TUTORIAL

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

Nginx Ubuntu Python Python Frameworks Ubuntu 18.04

By Justin Ellingwood and Kathleen Juell

Last Validated on December 3, 2021 • Originally Published on July 13, 2018 395.2k

English

Not using Ubuntu 18.04?
Choose a different version or distribution.

Ubuntu 18.04

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 Gunicorn application server and how to launch the application and configure Nginx to act as a front-end reverse proxy.

Prerequisites

To complete this tutorial, you will need:

- A server with Ubuntu 18.04 installed, a non-root user with sudo privileges, and a firewall enabled. 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 the WSGI specification, which the Gunicorn server will use to communicate with your Flask application. This discussion covers WSGI in more detail.

Step 1 — Installing the Components from the Ubuntu Repositories

The first step is to install all of the necessary packages from the default Ubuntu repositories. This includes `pip`, the Python package manager, which will manage your Python components. You'll also get the Python development files necessary to build some of the Gunicorn components.

First, update the local package:

```
$ sudo apt update
```

Then install the packages that will allow you to build your Python environment. These include `python3-pip`, 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-s
```

With these packages in place, move on to creating a virtual environment for your project.

Step 2 — Creating a Python Virtual Environment

Next, set up a virtual environment to isolate your 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, make a parent directory for yo

```
$ mkdir ~/myproject
```

Then change into the directory after you create it:

```
$ cd ~/myproject
```

Create a virtual environment to store your Flask project's Python requirements by entering the following:

```
$ python3.6 -m venv 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 by running the following:

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

Your prompt will change to indicate that you are now operating within the virtual environment. It will read like the following:

```
(myprojectenv)\ssammy@host:~/myproject$
```

Step 3 — Setting Up a Flask Application

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

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

```
(myprojectenv) $ pip install wheel
```

Note: Regardless of which version of Python you are using, when the virtual environment is activated, you should use the `pip` SCROLL TO TOP 3).

Next, install Flask and Gunicorn:

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

Now that you have Flask available, you can create a basic application in the next step.

Creating a Sample Application

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

While your application might be more complex, we'll create our Flask application in a single file, called `myproject.py`. Create this file using your preferred text editor; here we'll use `nano`:

```
(myprojectenv) $ nano ~/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:

~/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 defines what content to present when the root domain is accessed. Save and close the file when you're finished. If you're using nano, you can do this by pressing `CTRL + X` then `Y` and `ENTER`.

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

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

Then you can test your Flask application by running the following:

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

You will receive 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: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://your_server_ip:5000/ (Press CTRL+C to quit)
```

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

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

You should receive something like the following:

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

Creating the WSGI Entry Point

Next, create a file that will serve as the entry point for your application. This will tell your Gunicorn server how to interact with the application.

Create a new file using your preferred text editor and name it. Here, we'll call the file `wsgi.py`:

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

In this file, import the Flask instance from your 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 Gunicorn

Your application is now written with an entry point established and you can proceed to configuring Gunicorn.

But first, change into to the appropriate directory:

```
(myprojectenv) $ cd ~/myproject
```

Next, you can check that Gunicorn can serve the application correctly by passing it the name of your entry point. This is constructed as the name of the module (minus the `.py` extension), plus the name of the callable within the application. In our case, this is written as `wsgi:app`.

You'll also specify the interface and port to bind to so that the application will be started on a publicly available interface:

```
(myprojectenv) $ gunicorn --bind 0.0.0.0:5000 wsgi:app
```

You will receive output like the following:

```
Output
[2021-11-19 23:07:57 +0000] [8760] [INFO] Starting gunicorn 20.1.0
[2021-11-19 23:07:57 +0000] [8760] [INFO] Listening at: http://0.0.0.0:5000 (8760)
[2021-11-19 23:07:57 +0000] [8760] [INFO] Using worker: sync
[2021-11-19 23:07:57 +0000] [8763] [INFO] Booting worker with pid: 8763
[2021-11-19 23:08:11 +0000] [8760]   SCROLL TO TOP  gnal: int
[2021-11-19 23:08:11 +0000] [8760] [INFO] Shutting down: Master
```

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

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

Your application's output will generate the following:

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

Since now you're done with your virtual environment, deactivate it:

```
(myprojectenv) $ deactivate
```

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

Next, create the systemd service unit file. Creating a systemd unit file will allow Ubuntu's init system to automatically start Gunicorn 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, start with the `[Unit]` section, which is used to specify metadata and dependencies. Add a description of your 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=Gunicorn instance to serve myproject
After=network.target
```

Next, create a `[Service]` section. Th[illegible] user and group that you want the process to run under. Provide your regular user account ownership of the process since it

owns all of the relevant files. Also, give group ownership to the **www-data** group so that Nginx can communicate with the Gunicorn processes. Remember to replace the username here with your username:

/etc/systemd/system/myproject.service

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

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

Next, map out the working directory and set the `PATH` environment variable so that the init system knows that the executables for the process are located within your virtual environment. Also, specify the command to start the service. This command will do the following:

- Start 3 worker processes (though you should adjust this as necessary)
- Create and bind to a Unix socket file, `myproject.sock`, within your project directory.
- Set an umask value of `007` so that the socket file is created to give access to the owner and group, while restricting other access
- Specify the WSGI entry point file name, along with the Python callable within that file (`wsgi:app`)

Systemd requires that you give the full path to the Gunicorn executable, which is installed within your virtual environment.

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

/etc/systemd/system/myproject.service

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

[Service]
User=sammy
Group=www-data
WorkingDirectory=/home/sammy/myproject
Environment="PATH=/home/sammy/myproject/venv/bin"
```

```
ExecStart=/home/sammy/myproject/myprojectenv/bin/gunicorn --workers 3 --bind unix:myprc
app
```

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

/etc/systemd/system/myproject.service

```
[Unit]
Description=Gunicorn 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/gunicorn --workers 3 --bind unix:myprc

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

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

Now start the Gunicorn service you created:

```
$ sudo systemctl start myproject
```

Then enable it so that it starts at boot:

```
$ sudo systemctl enable myproject
```

Check the status:

```
$ sudo systemctl status myproject
```

You should receive output like the following:

Output

```
● myproject.service - Gunicorn instance to serve myproject
   Loaded: loaded (/etc/systemd/system/myproject.service; enabled; vendor preset
   Active: active (running) since Fri 2021-11-19 23:08:44 UTC; 6s ago
 Main PID: 8770 (gunicorn)
    Tasks: 4 (limit: 1151)
   CGroup: /system.slice/myproject.service
           ├─9291 /home/sammy/myproject/myprojectenv/bin/python3.6 /home/sammy/myproject/myprc
           ├─9309 /home/sammy/myproject/myprojectenv/bin/python3.6 /home/sammy/myproject/myprc
           ├─9310 /home/sammy/myproject/myprojectenv/bin/python3.6 /home/sammy/myproject/myprc
           └─9311 /home/sammy/myproject/myprojectenv/bin/python3.6 /home/sammy/myproject/myprc
…
```

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

Step 5 — Configuring Nginx to Proxy Requests

Your Gunicorn application server should now be up and running, waiting for requests on the socket file in the project directory. Next, configure Nginx to pass web requests to that socket by making some small additions to its configuration file.

Begin by creating a new server block configuration file in Nginx's `sites-available` directory. We'll call this `myproject` to stay consistent 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`. Also, tell it to use this block for requests for your server's domain name:

/etc/nginx/sites-available/myproject

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

Next, add a location block that matches every request. Within this block, include the `proxy_params` file that specifies some general proxying parameters that need to be set. Then pass the requests to the socket you defined using the `proxy_pass` directive:

/etc/nginx/sites-available/myproject

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

    location / {
        include proxy_params;
        proxy_pass http://unix:/home/sammy/myproject/myproject.sock;
    }
}
```

Save and close the file when you're finished.

To enable the Nginx server block configuration you've created, link the file to the `sites-enabled` directory. You can do this by running the `ln` command and the `-s` flag to create a symbolic or *soft* link, as opposed to a *hard link*:

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

With the link in that directory, you can test for syntax errors:

```
$ sudo nginx -t
```

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

```
$ sudo systemctl restart nginx
```

Finally, adjust the firewall again. Since you no longer need access through port `5000`, remove that rule:

```
$ sudo ufw delete allow 5000
```

Then allow full access to the Nginx server:

```
$ 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
```

Your application's output will appear in your browser:

If you encounter any errors, try 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.
- `sudo journalctl -u myproject`: checks your Flask app's Gunicorn logs.

Step 6 — Securing the Application

To ensure that traffic to your server remains secure, you should 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, install Certbot using `snap`:

```
$ sudo snap install --classic certbot
```

Your output will display the current version of Certbot and indicate a successful installation:

```
Output
certbot 1.21.0 from Certbot Project (certbot-eff✓) installed
```

Next, create a symbolic link to the newly installed `/snap/bin/certbot` executable from the `/usr/bin/` directory. This will ensure that the `certbot` command can run correctly on your server:

```
$ sudo ln -s /snap/bin/certbot /usr/bin/certbot
```

Certbot provides a variety of ways to obtain SSL certificates through plugins. The Nginx plugin will take care of reconfiguring Nginx and reloading the configuration 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 you'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 to request a certificate for your domain. If successful, you will receive the following output:

```
Output
Successfully received certificate.
Certificate is saved at: /etc/letsencrypt/live/jeanellehorcasitasphd.com/fullchain.pem
Key is saved at:         /etc/letsencrypt/live/jeanellehorcasitasphd.com/privkey.pem
This certificate expires on 2022-03-03.
These files will be updated when the certificate renews.
Certbot has set up a scheduled task to automatically renew this certificate in the backgrou

Deploying certificate
Successfully deployed certificate for your_domain to /etc/nginx/sites-enabled/myproject
Successfully deployed certificate for your_domain to /etc/nginx/sites-enabled/myproject
Congratulations! You have successfully enabled HTTPS on https://your_domain and https://yo

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
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, navigate once again to your domain, using `https://`:

```
https://your_domain
```

You should receive 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 basic 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 Gunicorn application server to provide this function. Afterward, 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 for relaying external requests and secured traffic to your server with Let's Encrypt.

Flask is an 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.

About the authors

Justin Ellingwood

Senior Technical Writer
@DigitalOcean

Kathleen Juell

Developer
@digitalocean/community