By: Justin Ellingwood Subscribe

How To Set Up uWSGI and Nginx to Serve Python Apps on Ubuntu 14.04

14
Posted Mar 6, 2015 56k Python Frameworks Python Nginx Ubuntu

Introduction

In this guide, we will be setting up a simple WSGI application served by uWSGI. We will use the Nginx web server as a reverse proxy to the application server to provide more robust connection handling. We will be installing and configuring these components on an Ubuntu 14.04 server.

Definitions and Concepts

Clarifying Some Terms

Before we jump in, we should address some confusing terminology associated with the interrelated concepts we will be dealing with. These three separate terms that appear interchangeable, but actually have distinct meanings:

- **WSGI**: A Python spec that defines a standard interface for communication between an application or framework and an application/web server. This was created in order to simplify and standardize communication between these components for consistency and interchangeability. This basically defines an API interface that can be used over other protocols.
- **uWSGI**: An application server container that aims to provide a full stack for developing and deploying web applications and services. The main component is an application server that can handle apps of different languages. It communicates with the application using the methods defined by the WSGI spec, and with other web servers over a variety of other protocols. This is the piece that translates requests from a conventional web server into a format that the application can process.

- **uwsgi**: A fast, binary protocol implemented by the uWSGI server to communicate with a more full-featured web server. This is a wire protocol, not a transport protocol. It is the preferred way to speak to web servers that are proxying requests to uWSGI.

WSGI Application Requirements

The WSGI spec defines the interface between the web server and application portions of the stack. In this context, "web server" refers to the uWSGI server, which is responsible for translating client requests to the application using the WSGI spec. This simplifies communication and creates loosely coupled components so that you can easily swap out either side without much trouble.

The web server (uWSGI) must have the ability to send requests to the application by triggering a defined "callable". The callable is simply an entry point into the application where the web server can call a function with some parameters. The expected parameters are a dictionary of environmental variables and a callable provided by the web server (uWSGI) component.

In response, the application returns an iterable that will be used to generate the body of the client response. It will also call the web server component callable that it received as a parameter. The first parameter when triggering the web server callable will be the HTTP status code and the second will be a list of tuples, each of which define a response header and value to send back to the client.

With the "web server" component of this interaction provided by uWSGI in this instance, we will only need to make sure our applications have the qualities described above. We will also set up Nginx to handle actual client requests and proxy them to the uWSGI server.

Install the Components

To get started, we will need to install the necessary components on our Ubuntu 14.04 server. We can mainly do this using `apt` and `pip`.

First, refresh your `apt` package index and then install the Python development libraries and headers, the `pip` Python package manager, and the Nginx web server and reverse proxy:

```
sudo apt-get update
sudo apt-get install python-dev python-pip nginx
```

Once the package installation is complete, you will have access to the `pip` Python package manager. We can use this to install the `virtualenv` package, which we will use to isolate our application's Python environment from any others that may exist on the system:

```
sudo pip install virtualenv
```

Once this is complete, we can begin to create the general structure for our application. We will create the virtual environment discussed above and will install the uWSGI application server within this environment.

Set up an App Directory and a Virtualenv

We will start by creating a folder for our app. This can hold a nested folder containing the actual application code in a more complete application. For our purposes, this directory will simply hold our virtual environment and our WSGI entry point:

```
mkdir ~/myapp/
```

Next, move into the directory so that we can set up the environment for our application:

```
cd ~/myapp
```

Create a virtual environment with the `virtualenv` command. We will call this `myappenv` for simplicity:

```
virtualenv myappenv
```

A new Python environment will be set up under a directory called `myappenv`. We can activate this environment by typing:

```
source myappenv/bin/activate
```

Your prompt should change to indicate that you are now operating within the virtual environment. It will look something like this:

```
(myappenv)username@host:~/my_app$
```

If you wish to leave this environment at any time, you can simply type:

```
deactivate
```

If you have deactivated your environment, re-activate it again to continue with the guide.

With this environment active, any Python packages installed will be contained within this directory hierarchy. They will not interfere with the system's Python environment. With this in mind, we can now install the uWSGI server into our environment using `pip`. The package for this is called `uwsgi` (this is still the uWSGI server and not the `uwsgi` protocol):

```
pip install uwsgi
```

You can verify that it is now available by typing:

```
uwsgi --version
```

If it returns a version number, the uWSGI server is available for use.

Create a WSGI Application

Next, we will create an incredibly simple WSGI application using the WSGI specification requirements we discussed earlier. To reiterate, the application component that we must provide should have the following properties:

- It must provide an interface through a callable (a function or other language construct that can be called)
- The callable must take as parameters a dictionary containing environmental variable-like key-value pairs and a callable that is accessible on the server (uWSGI).
- The application's callable should return an iterable that will produce the body to send the client.
- The application should call the web server's callable with the HTTP status and request

headers.

We will write our application in a file called `wsgi.py` in our application directory:

```
nano ~/myapp/wsgi.py
```

Inside of this file, we will create the simplest WSGI compliant application we can. As with all Python code, be sure to pay attention to the indentation:

```
def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return ["<h1 style='color:blue'>Hello There!</h1>"]
```

The above code constitutes a complete WSGI application. By default, uWSGI will look for a callable called `application`, which is why we called our function `application`. As you can see, it takes two parameters.

The first we called `environ` because it will be an environmental variable-like key-value dictionary. The second is called `start_response` and is the name the app will use internally to refer to the web server (uWSGI) callable that is sent in. Both of these parameter names were simply selected because of their use in the examples in the PEP 333 spec that defines WSGI interactions.

Our application must take this information and do two things. First, it must call the callable it received with an HTTP status code and any headers it wants to send back. In this case, we are sending a "200 OK" response and setting the `Content-Type` header to `text/html`.

Secondly, it needs to return with an iterable to use as the response body. Here, we've just used a list containing a single string of HTML. Strings are iterable as well, but inside of a list, uWSGI will be able to process the entire string with one iteration.

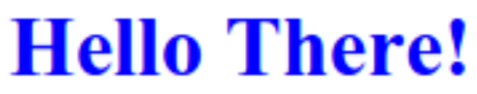
In a real world scenario, this file would likely be used as a link to the rest of your application code. For instance, Django projects include a `wsgi.py` file by default that translates requests from the web server (uWSGI) to the application (Django). The simplified WSGI interface stays the same regardless of how complex the actual application code is. This is one of the strengths of the interface.

Save and close the file when you are finished.

To test out the code, we can start up uWSGI. We will tell it to use HTTP for the time being and to listen on port `8080`. We will pass it the name of the script (suffix removed):

```
uwsgi --socket 0.0.0.0:8080 --protocol=http -w wsgi
```

Now, if you visit your server's IP address or domain name in your web browser followed by `:8080`, you should see the first level header text we passed as the body in our `wsgi.py` file:


Hello There!

Stop the server with CTRL-C when you have verified that this works.

We're done with designing our actual application at this point. You can deactivate our virtual environment if you desire:

```
deactivate
```

Configure a uWSGI Config File

In the above example, we manually started the uWSGI server and passed it some parameters on the command line. We can avoid this by creating a configuration file. The uWSGI server can read configurations in a variety of formats, but we will use the `.ini` format for simplicity.

To continue with the naming we've been using thus far, we'll call the file `myapp.ini` and place it in our application folder:

```
nano ~/myapp/myapp.ini
```

Inside, we need to establish a section called `[uwsgi]`. This section is where all of our configuration items will live. We'll start by identifying our application. The uWSGI server needs to know where the application's callable is. We can give the file and the function within:

```
[uwsgi]
module = wsgi:application
```

We want to mark the initial `uwsgi` process as a master and then spawn a number of worker processes. We will start with five workers:

```
[uwsgi]
module = wsgi:application

master = true
processes = 5
```

We are actually going to change the protocol that uWSGI uses to speak with the outside world. When we were testing our application, we specified `--protocol=http` so that we could see it from a web browser. Since we will be configuring Nginx as a reverse proxy in front of uWSGI, we can change this. Nginx implements a `uwsgi` proxying mechanism, which is a fast binary protocol that uWSGI can use to talk with other servers. The `uwsgi` protocol is actually uWSGI's default protocol, so simply by omitting a protocol specification, it will fall back to `uwsgi`.

Since we are designing this config for use with Nginx, we're also going to change from using a network port and use a Unix socket instead. This is more secure and faster. The socket will be created in the current directory if we use a relative path. We'll call it `myapp.sock`. We will change the permissions to "664" so that Nginx can write to it (we will be starting uWSGI with the `www-data` group that Nginx uses. We'll also add the `vacuum` option, which will remove the socket when the process stops:

```
[uwsgi]
module = wsgi:application

master = true
processes = 5

socket = myapp.sock
chmod-socket = 664
vacuum = true
```

We need one final option since we will be creating an Upstart file to start our application at boot. Upstart and uWSGI have different ideas about what the SIGTERM signal should do to an application. To sort out this discrepancy so that the processes can be handled as expected with Upstart, we just need to add an option called `die-on-term` so that uWSGI will kill the process instead of reloading it:

```
[uwsgi]
module = wsgi:application

master = true
processes = 5

socket = myapp.sock
chmod-socket = 664
vacuum = true

die-on-term = true
```

Save and close the file when you are finished. This configuration file is now set to be used with an Upstart script.

Create an Upstart File to Manage the App

We can launch a uWSGI instance at boot so that our application is always available. We will place this in the `/etc/init` directory that Upstart checks. We'll be calling this `myapp.conf`:

```
sudo nano /etc/init/myapp.conf
```

First, we can start out with a description of the service and pick out the system runlevels where it should automatically run. The standard user runlevels are 2 through 5. We will tell Upstart to stop the service when it's on any runlevel outside of this group (such as when the system is rebooting or in single user mode):

```
description "uWSGI instance to serve myapp"

start on runlevel [2345]
stop on runlevel [!2345]
```

Next, will tell Upstart about which user and group to run the process as. We want to run the application under our own account (we're using `demo` in this guide, but you should substitute your own user). We want to set the group to the `www-data` user that Nginx uses however. This is necessary because the web server needs to be able to read and write to the socket that our `.ini` file will create:

```
description "uWSGI instance to serve myapp"

start on runlevel [2345]
stop on runlevel [!2345]

setuid demo
setgid www-data
```

Next, we'll run the actual commands to start uWSGI. Since we installed uWSGI into a virtual environment, we have some extra work to do. We could just provide the entire path to the uWSGI executable, but instead, we will activate the virtual environment. This would make it easier if we were relying on additional software installed in the environment.

To do this, we'll use a `script` block. Inside, we'll change to our application directory, activate the virtual environment (we must use `.` in scripts instead of `source`), and start the uWSGI instance pointing at our `.ini` file:

```
description "uWSGI instance to serve myapp"

start on runlevel [2345]
stop on runlevel [!2345]

setuid demo
setgid www-data

script
    cd /home/demo/myapp
    . myappenv/bin/activate
    uwsgi --ini myapp.ini
end script
```

With that, our Upstart script is complete. Save and close the file when you are finished.

Now, we can start the service by typing:

```
sudo start myapp
```

We can verify that that it was started by typing:

```
ps aux | grep myapp
```

```
demo   14618  0.0  0.5  35868  5996 ?        S    15:02   0:00 uwsgi --ini
myapp.ini
demo   14619  0.0  0.5  42680  5532 ?        S    15:02   0:00 uwsgi --ini
myapp.ini
demo   14620  0.0  0.5  42680  5532 ?        S    15:02   0:00 uwsgi --ini
myapp.ini
demo   14621  0.0  0.5  42680  5532 ?        S    15:02   0:00 uwsgi --ini
myapp.ini
demo   14622  0.0  0.5  42680  5532 ?        S    15:02   0:00 uwsgi --ini
myapp.ini
demo   14623  0.0  0.5  42680  5532 ?        S    15:02   0:00 uwsgi --ini
myapp.ini
demo   15520  0.0  0.0  11740   936 pts/0    S+   15:53   0:00 grep --
color=auto myapp
```

This will start automatically on boot. You can stop the service at any time by typing:

```
sudo stop myapp
```

Configure Nginx to Proxy to uWSGI

At this point, we have a WSGI app and have verified that uWSGI can read and serve it. We have created a configuration file and an Upstart script. Our uWSGI process will listen on a socket and communicate using the `uwsgi` protocol.

We're now at the point where we can work on configuring Nginx as a reverse proxy. Nginx has the ability to proxy using the `uwsgi` protocol for communicating with uWSGI. This is a faster protocol than HTTP and will perform better.

The Nginx configuration that we will be setting up is extremely simple. Create a new file within the `sites-available` directory within Nginx's configuration hierarchy. We will call our file `myapp` to match the app name we have been using:

```
sudo nano /etc/nginx/sites-available/myapp
```

Within this file, we can specify the port number and the domain name that this server block should respond to. In our case, we'll be using the default port 80:

```
server {
    listen 80;
    server_name server_domain_or_IP;
}
```

Since we wish to send all requests on this domain or IP address to our WSGI application, we will create a single location block for requests beginning with `/`, which should match everything. Inside, we will use the `include` directive to include a number of parameters with reasonable defaults from a file in our Nginx configuration directory. The file containing these is called `uwsgi_params`. Afterwards, we will pass the traffic to our uWSGI instance over the `uwsgi` protocol. We will use the unix socket that we configured earlier:

```
server {
    listen 80;
    server_name server_domain_or_IP;

    location / {
        include         uwsgi_params;
        uwsgi_pass      unix:/home/demo/myapp/myapp.sock;
    }
}
```

That is actually all we need for a simple application. There are some improvements that could be made for a more complete application. For instance, we might define a number of upstream uWSGI servers outside of this block and then pass them to that. We might include some more uWSGI parameters. We might also handle any static files from Nginx directly and pass only dynamic requests to the uWSGI instance.

We do not need any of those features in our three-line app though, so we can save and close the file.

Enable the server configuration we just made by linking it to the `sites-enabled` directory:

```
sudo ln -s /etc/nginx/sites-available/myapp /etc/nginx/sites-enabled
```

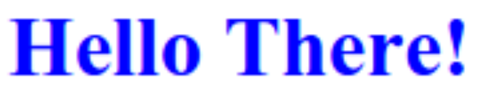
Check the configuration file for syntax errors:

```
sudo service nginx configtest
```

If it reports back that no problems were detected, restart the server to implement your changes:

```
sudo service nginx restart
```

Once Nginx restarts, you should be able to go to your server's domain name or IP address (without a port number) and see the application you configured:



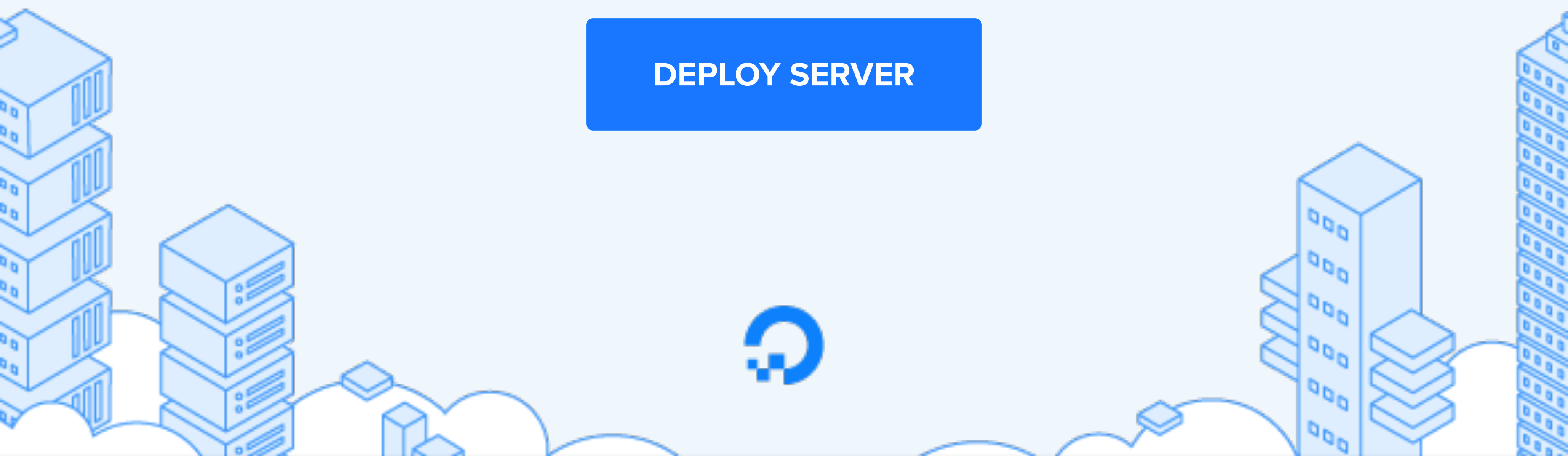
Hello There!

Conclusion

If you've made it this far, you've created a simple WSGI application and have some insight into how more complex applications would need to be designed. We have installed the uWSGI application container/server into a purpose-made virtual environment to serve our application. We've made a configuration file and an Upstart script to automate this process. In front of the uWSGI server, we've set up an Nginx reverse proxy that can speak to the uWSGI process using the `uwsgi` wire protocol.

You can easily see how this can be expanded when setting up an actual production environment. For instance, uWSGI has the ability to manage multiple applications using something called "emperor mode". You can expand the Nginx configuration to load balance between uWSGI instances, or to handle static files for your application. When serving multiple applications, it may be in your best interest to install uWSGI globally instead of in a virtual environment, depending on your needs. The components are all fairly flexible, so you should be able to tweak their configuration to accommodate many different scenarios.

Author:
Justin Ellingwood



Related Tutorials

How To Structure Large Flask Applications

How To Serve Flask Applications with Gunicorn and Nginx on Ubuntu 16.04

How To Serve Flask Applications with uWSGI and Nginx on Ubuntu 16.04

How To Set Up Django with Postgres, Nginx, and Gunicorn on Ubuntu 16.04

How To Use PostgreSQL with your Django Application on Ubuntu 16.04

10 Comments

Leave a comment...

Logged in as:

Notify me of replies to my comment

Comment

clyver April 25, 2015

Thank you for the tutorial. I am running into an issue as soon as I run: `sudo start myapp`. I am getting: `start: Job failed to start`. Grepping on `myapp` returns no results as well. Any help is appreciated!

0

hybby November 6, 2015

hit this myself. if you're on el6 and running an old version of upstart, some commands in the provided upstart job aren't recognised.

remove the following two lines:

0

```
setuid demo
setgid www-data
```

run the start command again and it'll work fine; i'm gonna check the man pages for the correct way to implement this functionality in el6 upstart, though.

MagicMe *June 9, 2015*

0

after i go to my domain, i see empty page.

in error log nothing, in access only favicon.ico and my domain

---

411skitrip *April 22, 2016*

0

I had this using Python 3.4 as my interpreter. Python 3+ requires bytes be sent rather than strings.

In the example wsgi.py making the following change worked for me:

```
return ["<h1 style='color:blue'>Hello There You!</h1>".encode('utf-
8')]
```

or

```
yield b"<h1 style='color:blue'>Hello There!</h1>"
```

uWSGI doc

---

yanchengcheok *August 12, 2015*

0

This tutorial is very helpful for beginner, to start from ground up. Thank you!

---

jarospirit *August 14, 2015*

0

Thank you,! Great tutorial, very informative and precise!

---

supremeanitabawa *October 17, 2015*

0

Hi , i went a bit far on this and configured HTTPS as well on NGINX,

But now when i open https://myserveraddress:8000/myapp/ , i am getting SSL*PROTOCOL*ERROR, and browser saying that the certificates that maybe required are not available.

Here are my files...

mysite_conf.conf the upstream component nginx needs to connect to
upstream django {
server unix:///home/ubuntu/docpad/Docpad/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 IP; # 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 /home/ubuntu/docpad/Docpad/assets;  # your Django project's
media files - amend as required
}

location /static {
    alias /home/ubuntu/docpad/Docpad/assets; # your Django project's
static files - amend as required
}

# Finally, send all non-media requests to the Django server.
location / {
    uwsgi_pass  django;
    include     /home/ubuntu/docpad/Docpad/uwsgi_params; # the
uwsgi_params file you installed
}
```

}

my default conf for nginx (/etc/nginx/sites-available/default)
server {
listen 443;
server_name IP;

```
root /;
index index.html index.htm;

ssl on;
ssl_certificate /home/ubuntu/docpad/self-ssl.crt;
ssl_certificate_key /home/ubuntu/docpad/self-ssl.key;

ssl_session_timeout 5m;

ssl_protocols SSLv3 TLSv1 TLSv1.1 TLSv1.2;
ssl_ciphers "HIGH:!aNULL:!MD5 or HIGH:!aNULL:!MD5:!3DES";
ssl_prefer_server_ciphers on;

location / {
    root html;
}
```

}

But no luck, anyone ?

---

0 lukasmoors *April 21, 2016*

Could someone please explain the benefits of having uWSGI sitting between nginx and, say, a flask-application?

---

0 411skitrip *April 23, 2016*

Quick tip, if you're on Ubuntu 15.10 or a droplet that uses systemd (rather than Upstart, as Ubuntu 14.04 uses) then this tutorial are more helpful for making the .ini and .service files.

---

0 sinclairnathan *May 19, 2016*

Thanks for giving such good advice on setting up python apps on ubuntu. I am having an issue with displaying the app once the nginx files have been updated. Just clarify for me, writing the myapp file in /nginx/sites is going to run when nginx starts? Im getting the default nginx page when I type in the ip address into the browser, even though I have set up the myapp file in sites-available and enabled