



# The Flask Mega-Tutorial, Part XVII: Deployment on Linux

Posted by [Miguel Grinberg](#) on [December 3, 2023](#) under [Raspberry Pi](#) [Cloud](#)  
[Python](#) [Flask](#) [Programming](#)

This is the seventeenth installment of the Flask Mega-Tutorial series, in which I'm going to deploy Microblog to a Linux server.

You are reading the 2024 edition of the Flask Mega-Tutorial. The complete course is also available to order in e-book and paperback formats from [Amazon](#). Thank you for your support!

If you are looking for the 2018 edition of this course, you can find it [here](#).

For your reference, here is the complete list of articles in this series:

- [Chapter 1: Hello, World!](#)
- [Chapter 2: Templates](#)
- [Chapter 3: Web Forms](#)
- [Chapter 4: Database](#)
- [Chapter 5: User Logins](#)
- [Chapter 6: Profile Page and Avatars](#)
- [Chapter 7: Error Handling](#)
- [Chapter 8: Followers](#)
- [Chapter 9: Pagination](#)
- [Chapter 10: Email Support](#)
- [Chapter 11: Facelift](#)
- [Chapter 12: Dates and Times](#)
- [Chapter 13: I18n and L10n](#)
- [Chapter 14: Ajax](#)
- [Chapter 15: A Better Application Structure](#)
- [Chapter 16: Full-Text Search](#)
- [Chapter 17: Deployment on Linux](#) (this article)
- [Chapter 18: Deployment on Heroku](#)
- [Chapter 19: Deployment on Docker Containers](#)
- [Chapter 20: Some JavaScript Magic](#)
- [Chapter 21: User Notifications](#)

- [Chapter 22: Background Jobs](#)
- [Chapter 23: Application Programming Interfaces \(APIs\)](#)

In this chapter I'm reaching a milestone in the life of my Microblog application, as I'm going to discuss ways in which the application can be deployed on a production server so that it is accessible to real users.

The topic of deployment is extensive, and for that reason it is impossible to cover all the possible options here. This chapter is dedicated to explore traditional hosting options, and as subjects I'm going to use a dedicated Linux server running Ubuntu, and also the widely popular Raspberry Pi mini-computer. I will cover other options such as cloud and container deployments in later chapters.

*The GitHub links for this chapter are: [Browse](#), [Zip](#), [Diff](#).*

## Traditional Hosting

When I refer to "traditional hosting", what I mean is that the application is installed manually or through a scripted installer on a stock server machine. The process involves installing the application, its dependencies and a production scale web server and configure the system so that it is secure.

The first question you need to ask when you are about to deploy your own project is where to find a server. These days there are many economic hosting services. For example, for \$5 per month, [Digital Ocean](#), [Linode](#), or [Amazon Lightsail](#) will rent you a virtualized Linux server in which to run your deployment experiments (Linode and Digital Ocean provision their entry level servers with 1GB of RAM, while Amazon provides only 512MB). If you prefer to practice deployments without spending any money, then [Vagrant](#) and [VirtualBox](#) are two tools that combined allow you to create a virtual server similar to the paid ones on your own computer.

As far as operating system choices, from a technical point of view, this application can be deployed on any of the major operating systems, a list which includes a large variety of open-source Linux and BSD distributions, and the commercial macOS and Microsoft Windows (macOS is a hybrid open-source/commercial option as it is based on Darwin, an open-source BSD derivative).

Since macOS and Windows are desktop operating systems that are not optimized to work as servers, I'm going to discard those as candidates. The choice between a Linux or a BSD operating system is largely based on preference, so I'm going to pick the most popular of the two, which is Linux. As far as Linux distributions, once again I'm going to choose by popularity and go with Ubuntu.

## Creating an Ubuntu Server

If you are interested in doing this deployment along with me, you obviously need a server to work on. I'm going to recommend two options for you to acquire a server, one paid and one free. If you are willing to spend a bit of money, you can get an account at Digital Ocean, Linode or Amazon Lightsail and create an Ubuntu virtual server with the current long-term support (LTS) release, which is 22.04 at the time I'm writing this. You should use the smallest server option, the one that costs around \$5 per month. The cost is prorated to the number of hours that you have the server up, so if you create the server, play with it for a few hours and then delete it, you would be paying just cents.

The free alternative is based on a virtual machine that you can run on your own computer. To use this option, install [Vagrant](#) and [VirtualBox](#) on your machine, and then create a file named *Vagrantfile* to describe the specs of your VM with the following contents:

*Vagrantfile*: Vagrant configuration.

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/jammy64"
  config.vm.network "private_network", ip: "192.168.56.10"
  config.vm.provider "virtualbox" do |vb|
    vb.memory = "2048"
  end
end
```

This file configures a Ubuntu 22.04 server with 2GB of RAM, which you will be able to access from the host computer at IP address 192.168.56.10. To create the server, run the following command:

```
$ vagrant up
```

Consult the Vagrant [command-line documentation](#) to learn about other options to manage your virtual server.

## Using an SSH Client

Your server is headless, so you are not going to have a desktop on it like you have on your own computer. You are going to connect to your server through a SSH client and work on it through the command-line. If you are using Linux or Mac OS X, you likely have [OpenSSH](#) already installed. If you are using Microsoft Windows, [Cygwin](#), [Git](#), and the [Windows Subsystem for Linux](#) provide OpenSSH, so you can install any of these options.

When using a virtual server from a third-party provider, you were given an IP address for it. You can open a terminal session with your brand-new server with the following command:

```
$ ssh root@<server-ip-address>
```

You will be prompted to enter a password. Depending on the service, the password may have been automatically generated and shown to you after you created the server, or you may have given the option to choose your own password.

If you are using a Vagrant VM, you can open a terminal session using the command:

```
$ vagrant ssh
```

If you are using Windows and have a Vagrant VM, note that you will need to run the above command from a shell that can invoke the `ssh` command from OpenSSH.

## Password-less Logins

If you are using a Vagrant VM, you can skip this section, since your VM is properly configured to use a non-root account named `vagrant` or `ubuntu`, without password automatically by Vagrant.

When using a virtual server, it is recommended that you create a regular user account to do your deployment work, and configure this account to log you in without using a password, which at first may seem like a bad idea, but you'll see that it is not only more convenient but also more secure.

I'm going to create a user account named `ubuntu` (you can use a different name if you prefer). To create this user account, log in to your server's root account using the `ssh` instructions from the previous section, and then type the following commands to create the user, give it `sudo` powers, and finally switch to it:

```
$ adduser --gecos "" ubuntu
$ usermod -aG sudo ubuntu
$ su ubuntu
```

Now I'm going to configure this new `ubuntu` account to use [public key](#) authentication so that you can log in without having to type a password.

Leave the terminal session you have open on your server for a moment, and start a second terminal on your local machine. If you are using Windows, this needs to be the terminal from where you have access to the `ssh` command, so it will probably be a `bash` or similar prompt and not a native Windows terminal. In that terminal session, check the contents of the `~/.ssh` directory:

```
$ ls ~/.ssh
id_rsa  id_rsa.pub
```

If the directory listing shows files named `id_rsa` and `id_rsa.pub` like above, then you already have a key. If you don't have these two files, or if you don't have the `~/.ssh` directory at all, then you need to create your SSH keypair by running the following command, also part of the OpenSSH toolset:

```
$ ssh-keygen
```

This application will prompt you to enter a few things, for which I recommend you accept the defaults by pressing Enter on all the prompts. If you know what you are doing and want to do otherwise, you certainly can.

After this command runs, you should have the two files listed above. The file `id_rsa.pub` is your *public key*, which is a file that you will provide to third parties as a way to identify you. The `id_rsa` file is your *private key*, which should not be shared with anyone.

You now need to configure your public key as an *authorized host* in your server. On the terminal that you opened on your own computer, print your public key to the screen:

```
$ cat ~/.ssh/id_rsa.pub  
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQjw...F8Xv4f/0+7WT miguel@migue
```

This is going to be a very long sequence of characters, possibly spanning multiple lines. You need to copy this data to the clipboard, and then switch back to the terminal on your remote server, where you will issue these commands to store the public key:

```
$ echo <paste-your-key-here> >> ~/.ssh/authorized_keys  
$ chmod 600 ~/.ssh/authorized_keys
```

The password-less login should now be working. The idea is that `ssh` on your machine will identify itself to the server by performing a cryptographic operation that requires the private key. The server then verifies that the operation is valid using your public key.

You can now log out of your `ubuntu` session, and then from your `root` session, and then try to log in directly to the `ubuntu` account with:

```
$ ssh ubuntu@<server-ip-address>
```

This time you should not have to enter a password!

## Securing Your Server

To minimize the risk of your server being compromised, there are a few steps that you can take, directed at closing a number of potential doors through which an attacker may gain access.

The first change I'm going to make is to disable root logins via SSH. You now have password-less access into the `ubuntu` account, and you can run administrator commands from this account via `sudo`, so there is really no need to expose the root account. To disable root logins, you need to edit the `/etc/ssh/sshd_config` file on your server. You probably have the `vi` and `nano` text editors installed in your server that you can use to edit files (if you are not familiar with either one, try `nano` first). You

will need to prefix your editor with `sudo`, because the SSH configuration is not accessible to regular users (i.e. `sudo vi /etc/ssh/sshd_config`). You need to change a single line in this file:

```
/etc/ssh/sshd_config: Disable root logins.
```

```
PermitRootLogin no
```

Note that to make this change you need to locate the line that starts with `PermitRootLogin` and change the value, whatever that might be in your server, to `no`.

The next change is in the same file. Now I'm going to disable password logins for all accounts. You have a password-less login set up, so there is no need to allow passwords at all. If you feel nervous about disabling passwords altogether you can skip this change, but for a production server it is a good idea, since attackers are constantly trying random account names and passwords on all servers hoping to get lucky. To disable password logins, change the following line in */etc/ssh/sshd\_config*:

```
/etc/ssh/sshd_config: Disable password logins.
```

```
PasswordAuthentication no
```

After you are done editing the SSH configuration, the service needs to be restarted for the changes to take effect:

```
$ sudo service ssh restart
```

The third change I'm going to make is to install a *firewall*. This is a software that blocks accesses to the server on any ports that are not explicitly enabled:

```
$ sudo apt-get install -y ufw
```

```
$ sudo ufw allow ssh
```

```
$ sudo ufw allow http
```

```
$ sudo ufw allow 443/tcp
```

```
$ sudo ufw --force enable
```

```
$ sudo ufw status
```

These commands install ufw, the Uncomplicated Firewall, and configure it to only allow external traffic on port 22 (ssh), 80 (http) and 443 (https).

Any other ports will not be allowed.

## Installing Base Dependencies

If you followed my advice and provisioned your server with the Ubuntu 20.04 release, then you have a system that comes with full support for Python 3.8, so this is the release that I'm going to use for the deployment.

The base Python interpreter is probably pre-installed on your server, but there are some extra packages that are likely not, and there are also a few other packages outside of Python that are going to be useful in creating a robust, production-ready deployment. For a database server, I'm going to switch from SQLite to MySQL. The postfix package is a mail transfer agent that I will use to send out emails. The supervisor tool will monitor the Flask server process and automatically restart it if it ever crashes, or also if the server is rebooted. The nginx server is going to accept all requests that come from the outside world, and forward them to the application. Finally, I'm going to use git as my tool of choice to download the application directly from its git repository.

```
$ sudo apt-get -y update
$ sudo apt-get -y install python3 python3-venv python3-dev
$ sudo apt-get -y install mysql-server postfix supervisor nginx git
```

These installations run mostly unattended. Depending on the Ubuntu version you are installing, you may get a prompt asking for services to restart, which you can accept with the default selections. While you run the third install statement you will be asked a couple of questions regarding the installation of the postfix package which you can also accept with their default answers.

Note that for this deployment I'm choosing not to install Elasticsearch. This service requires a large amount of RAM, so it is only viable if you have a large server with more than 2GB of RAM. To avoid problems with the server running out of memory I will leave the search functionality out. If you have a big enough server, you can download the official .deb package from the [Elasticsearch site](#) and follow their installation instructions to add it to your server.

I should also note that the default installation of postfix is likely insufficient for sending email in a production environment. To avoid



spam and malicious emails, many servers require the sender server to identify itself through security extensions, which means at the very least you have to have a domain name associated with your server. If you want to learn how to fully configure an email server so that it passes standard security tests, see the following Digital Ocean guides:

- [Postfix Configuration](#)
- [Adding an SPF Record](#)
- [DKIM Installation and Configuration](#)

## Installing the Application

Now I'm going to use `git` to download the Microblog source code from my GitHub repository. I recommend that you read [git for beginners](#) if you are not familiar with git source control.

To download the application to the server, make sure you are in the `ubuntu` user's home directory and then run:

```
$ git clone https://github.com/miguelgrinberg/microblog
$ cd microblog
$ git checkout v0.17
```

This installs the code on your server, and syncs it to this chapter. If you are keeping your version of this tutorial's code on your own git repository, you can change the repository URL to yours, and in that case you can skip the `git checkout` command.

Now I need to create a virtual environment and populate it with all the package dependencies, which I conveniently saved to the *requirements.txt* file in [Chapter 15](#):

```
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $ pip install -r requirements.txt
```

In addition to the common requirements in *requirements.txt*, I'm going to use three packages that are specific to this production deployment, so they are not included in the common requirements file. The `unicorn` package is a production web server for Python applications. The `pymysql` package contains the MySQL driver that enables SQLAlchemy

to work with MySQL databases. The `cryptography` package is used by `pymysql` to authenticate against the MySQL database server.

```
(venv) $ pip install gunicorn pymysql cryptography
```

I need to create a `.env` file, with all the needed environment variables:

`/home/ubuntu/microblog/.env`. Environment configuration.

```
SECRET_KEY=52cb883e323b48d78a0a36e8e951ba4a
MAIL_SERVER=localhost
MAIL_PORT=25
DATABASE_URL=mysql+pymysql://microblog:<db-password>@localhost:3306/mi
MS_TRANSLATOR_KEY=<your-translator-key-here>
```

This `.env` file is mostly similar to the example I shown in [Chapter 15](#), but I have used a random string for `SECRET_KEY`. You should generate your own secret key here. You can use the following command:

```
python3 -c "import uuid; print(uuid.uuid4().hex)"
```

For the `DATABASE_URL` variable I defined a MySQL URL. I will show you how to configure the database in the next section.

I need to set the `FLASK_APP` environment variable to the entry point of the application to enable the `flask` command to work. If you do not have a `.flaskenv` file in your project's repository, then it is time to add one. You can confirm that the `FLASK_APP` variable is configured by running `flask --help`. If the help message shows the `translate` command added by the application, then you know the application was found.

And now that the `flask` command is functional, I can compile the language translations:

```
(venv) $ flask translate compile
```

## Setting Up MySQL

The SQLite database that I've used during development is great for simple applications, but when deploying a full-blown web server that can potentially need to handle multiple requests at a time, it is better to use a

more robust database. For that reason I'm going to set up a MySQL database that I will call `microblog`.

To manage the database server I'm going to use the `mysql` command, which should be already installed on your server:

```
$ sudo mysql -u root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 8.0.25-0ubuntu0.20.04.1 (Ubuntu)

Copyright (c) 2000, 2021, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement

mysql>
```

Note that you will need to use `sudo` to access the MySQL root user from the administrator account.

These are the commands that create a new database called `microblog`, and a user with the same name that has full access to it:

```
mysql> create database microblog character set utf8 collate utf8_bin;
mysql> create user 'microblog'@'localhost' identified by '<db-password>';
mysql> grant all privileges on microblog.* to 'microblog'@'localhost';
mysql> flush privileges;
mysql> quit;
```

You will need to replace `<db-password>` with a password of your choice. The password that you select here needs to match the password that you included in the `DATABASE_URL` variable in the `.env` file.

If your database configuration is correct, you should now be able to run the database migrations that create all the tables:

```
(venv) $ flask db upgrade
```

Make sure the above command completes without producing any errors before you continue.

## Setting Up Gunicorn and Supervisor

When you run the server with `flask run`, you are using a web server that comes with Flask. This server is very useful during development, but it isn't a good choice to use for a production server because it wasn't built with performance and robustness in mind. Instead of the Flask development server, for this deployment I decided to use [Gunicorn](#), which is also a pure Python web server, but unlike Flask's, it is a robust production server that is used by a lot of people, while at the same time being very easy to configure.

To start Microblog under Gunicorn you can use the following command:

```
(venv) $ gunicorn -b localhost:8000 -w 4 microblog:app
```

The `-b` option tells Gunicorn where to listen for requests, which I set to the internal network interface at port 8000. It is usually a good idea to run Python web applications without external access, and then have a very fast web server that is optimized to serve static files accepting all requests from clients. This fast web server will serve static files directly, and forward any requests intended for the application to the internal server. I will show you how to set up nginx as the public facing server in the next section.

The `-w` option configures how many *workers* Gunicorn will run. Having four workers allows the application to handle up to four clients concurrently, which for a web application is usually enough to handle a decent amount of clients, since not all of them are constantly requesting content. Depending on the amount of RAM your server has, you may need to adjust the number of workers so that you don't run out of memory.

The `microblog:app` argument tells Gunicorn how to load the application instance. The name before the colon is the module that contains the application, which for this application is `microblog.py`. The name after the colon is the name of the application instance.

While Gunicorn is very simple to set up, running the server from the command-line is actually not a good solution for a production server. What I want to do is have the server running in the background, and have it under constant monitoring, because if for any reason the server crashes and exits, I want to make sure a new server is automatically started to take its place. And I also want to make sure that if the machine is rebooted, the server runs automatically upon startup, without me having to log in and start things up myself. I'm going to use the [supervisor](#) package that I installed above to do this.

The supervisor utility uses configuration files that tell it what programs to monitor and how to restart them when necessary. Configuration files must be stored in `/etc/supervisor/conf.d`. Here is a configuration file for Microblog, which I'm going to call *microblog.conf*.

*/etc/supervisor/conf.d/microblog.conf*. Supervisor configuration.

```
[program:microblog]
command=/home/ubuntu/microblog/venv/bin/gunicorn -b localhost:8000 -w
directory=/home/ubuntu/microblog
user=ubuntu
autostart=true
autorestart=true
stopasgroup=true
killasgroup=true
```

The `command`, `directory` and `user` settings tell supervisor how to run the application. The `autostart` and `autorestart` set up automatic restarts due to the computer starting up, or crashes. The `stopasgroup` and `killasgroup` options ensure that when supervisor needs to stop the application to restart it, it also reaches the child processes of the top-level Gunicorn process.

After you write this configuration file, you have to reload the supervisor service for it to be imported:

```
$ sudo supervisorctl reload
```

And just like that, the Gunicorn web server should be up and running and monitored!

## Setting Up Nginx

The microblog application server powered by Gunicorn is now running privately on port 8000. What I need to do now to expose the application to the outside world is to enable my public facing web server on ports 80 and 443, the two ports that I opened on the firewall to handle the web traffic of the application.

I want this to be a secure deployment, so I'm going to configure port 80 to forward all traffic to port 443, which is going to be encrypted. So I'm going to start by creating an SSL certificate. For now, I'm going to create a *self-signed SSL certificate*, which is okay for testing everything but not good for a real deployment because web browsers will warn users that the certificate was not issued by a trusted certificate authority. The command to create the SSL certificate for microblog is:

```
$ mkdir certs
$ openssl req -new -newkey rsa:4096 -days 365 -nodes -x509 \
  -keyout certs/key.pem -out certs/cert.pem
```

The command is going to ask you for some information about your application and yourself. This is information that will be included in the SSL certificate, and that web browsers will show to users if they request to see it. The result of the command above is going to be two files called *key.pem* and *cert.pem*, which I placed in a *certs* subdirectory of the Microblog root directory.

To have a website served by nginx, you need to write a configuration file for it. In most nginx installations this file needs to be in the */etc/nginx/sites-available* directory. Below you can see the nginx configuration file for Microblog, which goes in */etc/nginx/sites-available/microblog*:

*/etc/nginx/sites-available/microblog*: Nginx configuration.

```
server {
    # listen on port 80 (http)
    listen 80;
    server_name _;
    location / {
        # redirect any requests to the same URL but on https
        return 301 https://$host$request_uri;
    }
}

server {
    # listen on port 443 (https)
    listen 443 ssl;
```

```

server_name _;

# location of the self-signed SSL certificate
ssl_certificate /home/ubuntu/microblog/certs/cert.pem;
ssl_certificate_key /home/ubuntu/microblog/certs/key.pem;

# write access and error logs to /var/log
access_log /var/log/microblog_access.log;
error_log /var/log/microblog_error.log;

location / {
    # forward application requests to the gunicorn server
    proxy_pass http://localhost:8000;
    proxy_redirect off;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}

location /static {
    # handle static files directly, without forwarding to the appl
    alias /home/ubuntu/microblog/app/static;
    expires 30d;
}
}

```

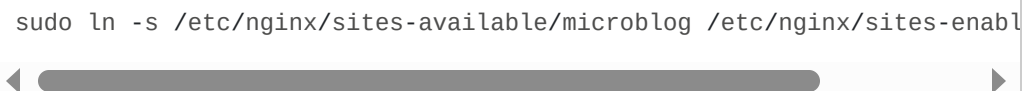
The nginx configuration is far from trivial, but I've added some comments so that at least you know what each section does. If you want to have information about a specific directive, consult the [nginx official documentation](#).

The website is now configured, but it isn't enabled yet. To enable it, a link to this file must be created in the `/etc/nginx/sites-enabled` directory. Nginx comes with a test site that is enabled that I don't really need, so I'm going to start by removing it:



```
$ sudo rm /etc/nginx/sites-enabled/default
```

Now I can create a link to the microblog configuration:



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

After you add this file, you need to tell nginx to reload the configuration:

```
$ sudo service nginx reload
```

And now the application should be deployed. In your web browser, you can type the IP address of your server (or 192.168.56.10 if you are using a Vagrant VM) and that will connect to the application. Because you are using a self-signed certificate, you will get a warning from the web browser, which you will have to dismiss.

After you complete a deployment with the above instructions for your own projects, I strongly suggest that you replace the self-signed certificate with a real one, so that the browser does not warn your users about your site. For this you will first need to purchase a domain name and configure it to point to your server's IP address. Once you have a domain, you can request a free [Let's Encrypt](#) SSL certificate. I have written a detailed article on my blog on how to [Run your Flask application over HTTPS](#).

## Deploying Application Updates

The last topic I want to discuss regarding the Linux based deployment is how to handle application upgrades. The application source code is installed in the server through `git`, so whenever you want to upgrade your application to the latest version, you can just run `git pull` to download the new commits that were made since the previous deployment.

But of course, downloading the new version of the code is not going to cause an upgrade. The server processes that are currently running will continue to run with the old code, which was already read and stored in memory. To trigger an upgrade you have to stop the current server and start a new one, to force all the code to be read again.

Doing an upgrade is in general more complicated than just restarting the server. You may need to apply database migrations, or compile new language translations, so in reality, the process to perform an upgrade involves a sequence of commands:

```
(venv) $ git pull                                # download the new vers
(venv) $ sudo supervisorctl stop microblog       # stop the current serv
(venv) $ flask db upgrade                         # upgrade the database
```



```
(venv) $ flask translate compile          # upgrade the translati
(venv) $ sudo supervisorctl start microblog  # start a new server
```

## Raspberry Pi Hosting

The [Raspberry Pi](#) is a low-cost revolutionary little Linux computer that has very low power consumption, so it is the perfect device to host a home based web server that can be online 24/7 without tying up your desktop computer or laptop. There are several Linux distributions that run on the Raspberry Pi. My choice is Raspberry Pi OS, which is the official distribution from the Raspberry Pi Foundation.

To prepare the Raspberry Pi, I'm going to install a fresh Raspberry Pi OS release. I will be using the Lite version, because I do not need the desktop user interface. You can find the latest release of the Raspberry Pi OS on their [operating systems page](#).

The Raspberry Pi OS image needs to be installed on an SD card, which you then plug into the Raspberry Pi so that it can boot with it. Instructions to copy the Raspberry Pi OS image to an SD card from Windows, Mac OS X and Linux are available on the [Raspberry Pi site](#).

When you boot your Raspberry Pi for the first time, do it while connected to a keyboard and a monitor, so that you can do the setup. At the very least you should enable SSH, so that you can log in from your computer to perform the deployment tasks more comfortably.

Like Ubuntu, Raspberry Pi OS is a derivative of Debian, so the instructions above for Ubuntu Linux for the most part work just as well for the Raspberry Pi. However, you may decide to skip some steps if you are planning on running a small application on your home network, without external access. For example, you may not need the firewall, or the password-less logins. And you may want to use SQLite instead of MySQL in such a small computer. You may opt to not use nginx, and just have the Gunicorn server listening directly for requests from clients. You will probably want just one or two Gunicorn workers. The supervisor service is useful in ensuring the application is always up, so my recommendation is that you also use it on the Raspberry Pi.

Continue on to the [next chapter](#).

## Buy me a coffee?

Thank you for visiting my blog! If you enjoyed this article, please consider supporting my work and keeping me caffeinated with a small one-time donation through [Buy me a coffee](#). Thanks!



## Share this post

Hacker News

Reddit

Twitter

LinkedIn

Facebook

E-Mail

49 comments



#1 **Michael Gutierrez** said 2 years ago

Hi Miguel, Thanks for sharing this guide, it's been really helpful. One issue I came across when setting up a linux host, was `url_for` serving http links after following your guide. Wrapping app w/ `werkzeug's proxy_fix` [https://werkzeug.palletsprojects.com/en/2.3.x/middleware/proxy\\_fix/](https://werkzeug.palletsprojects.com/en/2.3.x/middleware/proxy_fix/) and adding `X-Forwarded-Proto $schema` to the nginx server block fixed the issue. Have you encountered this as well and Is there another solution I should be considering?

Thanks again!



#2 **Miguel Grinberg** said 2 years ago

@Michael: What you are doing is the correct way to handle this problem. But not that `url_for()` in its normal usage returns relative URLs, which only have the path component of the URL. These should work fine for both `http://` and `https://` deployments. Your problem only exists when you request absolute URLs.



#3 **jimbo\_jones\_ii** said a year ago

Hi Miguel, great chapter. I'm a Linux newbie so I found it tough. One issue I struggled with but resolved was the passwordless access. Using Windows and DO I created a ubuntu droplet and then the ubuntu user and switched to it in Windows command prompt. when I tried echoing the SSH public key I got error "/home/ubuntu/.ssh/authorized\_keys: No such file or directory". After lots of trial and error I ended up exiting out of the ubuntu shell (ubuntu@ubuntu:/root\$ and ubuntu@ubuntu:/~\$) which I had switched to per the tutorial instructions after creating that ubuntu user. I was now at the root user prompt (root@ubuntu:~#) and was able to navigate to hidden directory /.ssh and then run the echo command with the SSH pub key successfully. I logged out of the DO droplet and SSH logged back in passwordlessly. If its helpful to update the chapter with these steps for other please do. thanks again!



#4 **Miguel Grinberg** said a year ago

@jimbo: this is explained in the tutorial. If the files in the `.ssh` directory do not exist, or if the whole directory is missing, then you have to use the `ssh-keygen` command to create your keypair. Dropping into the root account is not the correct solution, you do not want to use the root account to log in to your server.



#5 **Alex** said a year ago

I did everything according to your instructions... but the statics are not loaded (403) I checked all the paths. everything is fine... but it doesn't load static at all....

User

2024/05/13 03:46:42 [notice] 63903#63903: using inherited sockets from "6;7;"

2024/05/13 09:45:19 [notice] 74763#74763: signal process started

2024/05/13 11:46:21 [alert] 77972#77972: 28 open socket #20 left in connection 6

2024/05/13 11:46:21 [alert] 77972#77972: 26 open socket #19

left in connection 7

2024/05/13 11:46:21 [alert] 77973#77973: \*27 open socket #17

left in connection 3

2024/05/13 11:46:21 [alert] 77972#77972: aborting

2024/05/13 11:46:21 [alert] 77973#77973: aborting



#6 Miguel Grinberg said a year ago

@Alex: make sure that all the directories leading up to the static folder have permissions that allow access by everyone. Typically you want 755 on /home, /home/ubuntu, /home/ubuntu/microblog and so on.



#7 Jeff said a year ago

I feel like I'm in the deep end on this tutorial to begin with and this chapter is eating my lunch. I've only been using Python for 6 months. At Nginx step I get to `sudo rm /etc/nginx/sites-enabled/default`. When I try it I get

```
(venv) vagrant@ubuntu-jammy:~/microblog$ sudo rm /etc/
rm: cannot remove '/etc/nginx/sites-enabled/default':
```

then

```
(venv) vagrant@ubuntu-jammy:~/microblog$ sudo ln -s /
ln: failed to create symbolic link '/etc/nginx/sites-e
```

Then

```
(venv) vagrant@ubuntu-jammy:~/microblog$ ls -l /home/
ls: cannot access '/home/ubuntu/microblog/certs/cert.p
```

then

```
(venv) vagrant@ubuntu-jammy:~/microblog$ ls -l /home/
ls: cannot access '/home/ubuntu/microblog/certs/': Per
```

I'm on Windows 11 with Oracle VM VirtualBox Manager and Vagrant 2.4.1 amd64.

```
(venv) vagrant@ubuntu-jammy:~/microblog$ sudo service  
Job for nginx.service failed because the control process
```



#8 **Miguel Grinberg** said a year ago

@Jeff: some comments about your issues:

For the first issue, you are trying to delete a file and the system is telling the file does not exist. This happens probably because you already deleted the file. Once you do the delete, if you run this command again you will receive an error, since obviously you can only delete a file once.

Second issue is related. You are trying to create a symbolic link, and the system is telling you there is a file with the name you want to use for this symbolic link. This is likely because you have already created the link and now are re-running the command, which is also expected to fail once you've done it once.

For the third one you may need to use `sudo`. This is because the certificate files are owned by the root user.

For the fourth one you likely have a mistake in the nginx configuration file. This prevents nginx from restarting.



#9 **Jeff** said a year ago

@Miguel:

I finally successfully completed this chapter! A few things of note.

Because I was using Vagrant VM I needed to change ubuntu to vagrant in the microblog file for nginx server configuration.

Additionally, my versions did not match on Vagrant and vbguest.

```
$ vagrant plugin install vagrant-vbguest
```

```
$vagrant halt
```

```
$vagrant up
```

```
$vagrant halt
```

```
$vagrant up
```

Now Guest Additionas showed updated version.

I do not know if it matters but on Oracle VM VirtualBox Manager I went to Devices>Upgrade Guest Additions before executing the remainder of the chapter.

Either way ended up with microblog running from my VM. Thank you again for your reply that got me back on track. Maybe this comment will help someone else.



#10 Miguel Grinberg said a year ago

@Jeff: Congratulations! The guest additions are optional, there is no need to set that up when you are creating a test VM that you do not intend to keep for long so you can ignore any warnings about guest additions missing or having mismatched versions.

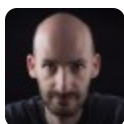


#11 William said a year ago

Hello Miguel, incredible tutorial, as someone who only knew HTML CSS, JavaScript and a bit of Docker, I feel like I have a deeper understanding of web development as a whole thanks to you.

I have a question for this chapter. I am using the docker version of Elasticsearch, however, this container is completely removed after I'm done, meaning that the Elasticsearch index is also removed. This means that, as of right now, I have to manually go into the python console to use `>>> Post.reindex()` to recreate the index each time I launch the container. Now, I know that, if I deploy the app with docker, I won't have to completely remove the Elasticsearch container, but would you know of a way to use `>>> Post.reindex()` each time the container is initialized, maybe through a dockerfile ?

And again, thank you for the incredible tutorial.



#12 Miguel Grinberg said a year ago

@William: the Docker installation instructions I used in this tutorial are only good for local use. For a production deployment of Elasticsearch that has proper data persistence you will need to consult their official documentation, or use a hosted solution.



#13 **Pascal** said a year ago

Hi, great blog and tutorial which helped me a lot! I have problems configuring Nginx to serve static files. I configured everything as described but it's not working. I suspect that the Nginx user has no permissions to access the files within my users home directory. How did you manage to solve this problem?

Thx again for all your work!



#14 **Miguel Grinberg** said a year ago

@Pascal: I don't recall having this problem myself. What permissions do you have on your static files? It should be fairly easy to check if this is the problem.



#15 **Pascal** said a year ago

@Miguel: I needed to grant ,Others' permission to read/ list to my users home directory. After that Nginx could load the static files. But that approach seems to be a bit insecure?



#16 **Miguel Grinberg** said a year ago

@Pascal: "others" in this context means other users on the same machine. I believe it is pretty standard to have "755" permissions, which gives group and other users read and execute access, but if you prefer not to, then you can either add your **ubuntu** account to the **www-data** group, so then the permissions that control access will be the group permissions, or else you can also change the ownership of the files to **www-data** and then the nginx process will see them as its own files. Really up to you how you want to configure the file access.



#17 **Skibity** said a year ago

@Miguel: Thanks so much for taking the time to produce and maintain such a wonderful tutorial.

I just wanted to share that I too experienced an issue that a couple of others had mentioned above regarding static files not being properly served (error 403 - forbidden). As far as I know, I followed the tutorial step-by-step, but the loading image that displays during translation was failing to load. I am using a Linode server.

I followed your advice in the comments above and gave 755 permissions to /home, /home/ubuntu, etc. all the way up to the static directory and this fixed the issue, but also created a new one. When I attempted to pull the changes, I received the following warning:

Permissions 0755 for '/home/ubuntu/.ssh/id\_ed25519' are too open.

It is required that your private key files are NOT accessible by others.

This private key will be ignored.

I then had to run `sudo chmod 600 ~/.ssh/id_ed25519` in order to be able to pull. Just wanted to share in case others experienced this issue. Thanks again!



#18 **Kenneth** said a year ago

@ Miguel: Thanks for the tutorial.

I tried running gunicorn and it failed to find the attribute app.

```
(myvenv) ubuntu@ubuntu-2cpu-2gb-sg-sin1:~$ gunicorn -b
localhost:8000 -w 4 kaixin:app
```

```
[2024-09-03 20:49:20 +0000] [11575] [INFO] Starting gunicorn
23.0.0
```

```
[2024-09-03 20:49:20 +0000] [11575] [INFO] Listening at:
http://127.0.0.1:8000 (11575)
```

```
[2024-09-03 20:49:20 +0000] [11575] [INFO] Using worker: sync
```

```
[2024-09-03 20:49:20 +0000] [11578] [INFO] Booting worker with
pid: 11578
```

Failed to find attribute 'app' in 'kaixin'.

What could I have missed?

#19 **Miguel Grinberg** said a year ago





@Kenneth: Do you have a "[kaixin.py](#)" file? The error is saying that there is no "app" variable in this file.



#20 **Kenneth** said a year ago

@ Miguel: I have the [kaixin.py](#) file. It was a rather silly mistake that I made. Earlier, I forgot to import the models for the shell context processor. After importing the models, as shown below, I was able to get Gunicorn to run.

```
from app import create_app, db, socketio

from app.cart.models import Cart, CartItem
from app.country.models import Country
from app.currency.models import Currency
from app.product.models import Product
from app.user.models import User

app = create_app()

@app.shell_context_processor
def make_shell_context():
    return {'db': db,
            'Cart': Cart,
            'CartItem': CartItem,
            'Country': Country,
            'Currency': Currency,
            'Product': Product,
            'User': User}

if __name__ == '__main__':
    socketio.run(app)
```



#21 **Ponniah** said 10 months ago

@Miguel: Thank you for sharing the in-depth tutorials. They helped me to learn the concepts and build apps. For my use case, I have built a Flask based API and I am hosting it on a server on just a single port. I am using Gunicorn with Uvicorn workers and I have set the number of worker processes as per the formula provided in Gunicorn documentation - based on number of CPU cores ( $2 * \text{CPU Cores} + 1$ ). I read in a StackOverflow post that Gunicorn relies on OS to do load balancing. However, in practice, I am noticing that the

distribution is highly skewed and one worker process is loaded with a significantly larger number of requests. Can you please share your experiences about load distribution in Gunicorn servers and whether this skewed distribution is a common trend? Can you please suggest ways to overcome this issue?



#22 Miguel Grinberg said 10 months ago

@Ponniah: what you read is correct, Gunicorn does not offer any options to configure load balancing, it relies on the OS for this. If you need to have more flexible load balancing you should use a load balancer such as Nginx or HAProxy.



#23 John said 8 months ago

@Miguel: Your book has been uniformly excellent, but now in section 17.2 I've hit a problem with getting Vagrant to run Linux virtually on my Apple silicon Mac. Do you have expanded instructions for that? Today I've installed the current versions of Vagrant (v2.4.3) and VirtualBox (v7.1.4, which supposedly supports Apple silicon). But despite trying multiple guest operating systems in my Vagrantfile, the `\vagrant up\` command yields an error: "The box you're attempting to add doesn't support the provider you requested." For guests I've tried `bento/ubuntu-22.04-arm64`, `bento/ubuntu-24.04`, and `perk/ubuntu-2204-arm64`. When I tried `ubuntu/jammy64` a VirtualBox error resulted (no surprise): code `NS_ERROR_FAILURE` (0x80004005), component `SessionMachine`, interface `ISession`. Thanks for any suggestions.



#24 Miguel Grinberg said 8 months ago

@John: I'm not sure if there are `ubuntu/arm` boxes that you can use on Apple silicon. Another option that you have is to replace `vagrant` and `virtualbox` with [multipass](#), which is Ubuntu's own virtualization solution.



#25 cafeconhielo said 8 months ago

Fantastic, comprehensive instructions! However, for some reason I cannot get NGINX to work. I followed the instructions twice on two separate linodes. Both times, when I check `service nginx status`

I get the following error:

nginx.service - A high performance web server and a reverse proxy server

Loaded: loaded (/usr/lib/systemd/system/nginx.service; enabled; preset: enabled)

Active: failed (Result: exit-code) since Sun 2024-12-08 04:28:40 UTC; 7min ago

Duration: 38min 7.459s

Docs: man:nginx(8)

Process: 7350 ExecStartPre=/usr/sbin/nginx -t -q -g daemon on; master\_process on; (code=exited, status=1/FAILURE)

CPU: 5ms

Do you have any suggestions what could be causing this? Or tips on how to debug? I have successfully hosted static websites, and even a super simple Flask site on NGINX and it worked. For some reason, when I follow the instructions here I get this error. Any tips would be very appreciated. I love your tutorial and don't want to get stuck now that I have reached this point!



## Leave a Comment

Name

Email

Comment

## Captcha



I'm not a robot

reCAPTCHA  
Privacy - Terms

Submit

## The Flask Mega-Tutorial

**New 2024 Edition!**



If you would you like to support my work on my [Flask Mega-Tutorial series](#) on this blog and as a reward have access to the complete tutorial nicely structured as a book and/or a set of videos, you can now order it from my [Courses](#) site or from [Amazon](#).

[Click here to get the Book!](#)

[Click here to get the Video Course!](#)

## About Miguel

Welcome to my blog!

I'm a software engineer and technical writer, currently living in Drogheda, Ireland.







You can also find me on [Github](#), [LinkedIn](#), [Bluesky](#), [Mastodon](#), [Twitter](#), [YouTube](#), and [Patreon](#).



Thank you for visiting!

## Categories

- AI 3
- Arduino 7
- Authentication 10
- Blog 1
- C++ 5
- CSS 1
- Cloud 11
- Database 23
- Docker 5
- Filmmaking 6
- Flask 129
- Games 1
- IoT 8
- JavaScript 36
- MicroPython 10
- Microdot 1
- Microservices 2
- Movie Reviews 5
- Personal 3
- Photography 7
- Product Reviews 2
- Programming 193
- Project Management 1
- Python 174
- REST 7
- Raspberry Pi 8
- React 19
- Reviews 1

-  [Robotics](#) 6
-  [Security](#) 12
-  [Video](#) 22
-  [WebSocket](#) 2
-  [Webcast](#) 3
-  [Windows](#) 1