How to Deploy Python WSGI Applications Using uWSGI Web Server with Nginx

Authored by: ***ASPHostServer Administrator*** *[asphostserver@gmail.com]*
Saved From: *http://faq.asphosthelpdesk.com/article.php?id=266*

Understanding uWSGI and Using Nginx

uWSGI is an ambitious project. Its toolset allows you to do so much more than simply hosting web applications. Since it does the job so well, and in such a performant way, over the years it has proven to be an indispensable tool for many system administrators and developers alike when it comes to deploying their applications.

Nginx, since version 0.8.40 supports the uwsgi protocol (uWSGI's own). This enables the WSGI applications running on uWSGI to communicate the best way possible with Nginx. What this means for you is the possibility of very simple-to-configure deployments which are highly flexible (and functional) and benefiting from many under-the-hood optimizations that comes along. Altogether, this makes uWSGI coupled with Nginx an excellent choice for many deployment scenarios.

uWSGI in Brief

"Despite its very confusing naming conventions, uWSGI itself is a vast project with many components, aiming to provide a full software stack for building hosting services. One of these components, the uWSGI server, runs Python WSGI applications. It is capable of using various protocols, including its own uwsgi wire protocol, which is quasi-identical to SCGI. In order to fulfil the understandable demand to use stand-alone HTTP servers in front of application servers, NGINX and Cherokee web servers are modularised to support uWSGI's [own] best performing uwsgi protocol to have direct control over its processes.â€•

uWSGI Highlights

- uWSGI comes with a WSGI adapter and it fully supports Python applications running on WSGI.
- It links with libpython. It loads the application code on startup and acts like a Python interpreter. It parses the incoming requests and invokes the Python callable.
- It comes with direct support for popular NGINX web server (along with Cherokee+ and lighttpd).
- It is written in C.
- Its various components can do much more than running an application, which might be handy for expansion.
-

Currently (as of late 2013), it is actively developed and has fast release cycles.

-

It has various engines for running applications (asynchronous and synchronous).

-

It can mean lower memory footprint to run.

Web Application Deployments with Nginx

Nginx is a very high performant web server / (reverse)-proxy. It has reached its popularity due to being light weight, relatively easy to work with and easy to extend (with add-ons / plug-ins). Thanks to its architecture, it is capable of handling a lot of requests (virtually unlimited), which - depending on your application or website load - could be really hard to tackle using some other, older alternatives.

Remember:* "Handling" connections technically means not dropping them and being able to serve them with *something. You still need your application and database functioning well in order to have Nginx serve clients responses that are not error messages.

Using Nginx as Reverse-Proxy with uWSGI

Many frameworks and application servers can serve static files (e.g. javascript, css, images etc.) together with responses from the actual applications. However, the better thing to do is to let a (reverse-proxy) server such as Nginx handle the task of serving these files and managing connections (requests). This relieves a lot of the load from the application servers, granting you a much better overall performance.

As your application grows, you will want to optimise it and when the time comes, distribute it across servers to be able to handle more connections simultaneously and have a generally more robust architecture. Having a reverse-proxy in front of your application server(s) helps you with this from the very beginning.

Nginx's extensibility (e.g. native caching along with failover and other mechanisms) is also a great feat that benefits web applications unlike (simpler) application servers.

Example of a Basic Server Architecture:

```
Client Request ----> Nginx (Reverse-Proxy) | /|\ | | `-> App. Server I.
127.0.0.1:8081 | `--> App. Server II. 127.0.0.1:8082 `----> App. Server III.
127.0.0.1:8083
```

Note:When an application is set to listen for incoming connections on`127.0.0.1`, it will only be possible to access it locally. If you use`0.0.0.0`, however, it will accept connections from the outside as well.

Preparing Your Server Instance for Production

In this section, we are going to prepare our virtual server for production (i.e. for deploying our application).

We will begin with:

- updating the default operating system
- downloading and installing common Python tools (i.e. pip, virtualenv)
- creating a virtual environment to contain the application (inside which its dependencies such as uWSGI reside)

Updating the default operating system

Note:We will be performing the following setup and preparations on a new server, using recent versions of operating systems. In theory, you should not have problems trying them on your server. However, if you already actively use it, we highly recommend switching to a new system before you try.

To ensure that we have the latest available versions of default applications, we need to update our system.

For Debian Based Systems (i.e. Ubuntu, Debian), run the following:

```
aptitude update aptitude -y upgrade
```

For RHEL Based Systems (i.e. CentOS), run the following:

```
yum -y update
```

Setting up Python, pip and virtualenv

Note for CentOS / RHEL Users:

CentOS / RHEL, by default, comes as a very lean server. Its toolset, which is likely to be dated for your needs, is not there to run your applications but to power the server's system tools (e.g. YUM).

In order to prepare your CentOS system, Python needs to be set up (i.e. compiled from the source) and pip/virtualenv need installing using that interpreter.

On Ubuntu and Debian, a recent version of Python interpreter which you can use comes by default. It leaves us with only a limited number of additional packages to install:

- python-dev (development tools)

- pip (to manage packages)
- virtualenv (to create isolated, virtual environments)

python-dev:

python-devis an operating-system level package which contains extended development tools for building Python modules.

Run the following command to install python-dev using aptitude:

```
aptitude install python-dev
```

pip:

pipis a package manager which will help us to install the application packages that we need.

Run the following commands to install pip:

```
curl https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py | python -
curl https://raw.github.com/pypa/pip/master/contrib/get-pip.py | python - export
PATH="/usr/local/bin:$PATH"
```

You might need sudo privileges.

virtualenv:

It is best to contain a Python application within its ownenvironment**together with all of its dependencies. An environment can be best described (in simple terms) as an isolated location (a directory) where everything resides. For this purpose, a tool called **virtualenv is used.

Run the following to install virtualenv using pip:

```
sudo pip install virtualenv
```

Creating a self-contained Virtual (Python) Environment

Having all the necessary tools ready at our disposal, we can create an environment deploy our application.

Remember: If you haven't got a virtualenv on your development (local) machine for your project, you should consider creating one and moving your application (and its dependencies) inside.

Let's begin with creating a folder which will contain both the virtual environment and your application module:

You can use any name here to suit your needs.

```
mkdir my_app
```

We can continue with entering this folder and creating a new virtual environment inside:

You can also choose any name you like for your virtual environment.

```
cd my_app virtualenv my_app_venv
```

Let's create a new folder there to contain your Python application module as well:

This is the folder where your application module will reside.

```
mkdir app
```

And activate the interpreter inside the virtual environment to use it:

Please make sure to use the name you chose for your virtual environment if you went for something other than "my_app_venv".

```
source my_app_venv/bin/activate
```

In the end, this is how your main application deployment directory should look like:

```
my_app # Main Folder to Contain Everything Together | |=== my_app_venv # V. Env.
folder with the Python Int. |=== app # Your application module |.. |.
```

Downloading and installing uWSGI

It is always the recommended way to contain all application related elements, as much as possible, together inside the virtual environment. So we will download and install uWSGI as such.

If you are not working inside an environment, uWSGI will be installedglobally*(i.e. available systemwide). This is not recommended. Always opt for using***virtualenv.***

To install uWSGI using pip, run the following:

```
pip install uwsgi
```

Downloading and installing Nginx

Run the following command to use the default system package manager aptitude install Nginx:

```
sudo aptitude install nginx
```

To run Nginx, you can use the following:

```
sudo service nginx start
```

To stop Nginx, you can use the following:

```
sudo service nginx stop
```

To restart Nginx, you can use the following:

After each time you reconfigure Nginx, a restart or reload is needed for the new settings to come into effect.

```
sudo service nginx restart
```

Serving Python WSGI applications with uWSGI

In this section, we will see how a Python WSGI application works with uWSGI web server. Working with uWSGI to serve Python WSGI applications is not dissimilar to other application containers. What uWSGI needs, just like other servers, is for your application to provide it with an entry point (a callable). During launch, this callable, alongside configuration variables, are passed to uWSGI and it starts to do its job. When a request arrives, it processes it and passes it to your application's controller to handle.

Architecture:

```
........ /|\ | | `-> App. Server I. 127.0.0.1:8080 <--> Application | `--> App.
Server II. 127.0.0.1:8081 <--> Application .....
```

WSGI

WSGI in a nutshell is an interface between a web server and the application itself. It exists to ensure a standardized way between various servers and applications (frameworks) to work with each other, allowing interchangeability when necessary (e.g. switching from development to production environment), which is a must-have need nowadays.

WSGI Application Object (Callable): "wsgi.pyâ€•

As mentioned above, web servers running on WSGI need an application object (i.e. your application's).

With most frameworks and applications, this consists of a wsgi.py to contain and provide an application object (or callable) to be used by the server.

We will begin with creating an exemplary wsgi.py which then will be imported and used by uWSGI to run the application.

You can choose any name instead of wsgi.py. However, these are the ones that are commonly used (e.g. by Django).

Let's begin with creating a wsgi.py file to contain a basic WSGI application.

Run the following command to create a wsgi.py using the text editor nano:

```
nano wsgi.py
```

And let's continue with moving (copy/paste) the basic WSGI application code inside (which should be replaced with your own application's callable for production):

```
def application(env, start_response): start_response('200 OK', [('Content-Type',
'text/html')]) return ["Hello!"]
```

This is the file that is included by the server and each time a request comes, the server uses this application callable to run the application's request handlers (i.e. controllers) upon parsing the URL (e.g. mysite.tld/controller/method/variable).

After placing the application code in, press CTRL+X and then confirm with Y to save this file inside the "my_appâ€• folder alongside the virtual environment and the app module containing your actual application.

Note: This WSGI application is the most basic example to its kind. You will need to replace this code block to include your own application object from the application module.

Once we are done, this is how your main application deployment directory should look like:

```
my_app # Main Folder to Contain Everything Together | |=== my_app_venv # V. Env.
folder with the Python Int. |=== app # Your application module | |--- wsgi.py #
File containing application callable |.. |.
```

Running the server

uWSGI has a lot of options and configurations with many possible ways of using them thanks to its flexibility. Without complicating things from the start, we will begin with working with it as simply as possible and continuing thereon with more advanced methods.

Simple usage example:

```
uwsgi [option] [option 2] .. -w [wsgi file with app. callable]
```

To simply run uWSGI to start serving the application from wsgi.py, run the following:

```
uwsgi --socket 127.0.0.1:8080 --protocol=http -w wsgi
```

This will run the server on the foreground. If you would like to stop it, press CTRL+C.

To run the server in the background, run the following:

```
uwsgi --socket 127.0.0.1:8080 --protocol=http -w wsgi &
```

[!] Important:* When you run the server to work with Nginx using the configuration from the section *Configuring Nginx, make sure to remove`--protocol=http`from the chain of arguments, otherwise Nginx and uWSGI will not be able to talk to each other.

When you run an application in the background, you will need to use a process manager (e.g. htop) to kill (or stop) it. See the section below for more details.

Managing uWSGI server and processes with signals

Managing uWSGI consists of actions taken during runtime. There are various commands set for this task to manipulate the process:

- **SIGHUP`-HUP`gracefully reloads the workers and the application**
- **SIGTERM`-TERM`"brutally" reloads**
- **SIGINT`-INT`andSIGQUIT`-QUIT`kills all the workers immediately**
- **SIGUSR1`-USR1`prints statistics (stdout)**
- **SIGUSR2`-USR2`prints worker status**
- **SIGURG`-URG`restores a snapshot**
- **SIGTSTP`-TSTP`pauses, suspends or resumes an instance**
- ***SIGWINCH*`-WINCH`wakes up a worker blocked in a*syscall***

Example management with signals:

Restarting the server with SIGHUP

This command gracefully restarts the server. What this means is, it waits for the current workers' job to be done, and then it terminates them before spawning them again, inheriting its settings.

Usage:`kill -HUP [PID]`

If you do not want to specify the PID, you can use the`pidfile`option to have uWSGI write it to a file, which you can then use to manage the processes.

Stopping the server with SIGINT

To stop the server and its processes, you need to use the`-INT`signal. This will terminate everything in the background.

Example:`kill -INT [PID]`

Configuring Nginx

After setting up uWSGI to run our application, we now need to do the same with Nginx for it to talk with the uWSGI server(s). For this, we need to modify Nginx's configuration file:`nginx.conf`

Run the following command to open up`nginx.conf`and edit it using nano text editor:

```
sudo nano /etc/nginx/nginx.conf
```

Afterwards, you can replace the file with the following example configuration to get Nginx work as a reverse-proxy, talking to your application.

Example configuration for web applications:

```
worker_processes 1; events { worker_connections 1024; } http { sendfile on; gzip on; gzip_http_version 1.0; gzip_proxied any; gzip_min_length 500; gzip_disable "MSIE [1-6]\."; gzip_types text/plain text/xml text/css text/comma-separated-values text/javascript application/x-javascript application/atom+xml; # Configuration containing list of application servers upstream uwsgicluster { server 127.0.0.1:8080; # server 127.0.0.1:8081; # .. # . } # Configuration for Nginx server { # Running port listen 80; # Settings to by-pass for static files location ^~ /static/ { # Example: # root /full/path/to/application/static/file/dir; root /app/static/; } # Serve a static file (ex. favico) outside static dir. location = /favico.ico { root /app/favico.ico; } # Proxying connections to application servers location / { include uwsgi_params; uwsgi_pass uwsgicluster; proxy_redirect off; proxy_set_header Host $host; proxy_set_header X-Real-IP $remote_addr; proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for; proxy_set_header X-Forwarded-Host $server_name; } } }
```

When you are done modifying the configuration, press CTRL+X and confirm with Y to save and exit. You will need to restart Nginx for changes to come into effect.

Run the following to restart Nginx:

```
sudo service nginx stop sudo service nginx start
```

Configuring uWSGI

When launching uWSGI to serve applications, there are several ways to supply it with necessary configurations such as the socket to run on, number of processes, master process settings etc. In this section, we will talk about three of them:

- Passing configurations as arguments
- Using `.ini`files for configurations
- Using `.json`files for configurations

Note:*uWSGI supports various protocols and ways to retrieve these configuration files such as*stdin andHTTP.

Option #1: Passing configurations as arguments:

Albeit easily getting confusing and hard to manage at times, the most basic way to run uWSGI is just like any other shell script - by supplying it the necessary configurations as arguments.

Usage example:

```
# uwsgi [option] [option 2] .. -w [wsgi.py with application callable] # Simple
server running *wsgi* uwsgi --socket 127.0.0.1:8080 -w wsgi # Running Pyramid
(Paster) applications uwsgi --ini-paste production.ini # Running web2py
applications uwsgi --pythonpath /path/to/app --module wsgihandler # Running WSGI
application with specific module / callable names uwsgi --module
wsgi_module_name --callable application_callable_name uwsgi -w
wsgi_module_name:application_callable_name
```

For more examples on how to run uWSGI, consider visiting its documentation for example configurations.

Option #2: Using`.ini`files for configurations

Another (and probably) a better way of providing uWSGI with configurations is through`.ini`files. These files have a simple structure (see example below) and need to passed explicitly each time uWSGI launch script is executed.

Example`.ini`structure (**example_config.ini):**

```
[uwsgi] # ------------- # Settings: # key = value # Comments >> # #
------------- # socket = [addr:port] socket = 127.0.0.1:8080 # Base application
directory # chdir = /full/path chdir = /my_app # WSGI module and callable #
module = [wsgi_module_name]:[application_callable_name] module = app:application
# master = [master process (true of false)] master = true # processes = [number
of processes] processes = 5 ..
```

Usage example:

```
uwsgi --ini example_config.ini
```

Option #3: Using `.json`files for configurations

Working with `.json`files is, apart from the structure, same as the examples shown above.

Example `.json`structure (**example_config.json):**

```
{ "uwsgi": { "socket": ["127.0.0.1:8080"], "module": "my_app:app", "master":
true, "processes": 5, } }
```

Usage example:

```
uwsgi --json example_config.json
```

To learn more about configuring uWSGI, consider reading its documentation for configuration.

Common configurations for uWSGI

By default, this server is designed to be framework / application / platform agnostic by default. Although it probably supports anything and everything you might need, certain options will need to be explicitly set to comply with your requirements.

As stated on its own documentation, the number of possible ways to configure uWSGI is almost infinite. In this section, we will try to go over the most commonly used or critical ones and explain the implementation, which when combined with the previous section, will enable you to have uWSGI running exactly the way you need it.

For optimisations, please refer to next section after this one.

The syntax used in the examples below are targeted for `.ini`files. You can modify them to suit your specific needs as desired (e.g. for `.json`based configurations, as explained in the previous section).

Sockets

http-socket

Sets uWSGI to be bound to a certain HTTP socket.

Example:`http-socket = :8080`

socket

Sets uWSGI to be bound to specified socket using the default protocol.

Example:`socket = 127.0.0.1:8080`

processes (workers)

Either term can be used to refer to the same thing: the amount of processes spawned to accept requests.

Example:`processes = 5`

protocol

By default, uWSGI runs on its own uwsgi protocol. This property allows you to change it.

Example:`protocol = http`

Management

master

This option is used to enable or disable a master uWSGI process. These processes are used for managing workers which accept and deal with the incoming requests. The advantages are many, including gracefully restarting workers without touching the sockets which would allow you upgrades without downtime.

Example: `master = true`

max-requests

If you are worried about memory leaks and can not think of a more solid way of dealing with it, this option will allow you to restart your processes automatically after dealing with the set amount of requests specified.

Example: `max-requests = 1001`

threads

Settings for running each process with the specified amount of threads in rethread mode. It is possible to combine this option withprocessesin order to obtain concurrency in varying degrees.

Example: `threads = 2`

Logging

disable-logging

Used for disabling the logging feature.

Example: `disable-logging = true`

uWSGI processes

procname

Allows you to set the process name to something of your choice.

Example:`procname = My Application`

uid

Set uWSGI server user`uid`to the specified one.

Example: `uid = 1001`

gid

Set uWSGI server`gid`to the specified one.

Example: `gid = 555`

vacuum

Removes all generated pidfiles / sockets upon exit.

daemonize

This setting daemonizes uWSGI and writes messages to supplied argument (log file).

Example:`daemonize = /tmp/uwsgi_daemonize.log`

pidfile

Set uWSGI to write the process PID to a file specified by the option. This option is very handy for the management of uWSGI processes running (see the Managing uWSGI section for more information).

Example:`pidfile = /tmp/proj_pid.pid`

Various

harakiri

This setting is used to set the maximum amount of time a process is allowed to complete its task before it gets killed and recycled for memory / management purposes.

Example:`harakiri = 30`

To learn all about hundreds of available configurations, you should read check out the full list located at the official configuration options documentation.