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

Justin Ellingwood

Introduction

In this guide, we will be setting up a simple Python application using the Flask micro-framework on Ubuntu 16.04. The bulk of this article will be about how to set up the uWSGI application server to launch the application and Nginx to act as a front end reverse proxy.

Prerequisites

Before starting on this guide, you should have a non-root user configured on your server. This user needs to have `sudo` privileges so that it can perform administrative functions. To learn how to set this up, follow our initial server setup guide.

To learn more about uWSGI, our application server and the WSGI specification, you can read the linked section of this guide. Understanding these concepts will make this guide easier to follow.

When you are ready to continue, read on.

Install the Components from the Ubuntu Repositories

Our first step will be to install all of the pieces that we need from the repositories. We will install `pip`, the Python package manager, in order to install and manage our Python components. We will also get the Python development files needed to build uWSGI and we'll install Nginx now as well.

We need to update the local package index and then install the packages. The packages you need depend on whether your project uses Python 2 or Python 3.

If you are using **Python 2**, type:

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

If, instead, you are using **Python 3**, type:

```
• sudo apt-get update
• sudo apt-get install python3-pip python3-dev nginx
```

Create 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 `virtualenv` package using `pip`.

If you are using **Python 2**, type:

```
• sudo pip install virtualenv
```

If you are using **Python 3**, type:

```
• sudo pip3 install virtualenv
```

Now, we can make a parent directory for our Flask project. Move into the directory after you create it:

```
• mkdir ~/myproject
• cd ~/myproject
```

We can create a virtual environment to store our Flask project's Python requirements by typing:

```
• virtualenv myprojectenv
```

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

Before we install applications within the virtual environment, we need to activate it. You can 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$`.

Set Up a Flask Application

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

Install Flask and uWSGI

We can use the local instance of `pip` to install Flask and uWSGI. Type the following commands to get these two components:

Note

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

```
• pip install uwsgi flask
```

Create a Sample App

Now that we have Flask available, we can create a simple application. Flask is a micro-framework. 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, which we will call `myproject.py`:

```
• nano ~/myproject/myproject.py
```

Within this file, we'll place our application code. Basically, we need to import Flask and instantiate a Flask object. We can use this to define the functions that should be run when a specific route is requested:

~/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. In order to test our application, we need to allow access to port 5000.

Open up port 5000 by typing:

```
• sudo ufw allow 5000
```

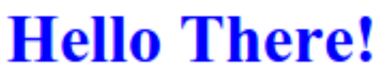
Now, you can test your Flask app by typing:

```
• python myproject.py
```

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

```
http://server_domain_or_IP:5000
```

You should see something like this:


Hello There!

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

Create the WSGI Entry Point

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

We will call the file `wsgi.py`:

```
• nano ~/myproject/wsgi.py
```

The file is incredibly simple, we can simply 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.

Configure uWSGI

Our application is now written and our entry point established. We can now move on to uWSGI.

Testing uWSGI Serving

The first thing we will do is 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, as usual) plus the name of the callable within the application. In our case, this would be `wsgi:app`.

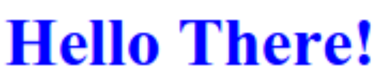
We'll also specify the socket so that it will be started on a publicly available interface and the protocol so that it will use HTTP instead of the `uwsgi` binary protocol. We'll use the same port number that we opened earlier:

```
• uwsgi --socket 0.0.0.0:5000 --protocol=http -w wsgi:app
```

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

```
http://server_domain_or_IP:5000
```

You should see your application's output again:


Hello There!

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:

```
• deactivate
```

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

Creating a uWSGI Configuration File

We have tested that uWSGI is able to serve our application, but we want something more robust for long-term usage. We can create a uWSGI configuration file with the options we want.

Let's place that 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 the module by referring to our `wsgi.py` file, minus the extension, and that the callable within the file is called "app":

~/myproject/myproject.ini

```
[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 we were testing, we exposed uWSGI on a network port. However, we'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 preferred because it is more secure and faster. We'll call the socket `myproject.sock` and place it in this directory.

We'll also have to 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 need to 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:

~/myproject/myproject.ini

```
[uwsgi]
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.

Create a systemd Unit File

The next piece we need to take care of is the systemd service unit file. Creating a systemd unit file will allow Ubuntu's init system to automatically start uWSGI and serve our 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. We'll 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
```

Next, we'll open up the `[Service]` section. We'll specify the user and group that we want the process to run under. We will give our regular user account ownership of the process since it owns all of the relevant files. We'll give group ownership to the `www-data` group so that Nginx can communicate easily with the uWSGI processes.

We'll then map out the working directory and set the `PATH` environmental variable so that the init system knows where our the executables for the process are located (within our virtual environment). We'll then specify the commanded 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:

/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, we'll 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:

/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

[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
```

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. We need to 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. We'll simply 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. We also need to tell it to use this block for requests for our server's domain name or IP address:

/etc/nginx/sites-available/myproject

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

The only other thing that we need to add is 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 server_domain_or_IP;

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

That's actually all we need to serve our application. Save and close the file when you're finished.

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

```
• sudo ln -s /etc/nginx/sites-available/myproject /etc/nginx/sites-enabl
```

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

```
• sudo nginx -t
```

If this returns without indicating any issues, we can restart the Nginx process to read the our new config:

```
• sudo systemctl restart nginx
```

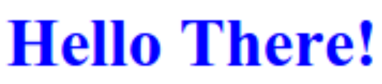
The last thing we need to do is adjust our firewall again. We no longer need access through port 5000, so we can remove that rule. We can then allow access to the Nginx server:

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

You should now be able to go to your server's domain name or IP address in your web browser:

```
http://server_domain_or_IP
```

You should see your application output:



Hello There!

Note

After configuring Nginx, the next step should be securing traffic to the server using SSL/TLS. This is important because without it, all information, including passwords are sent over the network in plain text.

The easiest way get an SSL certificate to secure your traffic is using Let's Encrypt. Follow this guide to set up Let's Encrypt with Nginx on Ubuntu 16.04.

Conclusion

In this guide, we've created a simple Flask application within a Python virtual environment. We create 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, we created a systemd service file to automatically launch the application server on boot. We created an Nginx server block that passes web client traffic to the application server, relaying external requests.

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.