Middleware

Middleware is a framework of hooks into Django’s request/response processing. It’s a light, low-level “plugin”

system for globally altering Django’s input or output.

Each middleware component is responsible for doing some specific function. For example, Django includes

a middleware component, ***AuthenticationMiddleware***, that associates users with requests using sessions.

This document explains how middleware works, how you activate middleware, and how to write your own

middleware. Django ships with some built-in middleware you can use right out of the box.

**Writing your own middleware**

A middleware factory is a callable that takes a **get\_response** callable and returns a middleware. A middleware is a callable that takes a request and returns a response, just like a view.

***A middleware can be written as a function that looks like this:***

**def simple\_middleware(get\_response):**

**# One-time configuration and initialization.**

**def middleware(request):**

**# Code to be executed for each request before**

**# The view (and later middleware) is called.**

**response = get\_response(request)**

**# Code to be executed for each request/response after**

**# The view is called.**

**return response**

**return middleware**

Or it can be written as a class whose instances are callable, like this:**class SimpleMiddleware:**

**def \_\_init\_\_(self, get\_response):**

**self.get\_response = get\_response**

**# One-time configuration and initialization.**

**def \_\_call\_\_(self, request):**

**# Code to be executed for each request before**

**# The view (and later middleware) is called.**

**response = self.get\_response(request)**

**# Code to be executed for each request/response after**

**# The view is called.**

**return response**The get\_response callable provided by Django might be the actual view (if this is the last listed middleware)

or it might be the next middleware in the chain. The current middleware doesn’t need to know or care what

exactly it is, just that it represents whatever comes next

The above is a slight simplification – the get\_response callable for the last middleware in the chain won’t be the actual view but rather a wrapper method from the handler which takes care of applying view middleware, calling the view with appropriate URL arguments, and applying template-response and exception

middleware.

Middleware can either support only synchronous Python (the default), only asynchronous Python, or both.

Middleware can live anywhere on your Python path.

**\_\_init\_\_(get\_response)**

Middleware factories must accept a get\_response argument. You can also initialize some global state for

the middleware. Keep in mind a couple of caveats:

• Django initializes your middleware with only the get\_response argument, so you can’t define

**\_\_init\_\_()** as requiring any other arguments.

• Unlike the **\_\_call\_\_()** method, which is called once per request, **\_\_init\_\_()** is called only once, when the web server starts.

## Marking middleware as unused

It’s sometimes useful to determine at startup time whether a piece of middleware should be used. In these

cases, your middleware’s ***\_\_init\_\_()*** method may raise ***MiddlewareNotUsed***. Django will then remove that

middleware from the middleware process and log a debug message to the ***django.request*** logger when DEBUG is True.

Activating middleware

To activate a middleware component, add it to the MIDDLEWARE list in your Django settings.

In MIDDLEWARE, each middleware component is represented by a string: the full Python path to the middleware factory’s class or function name. For example, here’s the default value created by django-admin

startproject:

**MIDDLEWARE = [**

**"django.middleware.security.SecurityMiddleware",**

**"django.contrib.sessions.middleware.SessionMiddleware",**

**"django.middleware.common.CommonMiddleware",**

**"django.middleware.csrf.CsrfViewMiddleware",**

**"django.contrib.auth.middleware.AuthenticationMiddleware",**

**"django.contrib.messages.middleware.MessageMiddleware","django.middleware.clickjacking.XFrameOptionsMiddleware",**

**]**

A Django installation doesn’t require any middleware — MIDDLEWARE can be empty, if you’d like — but it’s

strongly suggested that you at least use **CommonMiddleware**.

The order in MIDDLEWARE matters because a middleware can depend on other middleware. For instance,

**AuthenticationMiddleware** stores the authenticated user in the session; therefore, it must run after

**SessionMiddleware**. See Middleware ordering for some common hints about ordering of Django **middleware** classes.

Middleware order and layering

During the request phase, before calling the view, Django applies middleware in the order it’s defined in

**MIDDLEWARE**, top-down.

You can think of it like an onion: each middleware class is a “layer” that wraps the view, which is in the core

of the onion. If the request passes through all the layers of the onion (each one calls get\_response to pass

the request into the next layer), all the way to the view at the core, the response will then pass through every

layer (in reverse order) on the way back out.

If one of the layers decides to short-circuit and return a response without ever calling its get\_response,

none of the layers of the onion inside that layer (including the view) will see the request or the response. The

response will only return through the same layers that the request passed in through.

# Other middleware hooks

Besides the basic request/response middleware pattern described earlier, you can add three other special

methods to class-based middleware:

# process\_view()

**process\_view(request, view\_func, view\_args, view\_kwargs)**

request is an **HttpRequest** object. **view\_func** is the Python function that Django is about to use. (It’s the

actual function object, not the name of the function as a string.) **view\_args** is a list of positional arguments

that will be passed to the view, and view\_kwargs is a dictionary of keyword arguments that will be passed

to the view. Neither **view\_args** nor **view\_kwargs** include the first view argument (request).

**process\_view()** is called just before Django calls the view.

It should return either None or an **HttpResponse** object. If it returns None, Django will continue processing

this request, executing any other process\_view() middleware and then, the appropriate view. If it returns an **HttpResponse** object, Django won’t bother calling the appropriate view; it’ll apply response middleware

to that **HttpResponse** and return the result.

Note: Accessing **request.POST** inside middleware before the view runs or in **process\_view()** will prevent any view running after the middleware from being able to modify the upload handlers for the request and should normally be avoided.

The **CsrfViewMiddleware** class can be considered an exception, as it provides the **csrf\_exempt()** and **csrf\_protect()** decorators which allow views to explicitly control at what point the CSRF validation should occur.

# process\_exception()

**process\_exception(request, exception)**

request is an **HttpRequest** object. **exception** is an Exception object raised by the view function.

Django calls **process\_exception()** when a view raises an exception**. process\_exception()** should return either **None** or an **HttpResponse** object. If it returns an **HttpResponse** object, the **template response** and **response middleware** will be applied and the resulting response returned to the browser. Otherwise, default exception handling kicks in. Again, middleware is run in reverse order during the response phase, which includes **process\_exception**. If an exception middleware returns a response, the **process\_exception** methods of the middleware classes above that middleware won’t be called at all.

# process\_template\_response()

**process\_template\_response(request, response)**

request is an **HttpRequest** object. response is the **TemplateResponse** object (or equivalent) returned by a Django view or by a middleware. **process\_template\_response()** is called just after the view has finished executing, if the response instance has a **render()** method, indicating that it is a **TemplateResponse** or equivalent.

It must return a response object that implements a render method. It could alter the given response by

changing **response.template\_name** and **response.context\_data**, or it could create and return a brandnew **TemplateResponse** or equivalent.

You don’t need to explicitly render responses – responses will be automatically rendered once all template

response middleware has been called. Middleware are run in reverse order during the response phase, which includes **process\_template\_response().**

**from django.http import HttpResponse**

**class CustomMiddleware:**

**def \_\_init\_\_(self, get\_response):**

**self.get\_response = get\_response**

**def \_\_call\_\_(self, request):**

**# Code executed for each request before the view is called**

**response = self.get\_response(request)**

**# Code executed for each request/response after the view is called**

**return response**

**def process\_view(self, request, view\_func, view\_args, view\_kwargs):**

**"""**

**Called just before Django calls the view.**

**Logs the view being called and its arguments.**

**"""**

**print(f"CustomMiddleware: process\_view called for view {view\_func.\_\_name\_\_}")**

**print(f"View args: {view\_args}")**

**print(f"View kwargs: {view\_kwargs}")**

**# You can return None to continue processing, or an HttpResponse to short-circuit.**

**return None**

**def process\_exception(self, request, exception):**

**"""**

**Called when a view raises an exception.**

**Logs the exception and returns a custom error response.**

**"""**

**print(f"CustomMiddleware: process\_exception called with exception: {exception}")**

**# Optionally return an HttpResponse to handle the exception.**

**return HttpResponse("An error occurred: " + str(exception), status=500)**

**def process\_template\_response(self, request, response):**

**"""**

**Called just after the view has finished executing if the response is a TemplateResponse.**

**Logs the response and optionally modifies it.**

**"""**

**print("CustomMiddleware: process\_template\_response called")**

**if hasattr(response, 'template\_name'):**

**print(f"Template name: {response.template\_name}")**

**response.context\_data['middleware\_message'] = "This is a message from CustomMiddleware."**

**return response**

Dealing with streaming responses

Unlike **HttpResponse**, **StreamingHttpResponse** does not have a content attribute. As a result, middleware can no longer assume that all responses will have a content attribute. If they need access to the content, they must test for streaming responses and adjust their behavior accordingly:

**if response.streaming:**

**response.streaming\_content = wrap\_streaming\_content(response.streaming\_content)**

**else:**

**response.content = alter\_content(response.content)**

Note: **streaming\_content** should be assumed to be too large to hold in memory. Response middleware may wrap it in a new generator but must not consume it. Wrapping is typically implemented as follows:

**def wrap\_streaming\_content(content):**

**for chunk in content:**

**yield alter\_content(chunk)**

***StreamingHttpResponse*** allows both ***synchronous*** and ***asynchronous*** iterators. The wrapping function must

match. Check **StreamingHttpResponse.is\_async** if your middleware needs to support both types of iterators.

# Exception handling

Django automatically converts exceptions raised by the view or by middleware into an appropriate **HTTP**

response with an error status code. Certain exceptions are converted to 4xx status codes, while an unknown

exception is converted to a **500**-**status** code.

This conversion takes place before and after each middleware (you can think of it as the thin film in between

each layer of the onion), so that every middleware can always rely on getting some kind of **HTTP** response

back from calling its **get\_response** callable. Middleware don’t need to worry about wrapping their call to

**get\_response** in a try/except and handling an exception that might have been raised by a later middleware

or the view. Even if the very next middleware in the chain raises an **Http404** exception, for example, your

middleware won’t see that exception; instead, it will get an **HttpResponse** object with a status\_code of 404.

You can **set DEBUG\_PROPAGATE\_EXCEPTIONS** to **True** to skip this conversion and propagate exceptions upward.

Asynchronous support - -

Middleware can support any combination of synchronous and asynchronous requests. Django will adapt

requests to fit the middleware’s requirements if it cannot support both, but at a performance penalty.

By default, Django assumes that your middleware is capable of handling only synchronous requests. To

change these assumptions, set the following attributes on your middleware factory function or class:

• **sync\_capable** is a **boolean** indicating if the middleware can handle synchronous requests. Defaults to True.

• **async\_capable** is a **boolean** indicating if the middleware can handle asynchronous requests. Defaults to False.

If your middleware has both **sync\_capable = True** and **async\_capable = True**, then Django will pass the request without converting it. In this case, you can work out if your middleware will receive async

requests by checking if the **get\_response** object you are passed is a coroutine function, using

**asgiref.sync.iscoroutinefunction.**

The **django.utils.decorators** module contains **sync\_only\_middleware(),** **async\_only\_middleware(),**and **sync\_and\_async\_middleware()** decorators that allow you to apply these flags to middleware factory functions.

The returned callable must match the sync or async nature of the get\_response method. If you have an

asynchronous **get\_response**, you must return a coroutine function (**async def**).

**process\_view** , **process\_template\_response** and **process\_exception** methods, if they are provided, should also be adapted to match the **sync/async mode**. However, Django will individually adapt them as required if you do not, at an additional performance penalty.

from django.http import HttpResponse

from asgiref.sync import iscoroutinefunction

class CustomMiddleware:

sync\_capable = True

async\_capable = True

def \_\_init\_\_(self, get\_response):

self.get\_response = get\_response

self.is\_async = iscoroutinefunction(get\_response)

async def \_\_call\_\_(self, request):

if self.is\_async:

# Handle asynchronous requests

response = await self.get\_response(request)

else:

# Handle synchronous requests

response = self.get\_response(request)

return response

async def process\_view(self, request, view\_func, view\_args, view\_kwargs):

"""

Called just before Django calls the view.

Logs the view being called and its arguments.

"""

print(f"CustomMiddleware: process\_view called for view {view\_func.\_\_name\_\_}")

print(f"View args: {view\_args}")

print(f"View kwargs: {view\_kwargs}")

# You can return None to continue processing, or an HttpResponse to short-circuit.

return None

async def process\_exception(self, request, exception):

"""

Called when a view raises an exception.

Logs the exception and returns a custom error response.

"""

print(f"CustomMiddleware: process\_exception called with exception: {exception}")

# Optionally return an HttpResponse to handle the exception.

return HttpResponse("An error occurred: " + str(exception), status=500)

async def process\_template\_response(self, request, response):

"""

Called just after the view has finished executing if the response is a TemplateResponse.

Logs the response and optionally modifies it.

"""

print("CustomMiddleware: process\_template\_response called")

if hasattr(response, 'template\_name'):

print(f"Template name: {response.template\_name}")

response.context\_data['middleware\_message'] = "This is a message from CustomMiddleware."

return response

# EXAMPLE ------------------------------------------------

Here’s an example of how to create a middleware function that supports both:

**from asgiref.sync import iscoroutinefunction**

**from django.utils.decorators import sync\_and\_async\_middleware**

**@sync\_and\_async\_middleware**

**def simple\_middleware(get\_response):**

**# One-time configuration and initialization goes here.**

**if iscoroutinefunction(get\_response):**

**async def middleware(request):**

**# Do something here!**

**response = await get\_response(request)**

**return response**

**else: def middleware(request):**

**# Do something here!**

**response = get\_response(request)**

**return response**

**return middleware**

***Note***: If you declare a hybrid middleware that supports both ***synchronous*** and ***asynchronous*** calls, the kind of call you get may not match the underlying view. Django will optimize the middleware call stack to have as few sync/async transitions as possible.

Thus, even if you are wrapping an async view, you may be called in sync mode if there is other, synchronous

middleware between you and the view

When using an asynchronous class-based middleware, you must ensure that instances are correctly marked

as coroutine functions:

**from asgiref.sync import iscoroutinefunction, markcoroutinefunction**

**class AsyncMiddleware:**

**async\_capable = True**

**sync\_capable = False**

**def \_\_init\_\_(self, get\_response):**

**self.get\_response = get\_response**

**if iscoroutinefunction(self.get\_response):**

**markcoroutinefunction(self)**

**async def \_\_call\_\_(self, request):**

**response = await self.get\_response(request)**

**# Some logic ...**

**return response**

### Upgrading pre-Django 1.10-style middleware

**class django.utils.deprecation.MiddlewareMixin**

Django provides **django.utils.deprecation.MiddlewareMixin** to ease creating middleware classes that

are compatible with both **MIDDLEWARE** and the old **MIDDLEWARE\_CLASSES**, and support synchronous and **asyn**c**hronous** requests. All middleware classes included with Django are compatible with both settings.

The mixin provides an **\_\_init\_\_()** method that requires a get\_response argument and stores it in self.

**get\_response**.

The **\_\_call\_\_()** method:

1. Calls **self.process\_request(request**) (if defined).

2. Calls **self.get\_response(request**) to get the response from later middleware and the view.

3. Calls **self.process\_response**(request, response) (if defined).

4. Returns the response.

If used with **MIDDLEWARE\_CLASSES**, the **\_\_call\_\_()** method will never be used; Django calls

**process\_request()** and **process\_response()** directly.

In most cases, inheriting from this mixin will be sufficient to make an old-style middleware compatible with

the new system with sufficient backwards-compatibility. The new short-circuiting semantics will be harmless

or even beneficial to the existing middleware. In a few cases, a middleware class may need some changes to

adjust to the new semantics.

These are the behavioral differences between using **MIDDLEWARE** and **MIDDLEWARE\_CLASSES**:

1. Under **MIDDLEWARE\_CLASSES**, every middleware will always have its **process\_response** method called,

even if an earlier middleware short-circuited by returning a response from its **process\_request**

method. Under MIDDLEWARE, middleware behaves more like an onion: the layers that a response goes

through on the way out are the same layers that saw the request on the way in. If a middleware

**short-circuits**, only that middleware and the ones before it in MIDDLEWARE will see the response.

2. Under **MIDDLEWARE\_CLASSES**, **process\_exception** is applied to exceptions raised from a middleware

**process\_request** method. Under MIDDLEWARE, process\_exception applies only to exceptions raised

from the view (or from the render method of a TemplateResponse). Exceptions raised from a **middleware** are converted to the appropriate HTTP response and then passed to the next middleware.

3. Under **MIDDLEWARE\_CLASSES**, if a **process\_response** method raises an exception, the

**process\_response** methods of all earlier middleware are skipped and a **500** Internal Server

Error **HTTP** response is always returned (even if the exception raised was e.g. an **Http404**). Under

**MIDDLEWARE**, an **exception** raised from a **middleware** will immediately be converted to the appropriate

**HTTP** **response**, and then the next middleware in line will see that response. Middleware are never

skipped due to a middleware raising an exception.

from django.utils.deprecation import MiddlewareMixin

from django.http import HttpResponse

class CustomMiddleware(MiddlewareMixin):

def process\_request(self, request):

"""

Process the request before it reaches the view.

Logs the request and optionally modifies it.

"""

print("CustomMiddleware: process\_request called")

# You can modify the request here if needed.

# Return None to continue processing, or an HttpResponse to short-circuit.

return None

def process\_response(self, request, response):

"""

Process the response before it is sent to the client.

Logs the response and optionally modifies it.

"""

print("CustomMiddleware: process\_response called")

# You can modify the response here if needed.

return response

def process\_exception(self, request, exception):

"""

Process any exceptions raised during the view.

Logs the exception and returns a custom error response if necessary.

"""

print(f"CustomMiddleware: process\_exception called with exception: {exception}")

# Optionally return an HttpResponse to handle the exception.

return HttpResponse("An error occurred: " + str(exception), status=500)

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