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. They're documented in the built-in middleware reference.

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) are 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) are 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. See Asynchronous support for details of how to advertise what you support, and know what kind of request you are getting.

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 in to 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 are 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 brand-new `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()`.

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 iterator.

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 it 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.

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 asynchronous 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.