***Web Server GateWay Interface.***

**Django Middlewares and the Request/Response Cycle**

When setting up a new Django project, one of the first things you’ll do is wire up your URLconfs and set up some views. But what’s actually happening under the hood here? How does Django route traffic to the view, and what part do middlewares play in this cycle?

**WSGI**

WSGI is a tool created to solve a basic problem: connecting a web server to a web framework. WSGI has two sides: the ‘server’ side and the ‘application’ side. To handle a WSGI response, the server executes the application and provides a callback function to the application side. The application processes the request and returns the response to the server using the provided callback. Essentially, the WSGI handler acts as the gatekeeper between your web server (Apache, NGINX, etc) and your Django project.

Between the server and the application lie the middlewares. You can think of middlewares as a series of bidirectional filters: they can alter (or short-circuit) the data flowing back and forth between the network and your Django application.

**The Big Picture — Data Flow**

When the user makes a request of your application, a WSGI handler is instantiated, which:

1. imports your settings.py file and Django’s exception classes.
2. loads all the middleware classes it finds in the MIDDLEWARE\_CLASSES or MIDDLEWARES(depending on Django version) tuple located in settings.py
3. builds four lists of methods which handle processing of request, view, response, and exception.
4. loops through the request methods, running them in order
5. resolves the requested URL
6. loops through each of the view processing methods
7. calls the view function (usually rendering a template)
8. processes any exception methods
9. loops through each of the response methods, (from the inside out, reverse order from request middlewares)
10. finally builds a return value and calls the callback function to the web server

Let’s get started.

**Middlewares**

Middlewares are employed in a number of key pieces of functionality in a Django project: you use CSRF middlewares to prevent cross-site request forgery attacks, for example. They’re used to handle session data. Authentication and authorization is accomplished with the use of middlewares. You can write your own middleware classes to shape (or short-circuit) the flow of data through your application.

**process\_request**

Django middlewares must have at least one of the following methods: process\_request, process\_response, process\_view, and process\_exception. These are the methods which will be collected by the WSGI Handler and then called in the order they are listed. Let’s take a quick look at django.contrib.auth.middleware.AuthenticationMiddleware, one of the middlewares which are installed by default when you run django-admin.py startproject:

As you can see, this middleware only works on the ‘request’ step of the data flow to and from your Django application. This middleware first verifies that the session middleware is in use and has been called already, then it sets the userby calling the get\_user helper function. As the WSGI Handler iterates through the list of process\_request methods, it’s building up this requestobject which will eventually be passed into the view, and you’ll be able to reference request.user. Some of the middlewares in settings.py won’t have process\_requestmethods. No big deal; those just get skipped during this stage.

process\_requestshould either return None (as in this example), or alternately it can return an HttpResponse object. In the former case, the WSGI Handler will continue processing the process\_request methods, the latter will “short-circuit” the process and begin the process\_response cycle.

**Resolve the URL**

Now that the process\_request methods have each been called, we now have a request object which will be passed to the view. Before that can happen, Django must resolve the URL and determine which view function to call. This is simply done by regular expression matching. Your settings.pywill have a key called ROOT\_URLCONF which indicates the ‘root’ urls.py file, from which you’ll include the urls.py files for each of your apps. URL routing is covered pretty extensively in the Django tutorials so there’s no need to go into it here.

A view has three requirements:

1. It must be callable. It can be a function-based view, or a class-based view which inherits from View the as\_view() method to make it callable depending on the HTTP verb (GET, POST, etc)
2. It must accept an HttpRequest object as the first positional argument. This is the result of calling all the process\_request and process\_viewmiddleware methods.
3. It must return an HttpResponse object, or raise an exception. It’s this response object which is used to kick off the WSGI Handler’s process\_viewcycle.

**process\_view**

Now that the WSGI Handler knows which view function to call, it once again loops through its list of middleware methods. The process\_view method for any Django middleware is declared like this:

process\_view(request, view\_function, view\_args, view\_kwargs)

Just like with process\_request, the process\_view function must return either None or an HttpResponse object (or raise an exception), allowing the WSGI Handler to either continue processing views, or “short-circuiting” and returning a response. Take a look at the [source code for CSRF Middleware](https://github.com/django/django/blob/5a6f70b4281817656db2f36c5919036d38fcce7f/django/middleware/csrf.py) to see an example of process\_view in action. If a CSRF cookie is present, the process\_view method returns None and the execution of the view occurs. If not, the request is rejected and the process is short-circuited, resulting in a failure message.

**process\_exception**

If the view function raises an exception, the Handler will loop through its list of process\_exception methods. These methods are executed in reverse order, from the last middleware listed in settings.py to the first. If an exception is raised, the process will short-circuit and no other process\_exceptionmiddlewares will be called. Usually we rely on the exception handlers provided by [Django’s BaseHandler](https://github.com/django/django/blob/5a6f70b4281817656db2f36c5919036d38fcce7f/django/core/handlers/base.py), but you can certainly implement your own exception handling when you write your own custom middleware class.

**process\_response**

At this point, we’ll have an HttpResponse object, either returned by the view or by the list of process\_view methods built by the WSGI handler, and it’s time to loop through the response middlewares in turn. This is the last chance any middleware has to modify the data, and it’s executed from the inner layer outward (think of an onion, with the view at the center). Take a look at the [cache middleware source code](https://github.com/django/django/blob/5a6f70b4281817656db2f36c5919036d38fcce7f/django/middleware/cache.py) for an example of process\_response in action: depending on different conditions in your app (i.e. whether caching is turned off, if we’re dealing with a stream, etc), we’ll want the response stored in the cache or not.

**Note:** One difference between pre-1.10 Django and later versions: in the older style MIDDLEWARE\_CLASSES, every middleware will always have its process\_response method called, even if an earlier middleware short-circuited the process. In the new MIDDLEWARES style, only that middleware and the ones which executed before it will have their process\_response methods called. Consult the documentation for more details on the differences between MIDDLEWARES and MIDDLEWARE\_CLASSES.

**All Done!**

Finally, Django’s WSGI Handler builds a return value from the HttpResponseobject and executes the callback function to send that data to the web server and out to the user.

So, two key takeaways:

1. Now we know how the view function is matched to a URLconf and what actually calls it (the WSGI Handler).
2. There are four key points you can hook into the request/response cycle through your own custom middleware: process\_request’, process\_response, process\_view, and process\_exception. Think of an onion: request middlewares are executed from the outside-in, hit the view at the center, and return through response middlewares back to the surface.

**from** **django.http** **import** HttpResponse

**import** **datetime**

**def** current\_datetime(request):

now = datetime.datetime.now()

html = "<html><body>It is now **%s**.</body></html>" % now

**return** HttpResponse(html)

**from** **django.http** **import** HttpResponse, HttpResponseNotFound

**def** my\_view(request):

*# ...*

**if** foo:

**return** HttpResponseNotFound('<h1>Page not found</h1>')

**else**:

**return** HttpResponse('<h1>Page was found</h1>')

**from** **django.http** **import** Http404

**from** **django.shortcuts** **import** render

**from** **polls.models** **import** Poll

**def** detail(request, poll\_id):

**try**:

p = Poll.objects.get(pk=poll\_id)

**except**

Poll.DoesNotExist:

**raise** Http404("Poll does not exist")

**return** render(request, 'polls/detail.html', {'poll': p})