DigitalOcean

Subscribe Share Contents

How To Serve Flask Applications with uWSGI and Nginx on Ubuntu 18.04

11

Posted July 13, 2018 19.4k PYTHON PYTHON FRAMEWORKS NGINX UBUNTU UBUNTU 18.04

By: Justin Ellingwood By: Kathleen Juell

Not using **Ubuntu 18.04**? Choose a different version:

Introduction

In this guide, you will build a Python application using the Flask microframework on Ubuntu 18.04. The bulk of this article will be about how to set up the uWSGI application server and how to launch the application and configure Nginx to act as a front-end reverse proxy.

Prerequisites

Before starting this guide, you should have:

- A server with Ubuntu 18.04 installed and a non-root user with sudo privileges. Follow our initial server setup guide for guidance.
- Nginx installed, following Steps 1 and 2 of How To Install Nginx on Ubuntu 18.04.
- A domain name configured to point to your server. You can purchase one on Namecheap or get one for free on Freenom. You can learn how to point domains to DigitalOcean by following the relevant documentation on domains and DNS. Be sure to create the following DNS records:
    - An A record with `your_domain` pointing to your server's public IP address.
    - An A record with `www.your_domain` pointing to your server's public IP address.

- Familiarity with uWSGI, our application server, and the WSGI specification. This discussion of definitions and concepts goes over both in detail.

Our first step will be to install all of the pieces that we need from the Ubuntu repositories. We will install `pip`, the Python package manager, to manage our Python components. We will also get the Python development files necessary to build uWSGI.

First, let's update the local package index and install the packages that will allow us to build our Python environment. These will include `python3-pip`, along with a few more packages and development tools necessary for a robust programming environment:

```
$ sudo apt update
$ sudo apt install python3-pip python3-dev build-essential libssl-dev libffi-dev pyth
```

With these packages in place, let's move on to creating a virtual environment for our project.

Step 2 — Creating a Python Virtual Environment

Next, we'll set up a virtual environment in order to isolate our Flask application from the other Python files on the system.

Start by installing the `python3-venv` package, which will install the `venv` module:

```
$ sudo apt install python3-venv
```

Next, let's make a parent directory for our Flask project. Move into the directory after you create it:

```
$ mkdir ~/myproject
$ cd ~/myproject
```

Create a virtual environment to store your Flask project's Python requirements by typing:

```
$ python3.6 -m venv myprojectenv
```

This will install a local copy of Python and `pip` into a directory called `myprojectenv` within your project directory.

o so by typing:

```
$ source myprojectenv/bin/activate
```

Your prompt will change to indicate that you are now operating within the virtual environment. It will look something like this `(myprojectenv)user@host:~/myproject$`.

Step 3 — Setting Up a Flask Application

Now that you are in your virtual environment, you can install Flask and uWSGI and get started on designing your application.

First, let's install `wheel` with the local instance of `pip` to ensure that our packages will install even if they are missing wheel archives:

```
$ pip install wheel
```

Note

Regardless of which version of Python you are using, when the virtual environment is activated, you should use the `pip` command (not `pip3`).

Next, let's install Flask and uWSGI:

```
(myprojectenv) $ pip install uwsgi flask
```

Creating a Sample App

Now that you have Flask available, you can create a simple application. Flask is a microframework. It does not include many of the tools that more full-featured frameworks might, and exists mainly as a module that you can import into your projects to assist you in initializing a web application.

While your application might be more complex, we'll create our Flask app in a single file, called `myproject.py`:

```
(myprojectenv) $ nano ~/myproject/myproject.py
```

The application code will live in this file. It will import Flask and instantiate a Flask object. You can

~/myproject/myproject.py

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "<h1 style='color:blue'>Hello There!</h1>"

if __name__ == "__main__":
    app.run(host='0.0.0.0')
```

This basically defines what content to present when the root domain is accessed. Save and close the file when you're finished.

If you followed the initial server setup guide, you should have a UFW firewall enabled. To test the application, you need to allow access to port `5000`:

```
(myprojectenv) $ sudo ufw allow 5000
```

Now, you can test your Flask app by typing:

```
(myprojectenv) $ python myproject.py
```

You will see output like the following, including a helpful warning reminding you not to use this server setup in production:

```
Output
* Serving Flask app "myproject" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Visit your server's IP address followed by `:5000` in your web browser:

You should see something like this:

When you are finished, hit `CTRL-C` in your terminal window to stop the Flask development server.

Creating the WSGI Entry Point

Next, let's create a file that will serve as the entry point for our application. This will tell our uWSGI server how to interact with it.

Let's call the file `wsgi.py`:

```
(myprojectenv) $ nano ~/myproject/wsgi.py
```

In this file, let's import the Flask instance from our application and then run it:

~/myproject/wsgi.py

```
from myproject import app

if __name__ == "__main__":
    app.run()
```

Save and close the file when you are finished.

Step 4 — Configuring uWSGI

Your application is now written with an entry point established. We can now move on to configuring uWSGI.

Testing uWSGI Serving

Let's test to make sure that uWSGI can serve our application.

We can do this by simply passing it the name of our entry point. This is constructed by the name of the module (minus the `.py` extension) plus the name of the callable within the application. In our

Let's also specify the socket, so that it will be started on a publicly available interface, as well as the protocol, so that it will use HTTP instead of the `uwsgi` binary protocol. We'll use the same port number, `5000`, that we opened earlier:

```
(myprojectenv) $ uwsgi --socket 0.0.0.0:5000 --protocol=http -w wsgi:app
```

Visit your server's IP address with `:5000` appended to the end in your web browser again:

```
http://your_server_ip:5000
```

You should see your application's output again:

When you have confirmed that it's functioning properly, press `CTRL-C` in your terminal window.

We're now done with our virtual environment, so we can deactivate it:

```
(myprojectenv) $ deactivate
```

Any Python commands will now use the system's Python environment again.

Creating a uWSGI Configuration File

You have tested that uWSGI is able to serve your application, but ultimately you will want something more robust for long-term usage. You can create a uWSGI configuration file with the relevant options for this.

Let's place that file in our project directory and call it `myproject.ini`:

```
$ nano ~/myproject/myproject.ini
```

Inside, we will start off with the `[uwsgi]` header so that uWSGI knows to apply the settings. We'll specify two things: the module itself, by referring to the `wsgi.py` file minus the extension, and the callable within the file, `app`:


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

Next, we'll tell uWSGI to start up in master mode and spawn five worker processes to serve actual requests:

~/myproject/myproject.ini

```
[uwsgi]
module = wsgi:app

master = true
processes = 5
```

When you were testing, you exposed uWSGI on a network port. However, you're going to be using Nginx to handle actual client connections, which will then pass requests to uWSGI. Since these components are operating on the same computer, a Unix socket is preferable because it is faster and more secure. Let's call the socket `myproject.sock` and place it in this directory.

Let's also change the permissions on the socket. We'll be giving the Nginx group ownership of the uWSGI process later on, so we need to make sure the group owner of the socket can read information from it and write to it. We will also clean up the socket when the process stops by adding the `vacuum` option:

~/myproject/myproject.ini

```
[uwsgi]
module = wsgi:app

master = true
processes = 5

socket = myproject.sock
chmod-socket = 660
vacuum = true
```

The last thing we'll do is set the `die-on-term` option. This can help ensure that the init system and uWSGI have the same assumptions about what each process signal means. Setting this aligns the two system components, implementing the expected behavior:


```
module = wsgi:app

master = true
processes = 5

socket = myproject.sock
chmod-socket = 660
vacuum = true

die-on-term = true
```

You may have noticed that we did not specify a protocol like we did from the command line. That is because by default, uWSGI speaks using the `uwsgi` protocol, a fast binary protocol designed to communicate with other servers. Nginx can speak this protocol natively, so it's better to use this than to force communication by HTTP.

When you are finished, save and close the file.

Step 5 — Creating a systemd Unit File

Next, let's create the systemd service unit file. Creating a systemd unit file will allow Ubuntu's init system to automatically start uWSGI and serve the Flask application whenever the server boots.

Create a unit file ending in `.service` within the `/etc/systemd/system` directory to begin:

```
$ sudo nano /etc/systemd/system/myproject.service
```

Inside, we'll start with the `[Unit]` section, which is used to specify metadata and dependencies. Let's put a description of our service here and tell the init system to only start this after the networking target has been reached:

/etc/systemd/system/myproject.service

```
[Unit]
Description=uWSGI instance to serve myproject
After=network.target
```


we want the process to run under. Let's give our regular user account ownership of the process since it owns all of the relevant files. Let's also give group ownership to the `www-data` group so that Nginx can communicate easily with the uWSGI processes. Remember to replace the username here with your username:

/etc/systemd/system/myproject.service

```
[Unit]
Description=uWSGI instance to serve myproject
After=network.target

[Service]
User=sammy
Group=www-data
```

Next, let's map out the working directory and set the `PATH` environmental variable so that the init system knows that the executables for the process are located within our virtual environment. Let's also specify the command to start the service. Systemd requires that we give the full path to the uWSGI executable, which is installed within our virtual environment. We will pass the name of the `.ini` configuration file we created in our project directory.

Remember to replace the username and project paths with your own information:

/etc/systemd/system/myproject.service

```
[Unit]
Description=uWSGI instance to serve myproject
After=network.target

[Service]
User=sammy
Group=www-data
WorkingDirectory=/home/sammy/myproject
Environment="PATH=/home/sammy/myproject/myprojectenv/bin"
ExecStart=/home/sammy/myproject/myprojectenv/bin/uwsgi --ini myproject.ini
```

Finally, let's add an `[Install]` section. This will tell systemd what to link this service to if we enable it to start at boot. We want this service to start when the regular multi-user system is up and running:


```
Description=uWSGI instance to serve myproject
After=network.target

[Service]
User=sammy
Group=www-data
WorkingDirectory=/home/sammy/myproject
Environment="PATH=/home/sammy/myproject/myprojectenv/bin"
ExecStart=/home/sammy/myproject/myprojectenv/bin/uwsgi --ini myproject.ini

[Install]
WantedBy=multi-user.target
```

With that, our systemd service file is complete. Save and close it now.

We can now start the uWSGI service we created and enable it so that it starts at boot:

```
$ sudo systemctl start myproject
$ sudo systemctl enable myproject
```

Let's check the status:

```
$ sudo systemctl status myproject
```

You should see output like this:

```
Output
● myproject.service - uWSGI instance to serve myproject
   Loaded: loaded (/etc/systemd/system/myproject.service; enabled; vendor preset: enal
   Active: active (running) since Fri 2018-07-13 14:28:39 UTC; 46s ago
 Main PID: 30360 (uwsgi)
    Tasks: 6 (limit: 1153)
   CGroup: /system.slice/myproject.service
```

```
├─30360 /home/sammy/myproject/myprojectenv/bin/uwsgi --ini myproject.ini
├─30378 /home/sammy/myproject/myprojectenv/bin/uwsgi --ini myproject.ini
├─30379 /home/sammy/myproject/myprojectenv/bin/uwsgi --ini myproject.ini
├─30380 /home/sammy/myproject/myprojectenv/bin/uwsgi --ini myproject.ini
```


If you see any errors, be sure to resolve them before continuing with the tutorial.

Step 6 — Configuring Nginx to Proxy Requests

Our uWSGI application server should now be up and running, waiting for requests on the socket file in the project directory. Let's configure Nginx to pass web requests to that socket using the `uwsgi` protocol.

Begin by creating a new server block configuration file in Nginx's `sites-available` directory. Let's call this `myproject` to keep in line with the rest of the guide:

```
$ sudo nano /etc/nginx/sites-available/myproject
```

Open up a server block and tell Nginx to listen on the default port `80`. Let's also tell it to use this block for requests for our server's domain name:

/etc/nginx/sites-available/myproject

```
server {
    listen 80;
    server_name your_domain www.your_domain;
}
```

Next, let's add a location block that matches every request. Within this block, we'll include the `uwsgi_params` file that specifies some general uWSGI parameters that need to be set. We'll then pass the requests to the socket we defined using the `uwsgi_pass` directive:

/etc/nginx/sites-available/myproject

```
server {
    listen 80;
    server_name your_domain www.your_domain;

    location / {
```

```
        include uwsgi_params;
        uwsgi_pass unix:/home/sammy/myproject/myproject.sock;
    }
}
```


To enable the Nginx server block configuration you've just created, link the file to the `sites-enabled` directory:

```
$ sudo ln -s /etc/nginx/sites-available/myproject /etc/nginx/sites-enabled
```

With the file in that directory, we can test for syntax errors by typing:

```
$ sudo nginx -t
```

If this returns without indicating any issues, restart the Nginx process to read the new configuration:

```
$ sudo systemctl restart nginx
```

Finally, let's adjust the firewall again. We no longer need access through port `5000`, so we can remove that rule. We can then allow access to the Nginx server:

```
$ sudo ufw delete allow 5000
$ sudo ufw allow 'Nginx Full'
```

You should now be able to navigate to your server's domain name in your web browser:

```
http://your_domain
```

You should see your application output:

If you encounter any errors, trying checking the following:

- `sudo less /var/log/nginx/error.log`: checks the Nginx error logs.
- `sudo less /var/log/nginx/access.log`: checks the Nginx access logs.
- `sudo journalctl -u nginx`: checks the Nginx process logs.

Step 7 — Securing the Application

To ensure that traffic to your server remains secure, let's get an SSL certificate for your domain. There are multiple ways to do this, including getting a free certificate from Let's Encrypt, generating a self-signed certificate, or buying one from another provider and configuring Nginx to use it by following Steps 2 through 6 of How to Create a Self-signed SSL Certificate for Nginx in Ubuntu 18.04. We will go with option one for the sake of expediency.

First, add the Certbot Ubuntu repository:

```
$ sudo add-apt-repository ppa:certbot/certbot
```

You'll need to press ENTER to accept.

Next, install Certbot's Nginx package with `apt`:

```
$ sudo apt install python-certbot-nginx
```

Certbot provides a variety of ways to obtain SSL certificates through plugins. The Nginx plugin will take care of reconfiguring Nginx and reloading the config whenever necessary. To use this plugin, type the following:

```
$ sudo certbot --nginx -d your_domain -d www.your_domain
```

This runs `certbot` with the `--nginx` plugin, using `-d` to specify the names we'd like the certificate to be valid for.

If this is your first time running `certbot`, you will be prompted to enter an email address and agree to the terms of service. After doing so, `certbot` will communicate with the Let's Encrypt server, then run a challenge to verify that you control the domain you're requesting a certificate for.

If that's successful, `certbot` will ask how you'd like to configure your HTTPS settings.

Output


```
g HTTP access.
--------
.
2: Redirect - Make all requests redirect to secure HTTPS access. Choose this for
new sites, or if you're confident your site works on HTTPS. You can undo this
change by editing your web server's configuration.
-------------------------------------------------------------------------------
Select the appropriate number [1-2] then [enter] (press 'c' to cancel):
```

Select your choice then hit `ENTER`. The configuration will be updated, and Nginx will reload to pick up the new settings. `certbot` will wrap up with a message telling you the process was successful and where your certificates are stored:

Output

```
IMPORTANT NOTES:
 - Congratulations! Your certificate and chain have been saved at:
   /etc/letsencrypt/live/your_domain/fullchain.pem
   Your key file has been saved at:
   /etc/letsencrypt/live/your_domain/privkey.pem
   Your cert will expire on 2018-07-23. To obtain a new or tweaked
   version of this certificate in the future, simply run certbot again
   with the "certonly" option. To non-interactively renew *all* of
   your certificates, run "certbot renew"
 - Your account credentials have been saved in your Certbot
   configuration directory at /etc/letsencrypt. You should make a
   secure backup of this folder now. This configuration directory will
   also contain certificates and private keys obtained by Certbot so
   making regular backups of this folder is ideal.
 - If you like Certbot, please consider supporting our work by:

   Donating to ISRG / Let's Encrypt:   https://letsencrypt.org/donate
   Donating to EFF:                    https://eff.org/donate-le
```

If you followed the Nginx installation instructions in the prerequisites, you will no longer need the redundant HTTP profile allowance:

```
$ sudo ufw delete allow 'Nginx HTTP'
```

To verify the configuration, let's navigate once again to your domain, using `https://`:

You should see your application output once again, along with your browser's security indicator, which should indicate that the site is secured.

Conclusion

In this guide, you created and secured a simple Flask application within a Python virtual environment. You created a WSGI entry point so that any WSGI-capable application server can interface with it, and then configured the uWSGI app server to provide this function. Afterwards, you created a systemd service file to automatically launch the application server on boot. You also created an Nginx server block that passes web client traffic to the application server, relaying external requests, and secured traffic to your server with Let's Encrypt.

Flask is a very simple, but extremely flexible framework meant to provide your applications with functionality without being too restrictive about structure and design. You can use the general stack described in this guide to serve the flask applications that you design.

By: Justin Ellingwood By: Kathleen Juell

Related Tutorials

How To Install and Configure pgAdmin 4 in Server Mode

How To Build a Neural Network to Recognize Handwritten Digits with TensorFlow

How to Install, Run, and Connect to Jupyter Notebook on a Remote Server

How To Set Up a Jupyter Notebook with Python 3 on Debian 9

How To Set Up Django with Postgres, Nginx, and Gunicorn on Debian 9

11 Comments

Leave a comment...

Log In to Comment

jvm986 *August 4, 2018*

0 Great tutorial -- very well structured and easy to follow.

...es location.

jvm986 *August 5, 2018*

0 One more thing -- there's a typo in /etc/nginx/sites-available/myproject

```
server_name your_domain wwww.your_domain;
```

katjuell MOD *August 6, 2018*

1 @jvm986 thanks for the suggestion and observation!

cforsythe97 *August 24, 2018*

0 Does anyone know why I am getting this error when trying to start uwsgi?

```
uwsgi: error while loading shared libraries: libpcre.so.1: cannot open shared object file: No such file or directory
```

I have PCRE installed so if I type `sudo find / -name libpcre.so.1`

I get

```
/usr/local/lib/libpcre.so.1
/usr/local/mac-dev-env/pcre-8.42/lib/libpcre.so.1
/usr/local/src/pcre-8.42/.libs/libpcre.so.1
```

I've Googled everywhere and can't find anything.

For anyone that happens to run into the problem of loading shared libraries. I FINALLY found the solution.

weerachaikotrtum *September 22, 2018*

0 Thank you so much for this great tutorial.

matshidis *September 27, 2018*

0 Thank you for this, very easy to follow tutorial

following your
pt .py
document?

Do you need to stop the server and re-do everything, or would you just change the document and save it?

iamtony *October 8, 2018*

0 No just save the file and restart nginx!

$ sudo systemctl restart nginx

imebonia *October 12, 2018*

0 .sock failed (13: Permission denied) while connecting to upstream

any thoughts?
thanks

cferrante *October 16, 2018*

0 Although I don't recall getting the exact error you did, I noticed that to get my example working, I had to use /tmp/myproject.sock...perhaps that will work for you?

imebonia *October 17, 2018*

0 thanks @cferrante for the response. Figured out the reason. I had this whole project under root directory, where 'www-data' didn't have access. moved it to the different path and voila:)

mazzespazze *November 11, 2018*

0 Really a great tutorial. Sadly I am really new to Flask and nginx. I followed the whole tutorial and I get my website to work with the blue "Hello There".

How do I use it now? Let's say I have a registration form in a file registration.html, where do I put it in order to have a POST request?

Thanks in advance for any reply