Django’s request/response cycle

jacob@jacobian.org
Let’s recap:
What’s a view?

“A view function... is simply a Python function that takes a Web request and returns a Web response.”

— https://docs.djangoproject.com/en/1.9/topics/http/views/

def my_view(request):
    return HttpResponse(“it worked!”)
How does Django know to call my view?

“A URLconf... is a simple mapping between URL patterns (simple regular expressions) to Python functions (your views).”

— https://docs.djangoproject.com/en/1.9/topics/http/urls/

```python
patterns = [
    url('^view/$', my_view),
    ...
]
```
OK, but what does Django do before and after calling my view?

Well...
The Internet

Your View
The Internet

django

Your View
The Internet

GET /view/

HttpRequest

HttpResponse

200 OK

django

HttpRequest

HttpResponse

Your View
The Internet

GET /view/

HttpRequest

HttpResponse

200 OK

But what the heck happens in here?

Your View
Initial server load
Extension point “W”: WSGI middleware

Use cases:

• wrap the entire site in some behavior  
  (much code in Django happens “unprotected” by Django middleware)
• middleware that’s not Django-specific  
  (most doesn’t need to be…)

Examples:

• https://github.com/dahlia/wsgi-oauth2
• https://github.com/evansd/whitenoise

```python
from django.core.wsgi import get_wsgi_application
from wsgioauth2 import github

application = get_wsgi_application()
client = github.make_client(client_id='...', client_secret='...')
application = client.wsgi_middleware(application, secret='...')
```
First request
for each middleware object, instantiate then add to relevant middleware lists.

first request only

_load Middleware_()
**Extension point “M”: Django middleware**

**Use cases:** do something on every request — with more Django-specific context than WSGI middleware.

**Careful:** Middleware are classic Django foot-gun!

**Examples:**

- https://github.com/carljm/django-secure
- https://github.com/django-debug-toolbar/django-debug-toolbar
class SecurityMiddleware(object):
    def __init__(self):
        self.xss_filter = conf.SECURE_BROWSER_XSS_FILTER
        ...

    def process_request(self, request):
        ...
        path = request.path.lstrip("/")
        if (self.redirect and
            not request.is_secure() and
            not any(pattern.search(path)
                for pattern in self.redirect_exempt)):
            host = self.redirect_host or request.get_host()
            return HttpResponsePermanentRedirect(
                "https://%s%s" % (host, request.get_full_path()))

    def process_response(self, request, response):
        ...
        if self.xss_filter and not 'x-xss-protection' in response:
            response["x-xss-protection"] = "1; mode=block"
        return response
<table>
<thead>
<tr>
<th></th>
<th>WSGI middleware</th>
<th>vs</th>
<th>Django middleware</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Works with other Python frameworks, not just Django.</td>
<td>+ Simple API.</td>
<td>+ Easy integration with the rest of your app.</td>
<td></td>
</tr>
<tr>
<td>+ Limited access to other parts of the stack (e.g. databases).</td>
<td>- Django-specific.</td>
<td>- Easy to mess up and introduce performance problems.</td>
<td></td>
</tr>
<tr>
<td>- More confusing API.</td>
<td>- Misleading name.</td>
<td>- Misleading name.</td>
<td></td>
</tr>
<tr>
<td>- Hard to interoperate with Django-specific parts of your app.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Request phase
Extension point “S”: signals

Use case: I’m... not sure, exactly. Timing data?

```python
import time
from django.core import signal
from django.dispatch import receiver

@receiver(signals.request_started):
    def started(sender, **kwargs):
        print("at=request_started time={0} path={1}".format(
            time.time(), sender.environ['PATH_INFO'])))

@receiver(signals.request_finished)
    def ended(sender, **kwargs):
        print("at=request_finished time={0} path={1}".format(
            time.time(), sender.environ['PATH_INFO'])))
```

Yeah, silly example. But seriously, I don’t really know why you’d use signals — either middleware form works better.
Extension point “U”: request-specific urls

Use cases:

• Modifying URLs based on request details: multi-tenancy, internationalization, ...

• Probably under-used!

class MTMiddleware(object):
    def process_request(self, request):
        request.tenant = lookup_tenant(request)
        request.urlconf = 'mt.urls.%s' % request.tenant.slug
View phase
if view raises exception

go to exception flow
Response phase
Extension point “L”: lazy responses

Use cases:

- Defer response rendering until “later” in the response cycle.
- Better support more complex composed views.
- Allow composed views or middleware to inspect/modify template/context details.

**Note**: lazy responses are duck-typed!

def non_lazy(request):
    ...
    content = templ.render(context)
    return HttpResponse(content)

def lazy(request):
    ...
    return TemplateResponse(request, tmpl, context)
A short digression about WSGI

The Django control flow

- `call_(envicon, start_response)`
- `start_response(status, headers)`
- `response iterator`
- `loop`
- `next()`
- `response chunk`
- `close()`
Modifying Django’s request/response cycle:

- WSGI apps and middleware
- Django middleware
- Per-request URLconfs
- Signals
- Lazy responses
It’s a very elegant architecture you’ve got there. Be a shame if something were to happen to it...

https://github.com/andrewgodwin/channels
Channels

Django, today

The Internet

GET /view/

200 OK

HttpRequest

HttpResponse

Your View

Django, tomorrow

The Internet

Websocket

Channels

Your View
Thanks y’all!

jacob@jacobian.org
Bonus: Colophon
Generating diagrams with seqdiag

(seqdiag)

```plaintext

seqdiag {
  client -> server [label = "GET /resource"]; 
  server => database [label = "SELECT * FROM table;"]; 
  client <-- server [label = "{ ... json ... }"]; 
}
```
