Chapter 17: Middleware

On occasion, you'll need to run a piece of code on each and every request that Django handles. This code might need to modify the request before the view handles it, it might need to log information about the request for debugging purposes, and so forth.

You can do this with Django's *middleware* framework, which is a set of hooks into Django's request/response processing. It's a light, low-level "plug-in" system capable of globally altering both Django's input and output.

Each middleware component is responsible for doing some specific function. If you're reading this book straight through, you've seen middleware a number of times already:

- All of the session and user tools that we looked at in Chapter 14 are made possible by a few small pieces of middleware (more specifically, the middleware makes `request.session` and `request.user`available to you in views).
- The sitewide cache discussed in Chapter 15 is actually just a piece of middleware that bypasses the call to your view function if the response for that view has already been cached.
- The `flatpages`, `redirects`, and `csrf` applications from Chapter 16 all do their magic through middleware components.

This chapter dives deeper into exactly what middleware is and how it works, and explains how you can write your own middleware.

What's Middleware?

Let's start with a very simple example.

High-traffic sites often need to deploy Django behind a load-balancing proxy (see Chapter 12). This can cause a few small complications, one of which is that every request's remote IP (`request.META["REMOTE_IP"]`) will be that of the load balancer, not the actual IP making the request. Load balancers deal with this by setting a special header, `X-Forwarded-For`, to the actual requesting IP address.

So here's a small bit of middleware that lets sites running behind a proxy still see the correct IP address in`request.META["REMOTE_ADDR"]`:

```
class SetRemoteAddrFromForwardedFor(object):

    def process_request(self, request):

        try:

            real_ip =
request.META['HTTP_X_FORWARDED_FOR']

        except KeyError:

            pass
        else:
            # HTTP_X_FORWARDED_FOR can be a comma-
separated list of IPs.
            # Take just the first one.
            real_ip = real_ip.split(",")[0]
            request.META['REMOTE_ADDR'] = real_ip
```

(Note: Although the HTTP header is called `X-Forwarded-For`, Django makes it available as`request.META['HTTP_X_FORWARDED_FOR']`. With the exception of `content-length` and `content-type`, any HTTP headers in the request are converted to `request.META` keys by converting all characters to uppercase, replacing any hyphens with underscores and adding an `HTTP_` prefix to the name.)

If this middleware is installed (see the next section), every request's `X-Forwarded-For` value will be automatically inserted into `request.META['REMOTE_ADDR']`. This means your Django applications don't need to be concerned with whether they're behind a load-balancing proxy or not; they can simply access`request.META['REMOTE_ADDR']`, and that will work whether or not a proxy is being used.

In fact, this is a common enough need that this piece of middleware is a built-in part of Django. It lives in`django.middleware.http`, and you can read a bit more about it later in this chapter.

Middleware Installation

If you've read this book straight through, you've already seen a number of examples of middleware installation; many of the examples in previous chapters have required certain middleware. For completeness, here's how to install middleware.

To activate a middleware component, add it to the `MIDDLEWARE_CLASSES` tuple in your settings module. In`MIDDLEWARE_CLASSES`, each middleware component is represented by a string: the full Python path to the middleware's class name. For example, here's the default `MIDDLEWARE_CLASSES` created by`django-admin.py startproject`:

```
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddlewar
e',
)
```

A Django installation doesn't require any middleware – `MIDDLEWARE_CLASSES` can be empty, if you'd like – but we recommend that you activate `CommonMiddleware`, which we explain shortly.

The order is significant. On the request and view phases, Django applies middleware in the order given in`MIDDLEWARE_CLASSES`, and on the response and exception phases, Django applies middleware in reverse order. That is, Django treats `MIDDLEWARE_CLASSES` as a sort of "wrapper" around the view function: on the request it walks down the list to the view, and on the response it walks back up.

Middleware Methods

Now that you know what middleware is and how to install it, let's take a look at all the available methods that middleware classes can define.

Initializer: __init__(self)

Use `__init__()` to perform systemwide setup for a given middleware class.

For performance reasons, each activated middleware class is instantiated only *once* per server process. This means that `__init__()` is called only once – at server startup – not for individual requests.

A common reason to implement an `__init__()` method is to check whether the middleware is indeed needed. If `__init__()` raises `django.core.exceptions.MiddlewareNotUsed`, then Django will remove the middleware from the middleware stack. You might use this feature to check for some piece of software that the middleware class requires, or check whether the server is running debug mode, or any other such environment situation.

If a middleware class defines an `__init__()` method, the method should take no arguments beyond the standard `self`.

Request Preprocessor: process_request(self, request)

This method gets called as soon as the request has been received – before Django has parsed the URL to determine which view to execute. It gets passed the `HttpRequest` object, which you may modify at will.

`process_request()` should return either `None` or an `HttpResponse` object.

- If it returns `None`, Django will continue processing this request, executing any other middleware and then the appropriate view.
- If it returns an `HttpResponse` object, Django won't bother calling *any* other middleware (of any type) or the appropriate view. Django will immediately return that `HttpResponse`.

View Preprocessor: process_view(self, request, view, args, kwargs)

This method gets called after the request preprocessor is called and Django has determined which view to execute, but before that view has actually been executed.

The arguments passed to this view are shown in Table 17-1.

Table 17-1. Arguments Passed to process_view()

Argument	Explanation
`request`	The `HttpRequest` object.
`view`	The Python function that Django will call to handle this request. This is the actual function object itself, not the name of the function as a string.
`args`	The list of positional arguments that will be passed to the view, not including the`request` argument (which is always the first argument to a view).
`kwargs`	The dictionary of keyword arguments that will be passed to the view.

Just like `process_request()`, `process_view()` should return either `None` or an `HttpResponse` object.

- If it returns `None`, Django will continue processing this request, executing any other middleware and then the appropriate view.
- If it returns an `HttpResponse` object, Django won't bother calling *any* other middleware (of any type) or the appropriate view. Django will immediately return that `HttpResponse`.

Response Postprocessor: process_response(self, request, response)

This method gets called after the view function is called and the response is generated. Here, the processor can modify the content of a response. One obvious use case is content compression, such as gzipping of the request's HTML.

The parameters should be pretty self-explanatory: `request` is the request object, and `response` is the response object returned from the view.

Unlike the request and view preprocessors, which may return `None`, `process_response()` *must* return an`HttpResponse` object. That response could be the original one passed into the function (possibly modified) or a brand-new one.

Exception Postprocessor: process_exception(self, request, exception)

This method gets called only if something goes wrong and a view raises an uncaught exception. You can use this hook to send error notifications, dump postmortem information to a log, or even try to recover from the error automatically.

The parameters to this function are the same `request` object we've been dealing with all along, and`exception`, which is the actual `Exception` object raised by the view function.

`process_exception()` should return a either `None` or an `HttpResponse` object.

- If it returns `None`, Django will continue processing this request with the framework's built-in exception handling.
- If it returns an `HttpResponse` object, Django will use that response instead of the framework's built-in exception handling.

Note

Django ships with a number of middleware classes (discussed in the following section) that make good examples. Reading the code for them should give you a good feel for the power of middleware.

You can also find a number of community-contributed examples on Django's wiki:http://code.djangoproject.com/wiki/ContributedMiddleware

Built-in Middleware

Django comes with some built-in middleware to deal with common problems, which we discuss in the sections that follow.

Authentication Support Middleware

Middleware class: `django.contrib.auth.middleware.AuthenticationMiddleware`.

This middleware enables authentication support. It adds the `request.user` attribute, representing the currently logged-in user, to every incoming `HttpRequest` object.

See Chapter 14 for complete details.

"Common" Middleware

Middleware class: `django.middleware.common.CommonMiddleware`.

This middleware adds a few conveniences for perfectionists:

- *Forbids access to user agents in the ``DISALLOWED_USER_AGENTS`` setting*: If provided, this setting should be a list of compiled regular expression objects that are matched against the user-agent header for each incoming request. Here's an example snippet from a settings file:

```
import re

DISALLOWED_USER_AGENTS = (
    re.compile(r'^OmniExplorer_Bot'),
    re.compile(r'^Googlebot')
)
```

  Note the `import re`, because `DISALLOWED_USER_AGENTS` requires its values to be compiled regexes (i.e., the output of `re.compile()`). The settings file is regular Python, so it's perfectly OK to include Python`import` statements in it.

- *Performs URL rewriting based on the ``APPEND_SLASH`` and ``PREPEND_WWW`` settings*: If`APPEND_SLASH` is `True`, URLs that lack a trailing slash will be redirected to the same URL with a trailing slash, unless the last component in the path contains a period. So `foo.com/bar` is redirected to`foo.com/bar/`, but `foo.com/bar/file.txt` is passed through unchanged.

If `PREPEND_WWW` is `True`, URLs that lack a leading “www.” will be redirected to the same URL with a leading “www.”.

Both of these options are meant to normalize URLs. The philosophy is that each URL should exist in one – and only one – place. Technically the URL `example.com/bar` is distinct from `example.com/bar/`, which in turn is distinct from `www.example.com/bar/`. A search-engine indexer would treat these as separate URLs, which is detrimental to your site’s search-engine rankings, so it’s a best practice to normalize URLs.

- *Handles ETags based on the ``USE_ETAGS`` setting*: *ETags* are an HTTP-level optimization for caching pages conditionally. If `USE_ETAGS` is set to `True`, Django will calculate an ETag for each request by MD5-hashing the page content, and it will take care of sending `Not Modified` responses, if appropriate.

  Note there is also a conditional `GET` middleware, covered shortly, which handles ETags and does a bit more.

Compression Middleware

Middleware class: `django.middleware.gzip.GZipMiddleware`.

This middleware automatically compresses content for browsers that understand gzip compression (all modern browsers). This can greatly reduce the amount of bandwidth a Web server consumes. The tradeoff is that it takes a bit of processing time to compress pages.

We usually prefer speed over bandwidth, but if you prefer the reverse, just enable this middleware.

Conditional GET Middleware

Middleware class: `django.middleware.http.ConditionalGetMiddleware`.

This middleware provides support for conditional `GET` operations. If the response has an `Last-Modified` or`ETag` or header, and the request has `If-None-Match` or `If-Modified-Since`, the response is replaced by an 304 (“Not modified”) response. `ETag` support depends on on the `USE_ETAGS` setting and expects the `ETag`response header to already be set. As discussed above, the `ETag` header is set by the Common middleware.

It also removes the content from any response to a `HEAD` request and sets the `Date` and `Content-Length`response headers for all requests.

Reverse Proxy Support (X-Forwarded-For Middleware)

Middleware class: `django.middleware.http.SetRemoteAddrFromForwardedFor`.

This is the example we examined in the “What’s Middleware?” section earlier. It sets`request.META['REMOTE_ADDR']` based on `request.META['HTTP_X_FORWARDED_FOR']`, if the latter is set. This is useful if you’re sitting behind a reverse proxy that causes each request’s `REMOTE_ADDR` to be set to`127.0.0.1`.

Danger!

This middleware does *not* validate `HTTP_X_FORWARDED_FOR`.

If you’re not behind a reverse proxy that sets `HTTP_X_FORWARDED_FOR` automatically, do not use this middleware. Anybody can spoof the value of `HTTP_X_FORWARDED_FOR`, and because this sets`REMOTE_ADDR` based on `HTTP_X_FORWARDED_FOR`, that means anybody can fake his IP address.

Only use this middleware when you can absolutely trust the value of `HTTP_X_FORWARDED_FOR`.

Session Support Middleware

Middleware class: `django.contrib.sessions.middleware.SessionMiddleware`.

This middleware enables session support. See Chapter 14 for details.

Sitewide Cache Middleware

Middleware
classes: `django.middleware.cache.UpdateCacheMiddleware` and`django.middleware.cache.FetchFromCacheMiddleware`.

These middlewares work together to cache each Django-powered page. This was discussed in detail in Chapter 15.

Transaction Middleware

Middleware class: `django.middleware.transaction.TransactionMiddleware`.

This middleware binds a database `COMMIT` or `ROLLBACK` to the request/response phase. If a view function runs successfully, a `COMMIT` is issued. If the view raises an exception, a `ROLLBACK` is issued.

The order of this middleware in the stack is important. Middleware modules running outside of it run with commit-on-save – the default Django behavior. Middleware modules running inside it (coming later in the stack) will be under the same transaction control as the view functions.

See Appendix B for more about information about database transactions.