# View decorators

Django provides several decorators that can be applied to views to support various HTTP features.

Allowed HTTP methods --

The decorators in django.views.decorators.http can be used to restrict access to views based on the request method. These decorators will return a django.http.HttpResponseNotAllowed if the conditions are

not met.

require\_http\_methods(request\_method\_list)

Decorator to require that a view only accepts particular request methods. Usage:

from django.views.decorators.http import require\_http\_methods

@require\_http\_methods(["GET", "POST"])

def my\_view(request):

# I can assume now that only GET or POST requests make it this far

# ...

pass

Note that request methods should be in uppercase.

require\_GET()

Decorator to require that a view only accepts the GET method.

require\_POST()

Decorator to require that a view only accepts the POST method.

require\_safe()

Decorator to require that a view only accepts the GET and HEAD methods. These methods are commonly considered “safe” because they should not have the significance of taking an action other than

retrieving the requested resource.

# Conditional view processing ---

The following decorators in django.views.decorators.http can be used to control caching behavior on

particular views.

condition(etag\_func=None, last\_modified\_func=None)

conditional\_page()

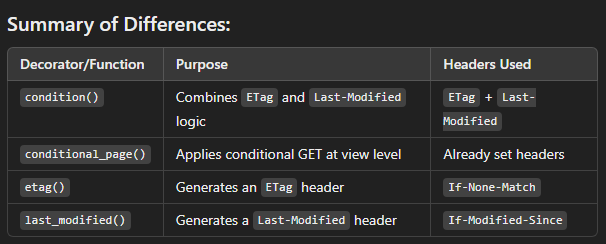
This decorator provides the conditional GET operation handling of ConditionalGetMiddleware to a

view.

etag(etag\_func)

last\_modified(last\_modified\_func)

These decorators can be used to generate ETag and Last-Modified headers .



from django.views.decorators.http import condition

from datetime import datetime

def my\_etag(request, \*args, \*\*kwargs):

return "etag-value" # Custom ETag value

def my\_last\_modified(request, \*args, \*\*kwargs):

return datetime(2024, 1, 1, 12, 0, 0) # Fixed last modified timestamp

@condition(etag\_func=my\_etag, last\_modified\_func=my\_last\_modified)

def my\_view(request):

# View logic here

return HttpResponse("This is a response")

# GZip compression --

The decorators in django.views.decorators.gzip control content compression on a per-view basis.

gzip\_page()

This decorator compress content if the browser allows gzip compression. It sets the Vary header

accordingly, so that caches will base their storage on the Accept-Encoding header.1. **Compression**:

* When a browser makes a request to a Django view, it sends an Accept-Encoding header indicating the types of compression it supports (e.g., gzip, deflate, etc.).
* If the browser supports gzip compression (as indicated by Accept-Encoding: gzip), the gzip\_page() decorator compresses the response using gzip before sending it to the client.
* This significantly reduces the size of the response, especially for large text-based content like HTML, CSS, or JSON.

2. **Vary Header**:

* The decorator sets the Vary: Accept-Encoding header in the response.
* This tells proxies and caches to store separate versions of the resource based on the Accept-Encoding header (i.e., one compressed version for gzip-supported clients and one uncompressed version for others).

**Use Case**

* Use gzip\_page() to compress content when the response size is large, and the browser supports gzip.
* Commonly applied to views that return:
  + HTML templates
  + JSON or XML responses
  + Large text-based content

from django.views.decorators.gzip import gzip\_page

from django.http import HttpResponse

@gzip\_page

def my\_view(request):

# Return some content

response\_content = "This is a large amount of text data that will be compressed."

return HttpResponse(response\_content)

1. **Request with gzip support**:

* Browser sends Accept-Encoding: gzip in its headers.
* The response will be compressed, and the browser will decompress it.

2. **Request without gzip support**:

* Browser does not send Accept-Encoding: gzip.
* The response will not be compressed.

# Vary headers ---

The decorators in django.views.decorators.vary can be used to control caching based on specific request

headers.

vary\_on\_cookie(func)

vary\_on\_headers(\*headers)

The Vary header defines which request headers a cache mechanism should consider when building its cache key.**1. vary\_on\_cookie(func)**

This decorator tells caches to vary the response based on the **cookies** sent in the request. It adds the **Vary: Cookie** header to the response.

**Use Case**

* Use this decorator when the response depends on the presence or value of cookies. For example:
  + When serving different content for authenticated vs. unauthenticated users (since the sessionid cookie determines if the user is logged in).
  + When displaying user-specific content based on cookies.

from django.views.decorators.vary import vary\_on\_cookie

from django.http import HttpResponse

@vary\_on\_cookie

def my\_view(request):

if request.COOKIES.get('theme') == 'dark':

return HttpResponse("This is the dark theme.")

else:

return HttpResponse("This is the light theme.")

* **What Happens**:
  + The response includes the Vary: Cookie header.
  + Caches will now consider the Cookie request header when determining if a cached response is valid.

**2. vary\_on\_headers(\*headers)**

This decorator tells caches to vary the response based on specific **request headers**. It allows you to specify which headers to include in the **Vary** response header.

**Use Case**

* Use this when the response depends on specific request headers, such as:
  + **Accept-Language**: When serving content in different languages.
  + **User-Agent**: When serving different content based on the client's device type.
  + Custom headers sent by the client.

from django.views.decorators.vary import vary\_on\_headers

from django.http import HttpResponse

@vary\_on\_headers('User-Agent')

def my\_view(request):

user\_agent = request.META.get('HTTP\_USER\_AGENT', '')

if 'Mobile' in user\_agent:

return HttpResponse("This is the mobile version.")

else:

return HttpResponse("This is the desktop version.")

**What Happens**:

* The response includes the Vary: User-Agent header.
* Caches will now consider the User-Agent request header when determining if a cached response is valid.

# Caching ---

The decorators in django.views.decorators.cache control server and client-side caching.

cache\_control(\*\*kwargs)

This decorator patch the response’s Cache-Control header by adding all of the keyword arguments

to it.

never\_cache(view\_func)

This decorator adds an Expires header to the current date/time.

This decorator adds a Cache-Control: max-age=0, no-cache, no-store, must-revalidate,

private header to a response to indicate that a page should never be cached.

Each header is only added if it isn’t already set.**1. cache\_control(\*\*kwargs)**

**What It Does:**

This decorator modifies the **Cache-Control** header in the HTTP response by adding specific instructions through keyword arguments (**kwargs**).

The Cache-Control header is used by browsers and caching servers to determine how long and under what conditions a response should be cached.

**How It Works:**

* It **adds or updates caching instructions** in the Cache-Control header of the HTTP response.
* A screenshot of a black and white page

  Description automatically generatedThe decorator leverages Django's patch\_cache\_control() utility to apply these settings.

from django.views.decorators.cache import cache\_control

from django.http import HttpResponse

@cache\_control(max\_age=3600, public=True)

def my\_view(request):

return HttpResponse("This response will be cached for 1 hour (3600 seconds).")

**2. never\_cache(view\_func)**

**What It Does:**

This decorator ensures that the response for the decorated view is **never cached** by any client (browser, proxy server, CDN, etc.).

It does this by setting the following headers in the HTTP response:

1. **Cache-Control: max-age=0, no-cache, no-store, must-revalidate, private**
   * Prevents all forms of caching.
2. **Expires** (current date/time):
   * Ensures older systems that rely on the Expires header won’t cache the response.

Each of these headers is only added if it’s not already set in the response.

from django.views.decorators.cache import never\_cache

from django.http import HttpResponse

@never\_cache

def my\_view(request):

return HttpResponse("This response will never be cached.")

Common - -

The decorators in django.views.decorators.common allow per-view customization of CommonMiddleware

Behavior.

no\_append\_slash()

This decorator allows individual views to be excluded from APPEND\_SLASH URL normalization.

**What Does It Do?**

The no\_append\_slash() decorator **excludes individual views** from the URL normalization behavior of the CommonMiddleware when APPEND\_SLASH=True.

In simple words, if Django is set to automatically add a **trailing slash (/)** to URLs (like turning /about into /about/), this decorator **stops that behavior for specific views**.

**Why Is This Needed?**

* Django's CommonMiddleware helps users by redirecting requests without trailing slashes to their proper URL with a slash if APPEND\_SLASH=True.
* This is great for most cases, but sometimes you may have specific views where **you don’t want the slash added**. For example:
  + An API endpoint where the trailing slash might cause issues.
  + Legacy systems that require URLs without trailing slashes.

from django.views.decorators.common import no\_append\_slash

from django.http import HttpResponse

@no\_append\_slash

def my\_view(request):

return HttpResponse("This view does NOT append a slash to the URL.")

----------------------- END ----------------------------------------------------------------------------------------