The Flask Mega-Tutorial Part XVII: Deployment on Linux (/post/the-flask-mega-tutorial-part-xvii-deployment-on-linux)

March 27 2018

Posted by Miguel Grinberg (/author/Miguel Grinberg) under Raspberry Pi (/category/Raspberry Pi), Cloud (/category/Cloud), Python (/category/Python), Flask (/category/Flask), Programming (/category/Programming).

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

For your reference, below is a list of the articles in this series.

- Chapter 1: Hello, World! (/post/the-flask-mega-tutorial-part-i-hello-world)
- Chapter 2: Templates (/post/the-flask-mega-tutorial-part-ii-templates)
- Chapter 3: Web Forms (/post/the-flask-mega-tutorial-part-iii-web-forms)
- Chapter 4: Database (/post/the-flask-mega-tutorial-part-iv-database)
- Chapter 5: User Logins (/post/the-flask-mega-tutorial-part-v-user-logins)
- Chapter 6: Profile Page and Avatars (/post/the-flask-mega-tutorial-part-vi-profile-page-and-avatars)
- Chapter 7: Error Handling (/post/the-flask-mega-tutorial-part-vii-error-handling)
- Chapter 8: Followers (/post/the-flask-mega-tutorial-part-viii-followers)
- Chapter 9: Pagination (/post/the-flask-mega-tutorial-part-ix-pagination)
- Chapter 10: Email Support (/post/the-flask-mega-tutorial-part-x-email-support)
- Chapter 11: Facelift (/post/the-flask-mega-tutorial-part-xi-facelift)
- Chapter 12: Dates and Times (/post/the-flask-mega-tutorial-part-xii-dates-and-times)
- Chapter 13: I18n and L10n (/post/the-flask-mega-tutorial-part-xiii-i18n-and-l10n)
- Chapter 14: Ajax (/post/the-flask-mega-tutorial-part-xiv-ajax)
- Chapter 15: A Better Application Structure (/post/the-flask-mega-tutorial-part-xv-a-better-application-structure)
- Chapter 16: Full-Text Search (/post/the-flask-mega-tutorial-part-xvi-full-text-search)
- Chapter 17: Deployment on Linux (/post/the-flask-mega-tutorial-part-xvii-deployment-on-linux) (this article)
- Chapter 18: Deployment on Heroku (/post/the-flask-mega-tutorial-part-xviii-deployment-on-heroku)
- Chapter 19: Deployment on Docker Containers (/post/the-flask-mega-tutorial-part-xix-deployment-on-docker-containers)
- Chapter 20: Some JavaScript Magic (/post/the-flask-mega-tutorial-part-xx-some-javascript-magic)

- Chapter 21: User Notifications (/post/the-flask-mega-tutorial-part-xxi-user-notifications)
- Chapter 22: Background Jobs (/post/the-flask-mega-tutorial-part-xxii-background-jobs)
- Chapter 23: Application Programming Interfaces (APIs) (/post/the-flask-mega-tutorial-part-xxiii-application-programming-interfaces-apis)

Note 1: If you are looking for the legacy version of this tutorial, it's here (/post/the-flask-mega-tutorial-part-i-hello-world-legacy).

Note 2: If you would like to support my work on this blog, or just don't have patience to wait for weekly articles, I am offering the complete version of this tutorial packaged as an ebook or a set of videos. For more information, visit courses.miguelgrinberg.com (https://courses.miguelgrinberg.com).

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 (https://github.com/miguelgrinberg/microblog/tree/v0.17), Zip (https://github.com/miguelgrinberg/microblog/archive/v0.17.zip), Diff (https://github.com/miguelgrinberg/microblog/compare/v0.16...v0.17).

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 (https://www.digitalocean.com/), Linode (https://www.linode.com/), or Amazon Lightsail (https://amazonlightsail.com/) 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 (https://www.vagrantup.com/) and VirtualBox (https://www.virtualbox.org/) 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 OS X and Microsoft Windows (though OS X is a hybrid open-source/commercial option as it is based on Darwin, an open-source BSD derivative).

Since OS X 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 little bit of money, you can get an account at Digital Ocean, Linode or Amazon Lightsail and create a Ubuntu 16.04 virtual server. You should use the smallest server option, which at the time I'm writing this, costs $5 per month for all three providers. 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 (https://www.vagrantup.com/) and VirtualBox (https://www.virtualbox.org/) 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/xenial64"
  config.vm.network "private_network", ip: "192.168.33.10"
  config.vm.provider "virtualbox" do |vb|
    vb.memory = "1024"
  end
end
```

This file configures a Ubuntu 16.04 server with 1GB of RAM, which you will be able to access from the host computer at IP address 192.168.33.10. To create the server, run the following command:

```
$ vagrant up
```

Consult the Vagrant command-line documentation (https://www.vagrantup.com/docs/cli/) to learn about other options to manage your virtual server.

Using a 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 (http://www.openssh.org/) already installed. If you are using Microsoft Windows, Cygwin (https://www.cygwin.com/), Git (https://git-scm.com/), and the Windows Subsystem for Linux (https://msdn.microsoft.com/en-us/commandline/wsl/about) provide OpenSSH, so you can install any of these options.

If you are using a virtual server from a third-party provider, when you created the server 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 `ubuntu`, without password automatically by Vagrant.

If you are 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 (http://en.wikipedia.org/wiki/Public-key_cryptography) 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 AAAAB3NzaC1yc2EAAAADAQABAAABAQCjw....F8Xv4f/0+7WT miguel@miguelspc
```

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 login 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 really 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 (https://wiki.ubuntu.com/UncomplicatedFirewall), 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 16.04 release, then you have a system that comes with full support for Python 3.5, 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 request 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, but at some point while you run the third install statement you will be prompted to choose a root password for the MySQL service, and you'll also be asked a couple of questions regarding the installation of the postfix package which you can 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 servers 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 (https://elastic.co) and follow their installation instructions to add it to your server. Note that the Elasticsearch package available in the Ubuntu 16.04 package repository is too old and will not work, you need version 6.x or newer.

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 (http://do.co/2FhdIes)
- Adding an SPF Record (http://do.co/2Ff8ksk)

- DKIM Installation and Configuration (http://do.co/2HW2oTD)

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 (http://ryanflorence.com/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 (/post/the-flask-mega-tutorial-part-xv-a-better-application-structure):

```
$ 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 two packages that are specific to this production deployment, so they are not included in the requirements file. The `gunicorn` package is a production web server for Python applications. The `pymysql` package contains the MySQL driver that enables SQLAlchemy to work with MySQL databases:

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

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/microblog
MS_TRANSLATOR_KEY=<your-translator-key-here>
```

This *.env* file is mostly similar to the example I shown in Chapter 15 (/post/the-flask-mega-tutorial-part-xv-a-better-application-structure), but I have used a random string for `SECRET_KEY`. To generate this random string I used 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, but this variable is needed before the *.env* file is parsed so it needs to be set manually. To avoid having to set it every time, I'm going to add it at the bottom of the *~/.profile* file for the `ubuntu` account, so that it is set automatically every time I log in:

```
$ echo "export FLASK_APP=microblog.py" >> ~/.profile
```

If you log out and back in, now `FLASK_APP` will be set for you. You can confirm that it is set 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:

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6
Server version: 5.7.19-0ubuntu0.16.04.1 (Ubuntu)

Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

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 type the MySQL root password that you selected when you installed MySQL to gain access to the MySQL command prompt.

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. This is going to be the password for the `microblog` database user, so it is a good idea to not use the same password you selected for the root user. The password for the `microblog` user 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 (http://gunicorn.org/), 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 it is very easy to use.

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, and the name after the colon is the name of this application.

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 (http://supervisord.org/) 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 4 microblog:app
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 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* sub-directory of the Microblog root directory.

To have a web site 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-enabled* directory. Nginx installs a test site in this location that I don't really need, so I'm going to start by removing it:

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

Below you can see the nginx configuration file for Microblog, which goes in */etc/nginx/sites-enabled/microblog*:

/etc/nginx/sites-enabled/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 application
        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 (https://nginx.org/en/docs/).

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

```
$ sudo service nginx reload
```

And now the application should be deployed. In your web browser, you can type the the IP address of your server (or 192.168.33.10 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 (https://letsencrypt.org/) SSL certificate. I have written a detailed article on my blog on how to Run your Flask application over HTTPS (https://blog.miguelgrinberg.com/post/running-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 version
(venv) $ sudo supervisorctl stop microblog      # stop the current server
(venv) $ flask db upgrade                       # upgrade the database
(venv) $ flask translate compile                # upgrade the translations
(venv) $ sudo supervisorctl start microblog     # start a new server
```

Raspberry Pi Hosting

The Raspberry Pi (http://www.raspberrypi.org/) 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 Raspbian (http://www.raspbian.org/), which is the official distribution from the Raspberry Pi Foundation.

To prepare the Raspberry Pi, I'm going to install a fresh Raspbian release. I will be using the September 2017 version of Raspbian Stretch Lite, but by the time you read this there is likely going to be newer versions out, so check the official downloads page (https://www.raspberrypi.org/downloads/raspbian/) to get the most current release.

The Raspbian 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 Raspbian image to an SD card from Windows, Mac OS X and Linux are available on the Raspberry Pi site (https://www.raspberrypi.org/documentation/installation/installing-images/).

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 set up. 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, Raspbian 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 of the 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 to requests from clients. You will probably want just one gunicorn worker. 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.