Django Deployment & Tips

2010. 6. 7

daybreaker

We have covered…

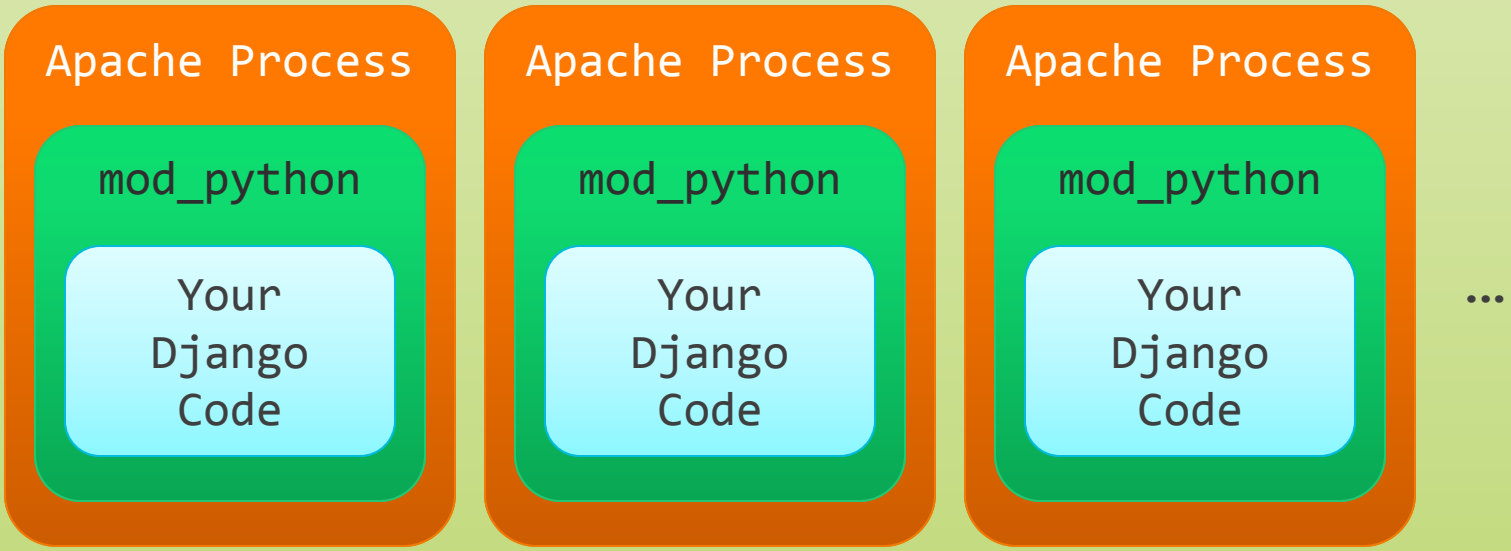
- Django Basics
  - Concept of MVC
  - Templates
  - Models
  - Admin Sites
  - Forms
  - Users (authentication)
- Thanks to battery…

Today's Contents

- Deployment: mod_python & WSGI
- Tips: More Convenient Development
- Tips: More Elegant Structure
- Tips: Faster Application

Deployment: mod_python

- mod_python
  - Your code runs in apache processes (like traditional PHP)



Directly handles HTTP requests…

Deployment: mod_python

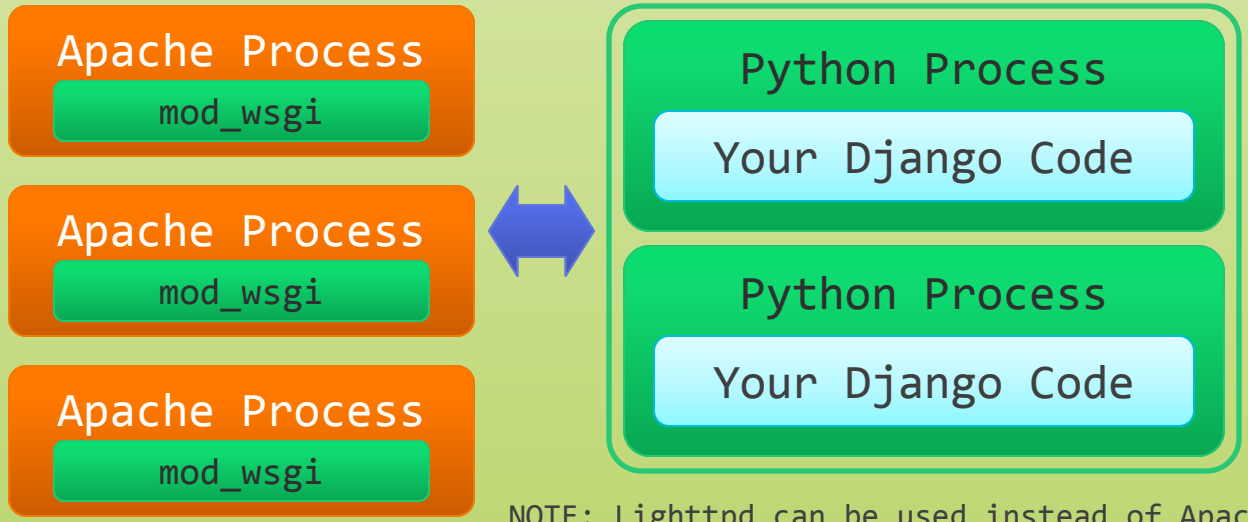
- Apache Configuration

```
DocumentRoot /projects/mysite/static/
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonOption django.root /
    PythonDebug On
    PythonPath "['/projects/'] + sys.path"
</Location>
<Location "/media">
    SetHandler None
</Location>
```

  - **django.root** specifies a global URL prefix for your site.

Deployment: WSGI

- WSGI (or FastCGI)
  - Your code runs in **separated Python *daemon* processes** via IPC.
  - More suitable for contemporary multi-processor systems



NOTE: Lighttpd can be used instead of Apache.

Deployment: WSGI

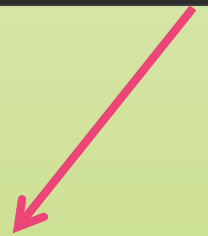
- Differences to traditional CGI
  - It *reuses* daemon processes, not launching them every time.
  - *Permissions* can be separated for better security.
- **WSGI** is the de-facto standard web application interface *for Python*.
  - FastCGI can also be used. (Modern PHP applications use it.)

Deployment: WSGI

- Apache Configuration (with mod_wsgi enabled)

optional

```
WSGIDaemonProcess mysite processes=2 threads=15
                  user=daybreaker group=www-data
WSGIProcessGroup mysite
WSGIScriptAlias / /projects/mysite/django.wsgi
```



- WSGI Handler Script

```
#! /usr/bin/env python
import os, sys
os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'
from django.core.handlers import wsgi
application = wsgi.WSGIHandler()
```

Deployment: WSGI

- Apache Configuration (extended)
  - We should set exception rules for static files.

```
Alias /robots.txt /projects/mysite/static/robots.txt
Alias /favicon.ico /projects/mysite/static/favicon.ico
Alias /media/ /projects/mysite/static/media/
<Directory /projects/mysite/static>
  Order deny,allow
  Allow from all
</Directory>
WSGIScriptAlias / /projects/mysite/django.wsgi
<Directory /projects/mysite>
  Order allow,deny
  Allow from all
</Directory>
```

You will know the tips or their necessity after some projects without this seminar.

But it’s better to know best practices even before you understand why they are good.

More Convenient Develeopment

- Don’t hard-code MEDIA_ROOT, TEMPLATE_DIRS and static files in templates (Use MEDIA_URL)

settings.py

```
import os.path
PROJECT_PATH = os.path.dirname(__file__)
MEDIA_ROOT = os.path.join(PROJECT_PATH, 'static/media/')
MEDIA_URL = '/media/'
            # may be other servers, eg. 'http://static.mysite.com/'
TEMPLATE_DIRS = (
  os.path.join(PROJECT_PATH, 'templates'),
)
```

templates

```
<link rel="stylesheet" type="text/css"
      href="{{MEDIA_URL}}styles/main.css" />
<script type="text/javascript" src="{{MEDIA_URL}}scripts/main.js" />
```

More Convenient Develeopment

- RequestContext instead of Context
  - Automatically adds *request*, *user* template variables.

In views

```
from django.template import RequestContext
def my_view(request):
    # do something
    return render_to_response('mytemplate.html', {
        … # some template variables
    }, context_instance=RequestContext(request))
```

More Convenient Develeopment

- DEBUG = True/False depending on the server

settings.py

```
import socket
if socket.gethostname() == 'productionserver.com':
    DEBUG = False
else:
    DEBUG = True
```

- Or, settings_local.py for each server

At the end of settings.py

```
try: from settings_local import *
except ImportError: pass
```

Then, create settings_local.py at the same directory.

More Convenient Develeopment

- Avoid using project name
  - Greatly increases reusability of apps
    (You don't have to fix all occurrences in all source files in an app when you try to use it in different projects.)
  - Manipulate PYTHON_PATH!
    (cont'd)

More Convenient Develeopment

- Avoid using project name

WSGI Handler Script

```
#! /usr/bin/env python
import os, sys
os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'
current_path = os.path.abspath(os.path.dirname(__file__))
sys.path.append(os.path.join(current_path, 'apps'))
from django.core.handlers import wsgi
application = wsgi.WSGIHandler()
```

In your apps

```
from ~~mysite.apps.~~myapp1.models import Model1
from ~~mysite.apps.~~myapp2.models import Model2, Model3
```

※ This is for the case that apps are in 'apps' sub-package.

More Convenient Develeopment

- Load 3rd-party templatetags automatically
  - No more manual {% load something %} in every template that uses it.

At the end of settings.py

```
from django import template
template.add_to_builtins(
  'someapp.templatetags.custom_tag_module'
)
```

More Convenient Develeopment

- Django command extensions
  - http://code.google.com/p/django-command-extensions/
  - Provides extra manage.py commands.
    - shell_plus : automatic imports of models
    - show_urls : lists all defined URLs
    - …and many others!

More Convenient Development

- Testing multiple Django version on a single server

WSGI Handler Script

```
#! /usr/bin/env python
import os, sys
os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'
sys.path = ['/path/to/secondary-django'] + sys.path
from django.core.handlers import wsgi
application = wsgi.WSGIHandler()
```

Note that the new path comes *first*.

More Elegant Structure

- Logic codes in models instead of views
  - For reusability and testability!
  - Views are coupled with HttpRequest & HttpResopnse, so they are dependent on their running environments.
    (It's tedious to provide simulation of them for unit-testing and granularity is too big.)
  - Models can be imported and used independently.

More Elegant Structure

- Use hierarchical urlpatterns
  - Even *without* additional urls.py files!

someapp/urls.py

```
labsite_urlpatterns = patterns('someapp.views',
    url(ur'^$', 'dashboard', name='dashboard'),
    url(ur'^join/$', 'new_join_request'),
    …
)
urlpatterns = patterns('someapp.views',
    url(ur'^$', 'index', name='index'),
    url(ur'^browse/$', 'browse', name='browse'),
    …
    url(ur'^(?P<url_key>[-a-zA-Z0-9]+)/', include(labsite_urlpatterns)),
)
```

More Elegant Structure

- Use named URL pattern (with namespaces)
  - *Reverse Lookup* improves reusability of your apps.

labsite/urls.py

```
labsite_urlpatterns = patterns('opencourselabs.labsite.views',
    …
    url(ur'^teams/(?P<team_id>[0-9]+)/$', 'view_team_console', name='team-console'),
)
urlpatterns = patterns('opencourselabs.labsite.views',
    …
    url(ur'^(?P<url_key>[-a-zA-Z0-9]+)/', include(labsite_urlpatterns)),
)
```

urls.py

```
urlpatterns = ('',
    …
    (ur'^lab/', include('opencourselabs.labsite.urls', namespace='labsite')),
)
```

In templates

```
{% for team in teams %}
    <li>…
        <a href="{% url labsite:team-console team.belongs_to.url_key
team.id %}">{{team.name}}</a>
    </li>
{% endfor %}
```

Faster Application

- Caching, caching, caching.
  - Use cache for dynamic contents that does not need real-time updates but requires large computation.
  - Be aware of cache consistency!!!
  - Example: course search in OTL
    https://project.sparcs.org/otl/browser/otl/apps/timetable/views.py#L45

Faster Application

- QuerySet.select_related()
  - Lazy evaluation sometimes makes excessive similar queries when accessing related objects from an object list.
  - Append select_related() at the end of queryset creation for complicated relational results.
  - Trades memory for performance.
  - OTL got vast performance improvements by this tip.

```
MyObject.objects.filter(…).select_related()
```

Other Advanced Topics

- Refer battery's slides
- Advanced Models & QuerySets
  - Custom fields
  - Q() function for complicated filter expressions
  - Using signals for "ON DELETE", "ON UPDATE"
- Advanced Admin
  - Custom pages, custom forms and fields
- Making your own authentication backends
  - Using OpenID, oAuth, …
- Making your own management commands
  - ./manage.py mycommand blah blah
- Using non-default template/db engine
- Caching – The secret how **OTL** survived.
- Internationalization (providing multilingual text)
- Tricks with Middleware – like global decorator

References

- Django 1.2 Documentation
  http://docs.djangoproject.com/en/1.2/
  - Put this under your pillow!
- “Top 10 Tips to a New Django Developer”
  http://www.ramavadakattu.com/top-10-tips-to-a-new-django-developer
- CCIU Open Course Labs
  http://code.google.com/p/cciu-open-course-labs/
  - Read commercial-level codes and applications
- OTL
  https://project.sparcs.org/otl/