Setting up Django and your web server with uWSGI and nginx

This tutorial is aimed at the Django user who wants to set up a production web server. It takes you through the steps required to set up Django so that it works nicely with uWSGI and nginx. It covers all three components, providing a complete stack of web application and server software.

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design.

nginx (pronounced *engine-x*) is a free, open-source, high-performance HTTP server and reverse proxy, as well as an IMAP/POP3 proxy server.

Some notes about this tutorial

> **Note**
>
> This is a **tutorial**. It is not intended to provide a reference guide, never mind an exhaustive reference, to the subject of deployment.

nginx and uWSGI are good choices for Django deployment, but they are not the only ones, or the 'official' ones. There are excellent alternatives to both, and you are encouraged to investigate them.

The way we deploy Django here is a good way, but it is **not** the *only* way; for some purposes it is probably not even the best way.

It is however a reliable and easy way, and the material covered here will introduce you to concepts and procedures you will need to be familiar with whatever software you use for deploying Django. By providing you with a working setup, and rehearsing the steps you must take to get there, it will offer you a basis for exploring other ways to achieve this.

> **Note**
>
> This tutorial makes some assumptions about the system you are using.

It is assumed that you are using a Unix-like system, and that it features an aptitude-like package manager. However if you need to ask questions like “What’s the equivalent of aptitude on Mac OS X?”, you’ll be able to find that kind of help fairly easily.

While this tutorial assumes Django 1.4 or later, which will automatically create a wsgi module in your new project, the instructions will work with earlier versions. You will though need to obtain that Django wsgi module yourself, and you may find that the Django project directory structure is slightly different.

Concept

A web server faces the outside world. It can serve files (HTML, images, CSS, etc) directly from the file system. However, it can’t talk *directly* to Django applications; it needs something that will run the application, feed it requests from web clients (such as browsers) and return responses.

A Web Server Gateway Interface - WSGI - does this job. WSGI is a Python standard.

uWSGI is a WSGI implementation. In this tutorial we will set up uWSGI so that it creates a Unix socket, and serves responses to the web server via the uwsgi protocol. At the end, our complete stack of components will look like this:

```
the web client <-> the web server <-> the socket <-> uwsgi <-> Django
```

Before you start setting up uWSGI

virtualenv

Make sure you are in a virtualenv for the software we need to install (we will describe how to install a system-wide uwsgi later):

```
virtualenv uwsgi-tutorial
cd uwsgi-tutorial
source bin/activate
```

Django

Install Django into your virtualenv, create a new project, and `cd` into the project:

```
pip install Django
django-admin.py startproject mysite
cd mysite
```

About the domain and port

In this tutorial we will call your domain `example.com`. Substitute your own FQDN or IP address.

Throughout, we'll be using port 8000 for the web server to publish on, just like the Django runserver does by default. You can use whatever port you want of course, but I have chosen this one so it doesn't conflict with anything a web server might be doing already.

Basic uWSGI installation and configuration

Install uWSGI into your virtualenv

```
pip install uwsgi
```

Of course there are other ways to install uWSGI, but this one is as good as any. Remember that you will need to have Python development packages installed. In the case of Debian, or Debian-derived systems such as Ubuntu, what you need to have installed is `pythonX.Y-dev`, where X.Y is your version of Python.

Basic test

Create a file called `test.py`:

```
# test.py
def application(env, start_response):
    start_response('200 OK', [('Content-Type','text/html')])
    return [b"Hello World"] # python3
    #return ["Hello World"] # python2
```

> **Note**
>
> Take into account that Python 3 requires `bytes()`.

Run uWSGI:

```
uwsgi --http :8000 --wsgi-file test.py
```

The options mean:

- `http :8000`: use protocol http, port 8000
- `wsgi-file test.py`: load the specified file, test.py

This should serve a 'hello world' message directly to the browser on port 8000. Visit:

```
http://example.com:8000
```

to check. If so, it means the following stack of components works:

```
the web client <-> uWSGI <-> Python
```

Test your Django project

Now we want uWSGI to do the same thing, but to run a Django site instead of the `test.py` module.

If you haven't already done so, make sure that your `mysite` project actually works:

```
python manage.py runserver 0.0.0.0:8000
```

And if that works, run it using uWSGI:

```
uwsgi --http :8000 --module mysite.wsgi
```

- `module mysite.wsgi`: load the specified wsgi module

Point your browser at the server; if the site appears, it means uWSGI is able to serve your Django application from your virtualenv, and this stack operates correctly:

```
the web client <-> uWSGI <-> Django
```

Now normally we won't have the browser speaking directly to uWSGI. That's a job for the webserver, which will act as a go-between.

Basic nginx

Install nginx

```
sudo apt-get install nginx
sudo /etc/init.d/nginx start    # start nginx
```

And now check that nginx is serving by visiting it in a web browser on port 80 - you should get a message from nginx: "Welcome to nginx!". That means these components of the full stack are working together:

```
the web client <-> the web server
```

If something else *is* already serving on port 80 and you want to use nginx there, you'll have to reconfigure nginx to serve on a different port. For this tutorial though, we're going to be using port 8000.

Configure nginx for your site

You will need the `uwsgi_params` file, which is available in the `nginx` directory of the uWSGI distribution, or from https://github.com/nginx/nginx/blob/master/conf/uwsgi_params

Copy it into your project directory. In a moment we will tell nginx to refer to it.

Now create a file called mysite_nginx.conf in the /etc/nginx/sites-available/ directory, and put this in it:

```
# mysite_nginx.conf

# the upstream component nginx needs to connect to
upstream django {
    # server unix:///path/to/your/mysite/mysite.sock; # for a file socket
    server 127.0.0.1:8001; # for a web port socket (we'll use this first)
}

# configuration of the server
server {
    # the port your site will be served on
    listen      8000;
    # the domain name it will serve for
    server_name example.com; # substitute your machine's IP address or FQDN
    charset     utf-8;

    # max upload size
    client_max_body_size 75M;   # adjust to taste

    # Django media
    location /media  {
        alias /path/to/your/mysite/media;  # your Django project's media files - amend as required
    }

    location /static {
        alias /path/to/your/mysite/static; # your Django project's static files - amend as
required
    }

    # Finally, send all non-media requests to the Django server.
    location / {
        uwsgi_pass  django;
        include     /path/to/your/mysite/uwsgi_params; # the uwsgi_params file you installed
    }
}
```

This conf file tells nginx to serve up media and static files from the filesystem, as well as handle requests that require Django's intervention. For a large deployment it is considered good practice to let one server handle static/media files, and another handle Django applications, but for now, this will do just fine.

Symlink to this file from /etc/nginx/sites-enabled so nginx can see it:

```
sudo ln -s ~/path/to/your/mysite/mysite_nginx.conf /etc/nginx/sites-enabled/
```

Deploying static files

Before running nginx, you have to collect all Django static files in the static folder. First of all you have to edit mysite/settings.py adding:

```
STATIC_ROOT = os.path.join(BASE_DIR, "static/")
```

and then run

```
python manage.py collectstatic
```

Basic nginx test

Restart nginx:

```
sudo /etc/init.d/nginx restart
```

To check that media files are being served correctly, add an image called `media.png` to the `/path/to/your/project/project/media directory`, then visit http://example.com:8000/media/media.png - if this works, you'll know at least that nginx is serving files correctly.

It is worth not just restarting nginx, but actually stopping and then starting it again, which will inform you if there is a problem, and where it is.

nginx and uWSGI and test.py

Let's get nginx to speak to the "hello world" `test.py` application.

```
uwsgi --socket :8001 --wsgi-file test.py
```

This is nearly the same as before, except this time one of the options is different:

- `socket :8001`: use protocol uwsgi, port 8001

nginx meanwhile has been configured to communicate with uWSGI on that port, and with the outside world on port 8000. Visit:

http://example.com:8000/

to check. And this is our stack:

```
the web client <-> the web server <-> the socket <-> uWSGI <-> Python
```

Meanwhile, you can try to have a look at the uwsgi output at http://example.com:8001 - but quite probably, it won't work because your browser speaks http, not uWSGI, though you should see output from uWSGI in your terminal.

Using Unix sockets instead of ports

So far we have used a TCP port socket, because it's simpler, but in fact it's better to use Unix sockets than ports - there's less overhead.

Edit `mysite_nginx.conf`, changing it to match:

```
server unix:///path/to/your/mysite/mysite.sock; # for a file socket
# server 127.0.0.1:8001; # for a web port socket (we'll use this first)
```

and restart nginx.

Run uWSGI again:

```
uwsgi --socket mysite.sock --wsgi-file test.py
```

This time the `socket` option tells uWSGI which file to use.

Try http://example.com:8000/ in the browser.

If that doesn't work

Check your nginx error log(/var/log/nginx/error.log). If you see something like:

```
connect() to unix:///path/to/your/mysite/mysite.sock failed (13: Permission
denied)
```

then probably you need to manage the permissions on the socket so that nginx is allowed to use it.

Try:

```
uwsgi --socket mysite.sock --wsgi-file test.py --chmod-socket=666 # (very permissive)
```

or:

```
uwsgi --socket mysite.sock --wsgi-file test.py --chmod-socket=664 # (more sensible)
```

You may also have to add your user to nginx's group (which is probably www-data), or vice-versa, so that nginx can read and write to your socket properly.

It's worth keeping the output of the nginx log running in a terminal window so you can easily refer to it while troubleshooting.

Running the Django application with uwsgi and nginx

Let's run our Django application:

```
uwsgi --socket mysite.sock --module mysite.wsgi --chmod-socket=664
```

Now uWSGI and nginx should be serving up not just a "Hello World" module, but your Django project.

Configuring uWSGI to run with a .ini file

We can put the same options that we used with uWSGI into a file, and then ask uWSGI to run with that file. It makes it easier to manage configurations.

Create a file called `mysite_uwsgi.ini`:

```
# mysite_uwsgi.ini file
[uwsgi]

# Django-related settings
# the base directory (full path)
chdir           = /path/to/your/project
# Django's wsgi file
module          = project.wsgi
# the virtualenv (full path)
home            = /path/to/virtualenv

# process-related settings
# master
master          = true
# maximum number of worker processes
processes       = 10
# the socket (use the full path to be safe
socket          = /path/to/your/project/mysite.sock
# ... with appropriate permissions - may be needed
# chmod-socket    = 664
# clear environment on exit
vacuum          = true
```

And run uwsgi using this file:

```
uwsgi --ini mysite_uwsgi.ini # the --ini option is used to specify a file
```

Once again, test that the Django site works as expected.

Install uWSGI system-wide

So far, uWSGI is only installed in our virtualenv; we'll need it installed system-wide for deployment purposes.

Deactivate your virtualenv:

```
deactivate
```

and install uWSGI system-wide:

```
sudo pip install uwsgi

# Or install LTS (long term support).
pip install https://projects.unbit.it/downloads/uwsgi-lts.tar.gz
```

The uWSGI wiki describes several installation procedures. Before installing uWSGI system-wide, it's worth considering which version to choose and the most apppropriate way of installing it.

Check again that you can still run uWSGI just like you did before:

```
uwsgi --ini mysite_uwsgi.ini # the --ini option is used to specify a file
```

Emperor mode

uWSGI can run in 'emperor' mode. In this mode it keeps an eye on a directory of uWSGI config files, and will spawn instances ('vassals') for each one it finds.

Whenever a config file is amended, the emperor will automatically restart the vassal.

```
# create a directory for the vassals
sudo mkdir /etc/uwsgi
sudo mkdir /etc/uwsgi/vassals
# symlink from the default config directory to your config file
sudo ln -s /path/to/your/mysite/mysite_uwsgi.ini /etc/uwsgi/vassals/
# run the emperor
uwsgi --emperor /etc/uwsgi/vassals --uid www-data --gid www-data
```

You may need to run uWSGI with sudo:

```
sudo uwsgi --emperor /etc/uwsgi/vassals --uid www-data --gid www-data
```

The options mean:

- `emperor`: where to look for vassals (config files)
- `uid`: the user id of the process once it's started
- `gid`: the group id of the process once it's started

Check the site; it should be running.

Make uWSGI startup when the system boots

The last step is to make it all happen automatically at system startup time.

For many systems, the easiest (if not the best) way to do this is to use the `rc.local` file.

Edit `/etc/rc.local` and add:

```
/usr/local/bin/uwsgi --emperor /etc/uwsgi/vassals --uid www-data --gid www-data --daemonize
/var/log/uwsgi-emperor.log
```

before the line “exit 0”.

And that should be it!

Further configuration

It is important to understand that this has been a *tutorial*, to get you started. You **do** need to read the nginx and uWSGI documentation, and study the options available before deployment in a production environment.

Both nginx and uWSGI benefit from friendly communities, who are able to offer invaluable advice about configuration and usage.

nginx

General configuration of nginx is not within the scope of this tutorial though you’ll probably want it to listen on port 80, not 8000, for a production website.

You should also configure a separate nginx location block for serving non-Django files. For example, it’s inefficient to serve static files via uWSGI. Instead, serve them directly from Nginx and completely bypass uWSGI.

uWSGI

uWSGI supports multiple ways to configure it. See uWSGI’s documentation and examples.

Some uWSGI options have been mentioned in this tutorial; others you ought to look at for a deployment in production include (listed here with example settings):

```
env = DJANGO_SETTINGS_MODULE=mysite.settings # set an environment variable
safe-pidfile = /tmp/project-master.pid # create a pidfile
harakiri = 20 # respawn processes taking more than 20 seconds
limit-as = 128 # limit the project to 128 MB
max-requests = 5000 # respawn processes after serving 5000 requests
daemonize = /var/log/uwsgi/yourproject.log # background the process & log
```