point at the same view, each URL will be cached separately.

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

```
@cache_page(timeout, cache, key_prefix)
```

```
def my_view(request):
```

```
@cache_page(60, cache="some_cache", key_prefix="some_key")
```

```
def home(request):
```

`timeout` - The cache timeout, in seconds.

`cache` – This directs the decorator to use a specific cache (from your `CACHES` setting) when caching view results. By default, the default cache will be used.

`Key_prefix` - You can also override the cache prefix on a per-view basis. It works in the same way as the `CACHE_MIDDLEWARE_KEY_PREFIX` setting for the middleware.

The per-view cache

Specifying per-view cache in the URLconf

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


urlpatterns = [
    path('route/', cache_page(timeout, cache, key_prefix)(view_function)),
]
```

```
urlpatterns = [
    path('home/', cache_page(60)(views.home), name="home"),
]
```

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.

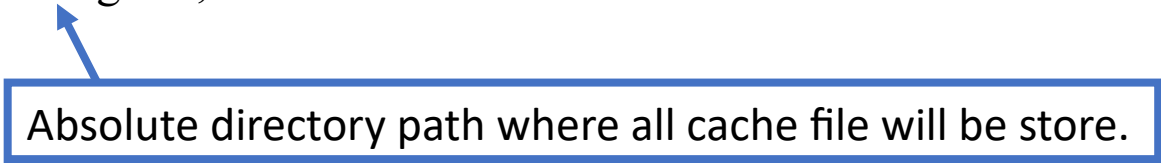
Cache Arguments

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',  
        'LOCATION': 'c:/Django code/gs80',  
        '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/gs80',  
    }  
}
```



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',  
    }  
}
```