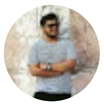
Deploy Flask Applications With uWSGI and Nginx on Ubuntu 18.04



Maanav Shah Follow
Jul 7, 2020 · 8 min read ★

If you are interested to know, why do we use Nginx in front of an application such as Flask, Django, Ruby on Rails, NodeJS, etc?

You can also read about, why is WSGI necessary?

When you have an Ubuntu or any Linux server and want to set up Flask application using Nginx and uWSGI, you need to log in as your non-root user to begin using:

```
ssh ip_address
```

Step 1 — Installing Nginx

Because Nginx is available in Ubuntu's default repositories, it is possible to install it from these repositories using the `apt` packaging system.

Since this is our first interaction with the `apt` packaging system in this session, we will update our local package index so that we have access to the most recent package listings. Afterward, we can install `nginx` :

```
sudo apt-get update
sudo apt-get install nginx
```

Step 2 — Checking your Web Server

At the end of the installation process, Ubuntu 18.04 starts Nginx. The web server should already be up and running.

We can check with the `systemd` init system to make sure the service is running by typing:

```
sudo systemctl status nginx
```

As you can see below, the service appears to have started successfully.

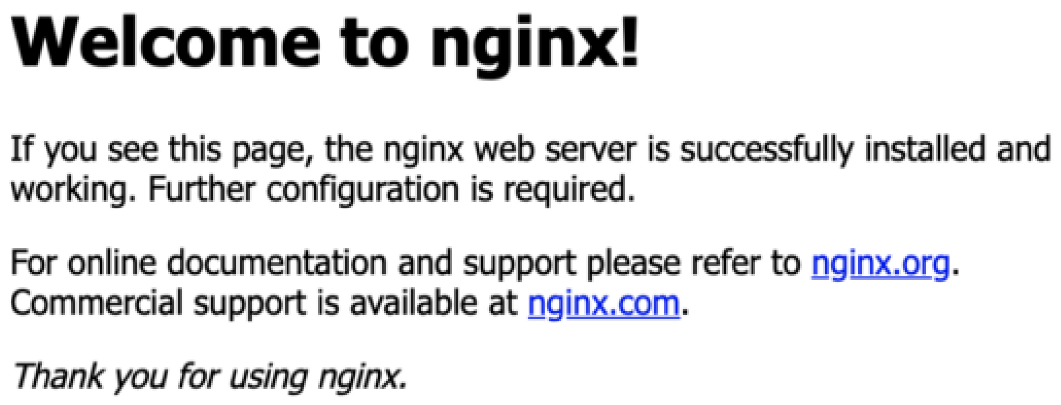
```
Output
● nginx.service - A high performance web server and a reverse proxy
server
   Loaded: loaded (/lib/systemd/system/nginx.service; enabled; vendor
preset: enabled)
   Active: active (running) since Tue 2020-07-07 07:50:48 UTC; 53min
ago
     Docs: man:nginx(8)
 Main PID: 10441 (nginx)
    Tasks: 2 (limit: 4373)
   Memory: 2.9M
   CGroup: /system.slice/nginx.service
           ├─10441 nginx: master process /usr/sbin/nginx -g daemon
on; master_process on;
           └─10443 nginx: worker process
```

However, the best way to test this is to actually request a page from Nginx.

When you have your server's IP address, enter it into your browser's address bar:

```
http://your_server_ip
```

You should see the default Nginx landing page:



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

This page is included with Nginx to show you that the server is running correctly.

Step 3 — Managing the Nginx Process

Now that you have your web server up and running, let's review some basic management commands.

To stop your web server, type:

```
sudo systemctl stop nginx
```

To start the webserver when it is stopped, type:

```
sudo systemctl start nginx
```

To stop and then start the service again, type:

```
sudo systemctl restart nginx
```

If you are simply making configuration changes, Nginx can often reload without dropping connections. To do this, type:

```
sudo systemctl reload nginx
```

By default, Nginx is configured to start automatically when the server boots. If this is not what you want, you can disable this behavior by typing:

```
sudo systemctl disable nginx
```

To re-enable the service to start up at boot, you can type:

```
sudo systemctl enable nginx
```

Step 4 — Installing the Components from the Ubuntu Repositories

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 `python-pip3` , along with a few more packages and development tools necessary for a robust programming environment:

```
sudo apt install python3-pip python3-dev build-essential libssl-dev
libffi-dev python3-setuptools
```

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

Step 5 — 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:

```
pip3 install virtualenv
```

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 -m virtualenv myprojectenv
```

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

Before installing applications within the virtual environment, you need to activate it. Do 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 6 — 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
```

Next, let's install Flask and uWSGI:

```
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`:

```
vi ~/myproject/myproject.py
```

The application code will live in this file. It will import Flask and instantiate a Flask object. You can use this to define the functions that should be run when a specific route is requested:

```
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.

Now, you can test your Flask app by typing:

```
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)
```

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` :

```
vi ~/myproject/wsgi.py
```

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

```
from myproject import app

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

Save and close the file when you are finished.

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

```
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`:

```
vi ~/myproject/myproject.ini
```

Let's put the content of our configuration file:

```
[uwsgi]
module = wsgi:app

master = true
processes = 5

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

die-on-term = true

logto = /home/maanav/myproject/myproject.log
```

When you are finished, save and close the file.

Note - *Please remember to change* `maanav` *to your username.*

Step 7 — Creating a systemd Unit File

Next, let’s create a 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 vi /etc/systemd/system/myproject.service
```

Let’s put the content of our server file:

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

[Service]
User=maanav
Group=www-data
WorkingDirectory=/home/maanav/myproject
Environment="PATH=/home/maanav/myproject/myprojectenv/bin"
ExecStart=/home/maanav/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: enabled)
   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/maanav/myproject/myprojectenv/bin/uwsgi --
ini myproject.ini
           ├─30378 /home/maanav/myproject/myprojectenv/bin/uwsgi --
ini myproject.ini
           ├─30379 /home/maanav/myproject/myprojectenv/bin/uwsgi --
ini myproject.ini
           ├─30380 /home/maanav/myproject/myprojectenv/bin/uwsgi --
ini myproject.ini
           ├─30381 /home/maanav/myproject/myprojectenv/bin/uwsgi --
ini myproject.ini
           └─30382 /home/maanav/myproject/myprojectenv/bin/uwsgi --
ini myproject.ini
```

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

Step 8 — 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:

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

location / {
        include uwsgi_params;
        uwsgi_pass unix:/home/maanav/myproject/myproject.sock;
    }
}
```

If we do not have a domain registered, then we can add IP address as server name:

```
server {
    listen 80;
    server_name ip_address;

location / {
        include uwsgi_params;
        uwsgi_pass unix:/home/maanav/myproject/myproject.sock;
    }
}
```

Save and close the file when you're finished.

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
```

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

http://your_domain

http://ip_address

You should see your application output:

Step 9 — Managing the application process

Now that you have your application up and running, let's review some basic management commands.

To stop your application, type:

```
sudo systemctl stop myproject
```

To start the application when it is stopped, type:

```
sudo systemctl start myproject
```

To stop and then start the service again, type:

```
sudo systemctl restart myproject
```

To check the status of the application:

```
sudo systemctl status myproject
```

Logs

Application Logs

`/home/maanav/myproject/myproject.log:` Every application request is recorded is in this log file.

Server Logs

`/var/log/nginx/access.log:` Every request to your web server is recorded in this log file unless Nginx is configured to do otherwise.
`/var/log/nginx/error.log:` Any Nginx errors will be recorded in this log.

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. Afterward, you created a systemd service file to automatically launch the application server on boot.

That's it. Hope this helps.

If you enjoyed this post, I'd be very grateful if you'd help it spread by emailing it to a friend or sharing it on Twitter or Facebook. Thank you!

Flask Wsgi Nginx Deployment Python