How to deploy a Python Flask application to the web with Google Cloud Platform - for free

Mar 30, 2019 18:00 · 1741 words · 9 minute read

Flask is a simple but powerful Python web framework. During development, a Flask application is available on `localhost:8080`. It gets served by Flask's development web server. **But how can you actually make your Flask application available from the web and let other people use your site?**

In this post, I will describe this process in detail. My post is primarily directed at ambitious hobby coders who want to show off their projects while keeping expenses low. I will be using a free virtual machine by Google Cloud Platform, Gunicorn as WSGI web server, and nginx as reverse proxy. I expect you to already have your Flask app in a git repository on GitHub.

Setting up the Virtual Machine on Google Cloud Platform

The Google Cloud Platform service offers a free `f1-micro` virtual machine that you can use to host your Flask application (see Free on GCP - Google Cloud). With 0.2 vCPUs and 0.6 GB RAM, it is not a very powerful machine, but sufficient for small projects.

Note: All shell commands should be executed on the VM, not on your local machine!

1. Create a `f1-micro` machine, using Ubuntu 18.04.2 LTS (Minimal Version) as operating system (OS).
2. Log in to the machine, e.g. via the Google Cloud Shell.
3. Generate an SSH key on the machine (more info):

- `ssh-keygen -t rsa -b 4096 -C "deploy-key-vm-1"`
- `eval "$(ssh-agent -s)"`

- `ssh-add ~/.ssh/id_rsa`

4. Add the SSH public key to this project as a deploy key on GitHub. So the VM has read-access to this git repository and can pull it.

- Display SSH public key with: `cat ~/.ssh/id_rsa.pub`
- Add it as a deploy key on `https://github.com/<repository>/settings/keys`

5. Clone the repository on the virtual machine:

- Install git with `sudo apt-get install git` (for the package install to work, you first need to update Ubuntu's package lists with `sudo apt-get update`)
- `git clone git@github.com:<username>/<repository>.git`

6. Install your project's dependencies. For that, you certainly need to install pip, which is by default not installed in Ubuntu's minimal version.

- `sudo apt-get install python3-pip`
- Check that the pip binary gets found with `pip3 --version`

Making the virtual machine available via a (sub-)domain

If you have a domain, I recommend to make the domain or one of its subdomains resolve to your virtual machine. Easiest is to set the (sub)domain's A-Record to `<VM_IP>`. As for domain providers, I use Gandi for important domains, and Domainssaubillig for cheap ones (e.g. 2,28€/year for a .de-domain). But beware, apparently Domainssaubillig is a one-man shop - I would not use it for business-critical things.

Gunicorn as web server

Flask recommends against using `flask run` (its development server) for production cases (see here). Therefore we use Gunicorn to run the Flask application for us (more info). Gunicorn is a web application server. It works with the Flask application code as both use the standardized Web Server Gateway Interface (WSGI).

1. Add Gunicorn to your project dependencies.

2. Start Gunicorn with `gunicorn src.app:app`. **Note:** You need to have your Python virtual environment activated or else the `gunicorn` command will not work. Also, `src.app:app` needs to correspond to your Flask main file path (`src.app`) & the start method's name (`app`).
3. The webserver is not reachable from outside the VM itself yet. To make the webserver also listen to the outside world, the command is: `gunicorn src.app:app –b 0.0.0.0:8000`.

Then, our app is reachable from `<domain>:8000` (if not, you need to change the firewall settings in the Google Cloud Platform VM settings and open port `8000` to the outside). That is already awesome! **But now, we want our app to be available on `<DOMAIN>`!** This means it should be running on port 80 (`<DOMAIN>:80`)!

nginx as reverse proxy

Having an app listen on port `80` requires special configuration. That is because ports below 1024 are protected by the operating system. We can use the webserver application nginx that *proxies* outside requests from port `80` to the internal port `8000`, where Gunicorn listens.

1. `sudo apt-get install nginx`. After it, a `Welcome to nginx!` page will appear when you open `<DOMAIN>` in your browser.
2. We then use the nginx configuration from the Gunicorn documentation and modify it a bit:

```
worker_processes 1;

user nobody nogroup;
error_log  /var/log/nginx/error.log warn;
pid /var/run/nginx.pid;

events {
  worker_connections 1024; # increase if you have lots of clients
  accept_mutex off; # set to 'on' if nginx worker_processes > 1
}

http {
  include mime.types;
  # fallback in case we can't determine a type
```

```
    default_type application/octet-stream;
    access_log /var/log/nginx/access.log combined;
    sendfile on;

    upstream app_server {
      # for a TCP configuration
      server 127.0.0.1:8000 fail_timeout=0;
    }

    server {
      # if no Host match, close the connection to prevent host spoofing
      listen 80 default_server;
      return 444;
    }

    server {
      listen 80;
      client_max_body_size 4G;

      # set the correct host(s) for your site
      server_name <DOMAIN>; # example.com www.example.com;

      keepalive_timeout 5;

      location / {
        try_files $uri @proxy_to_app;
      }

      location @proxy_to_app {
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header Host $http_host;
        # we don't want nginx trying to do something clever with
        # redirects, we set the Host: header above already.
        proxy_redirect off;
        proxy_pass http://app_server;
      }
    }
}
```

Note that you need to set `<DOMAIN>` to your own (sub)domain.

We overwrite the file `/etc/nginx/nginx.conf` with our `nginx.conf`. Restart nginx with `sudo systemctl restart nginx`, then view the status with `systemctl status nginx` to see if everything went fine.

Now, when we run `gunicorn src.app:app`, your domain (`<DOMAIN>`) will answer with the Flask webserver response.

Always-on web server

Setting up a background service

You might have noticed it - as soon as you close your command line window, the running gunicorn command gets terminated as well. The web server is not running anymore, and therefore not reachable anymore on `<DOMAIN>`. Does this mean that you have to keep your personal computer and the command line open all the time? Of course not, you can keep the process running in background. One way is to set up a service in Ubuntu via `systemd` ("System and Service Manager").

We create a `gunicorn.service` file in `/etc/systemd/system` (more info):

```
[Unit]
Description=gunicorn daemon
After=network.target

[Service]
PIDFile=/run/gunicorn/pid
User=<user>
Group=<user>
RuntimeDirectory=gunicorn
WorkingDirectory=/home/<user>/<repository>
ExecStart=/home/<user>/.local/bin/pipenv run gunicorn src.app:app --pid /ru
ExecReload=/bin/kill -s HUP $MAINPID
ExecStop=/bin/kill -s TERM $MAINPID
PrivateTmp=true

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

Note that you need to replace the `<...>` placeholders with your respective configuration. Also, you might need to change the `ExecStart` command depending on how

you installed your project dependencies. In my command, I assume that Pipenv was used. Pipenv is an awesome tool that makes the whole workflow around installing Python pip packages, dependency management, and virtual environments a breeze. I highly recommend it!

Then, add the file `gunicorn.conf` to `/etc/tmpfiles.d/gunicorn.conf`:

```
1 d /run/gunicorn 0755 <user> <user> -
```

Start the service with `sudo systemctl start gunicorn.service` (`systemctl` is part of systemd). As you can see in `systemctl status gunicorn.service`, the service is now active and your domain should again show your web application. The service stays active when you close the command line, and will stay active as long as the virtual machine is running. Also note that because of the `--reload` command flag in `ExecStart`, Gunicorn automatically reloads the server if the `app.py` file contents change. Without that, we would have to do `sudo systemctl restart gunicorn.service` after every file change.

Automatically start the web server on reboot

Right now, the newly created Gunicorn web server service stays active as long as the virtual machine is running. But when the virtual machine gets rebooted, it will not automatically start. In some circumstances, a reboot can happen without you triggering it. By default, GCP can automatically restart your instance, e.g. in case of maintenance events, hardware or software failures (see “Management, security, disks, networking, sole tenancy” settings when creating a VM).

We want our Gunicorn service to automatically start on boot. For that, we need to *enable* the service. As we can see in `systemctl status gunicorn.service`, the service is currently *disabled*. Disabled means, it won’t automatically start on boot. We can enable the service with `sudo systemctl enable gunicorn.service`. A bit of background info: nginx is also running as a service (the service was automatically installed when we installed it). Its service also needs to be enabled for our app to be reachable. Luckily, the service is enabled by default.

To make sure nginx and Gunicorn get automatically started on boot, reboot the VM with `sudo reboot now -h`. Then, check the status of the respective services (all should be *active*). Finally, check that the webserver is accessible by visiting `<DOMAIN>`.

Securing your website with HTTPS (SSL/TLS certificate)

Before showing off your website to a larger audience, you should seriously consider adding HTTPS by obtaining a SSL certificate. As a starting point, I recommend looking into Let's Encrypt and their Certbot client. With that, you can easily obtain and configure free SSL certificates. If you followed the instructions in this post, this official tutorial is the way to go.

I will outline in a separate post why HTTPS is useful (**UPDATE 2019-07-06**: "Why HTTPS is important & how to make your website support it"), ~~and how to automate the certificate renewal process, something that I find not very well explained in the official documentation.~~

Update 2019-07-06

When configuring Certbot according to the official tutorial, **Certbot will automatically set up automatic renewal**. While this is a very neat feature and saves us some work, this was not well-documented until recently (at least not on the official tutorial page). The in-detail user-guide as well as some GitHub issues (#5398, #5034) give more context on this. If your Certbot did not come with automatic renewal, it will message you a couple of weeks before the SSL certificate expires. Then, you should set up a cronjob manually. This is the code I use:

```
# m h  dom mon dow   command
# Every Tuesday 5:00, renew certificate.
# Useful service for writing cron schedule expressions: https://crontab.guru
0 5 * * 2 /usr/bin/certbot --quiet renew
```

Wrap-up

In this post, I have shown how to host a Python Flask web application for free on a Google Cloud Platform virtual machine, using GitHub as code repository platform, Gunicorn as web server, and nginx as reverse proxy server. In the end, whenever we push new stuff in our project to GitHub, we just have to do a `git pull` on the VM and the website gets updated! 😊🎉

I hope you enjoyed this post and could make use of it. In case you have questions or remarks, just send me an e-mail to mail@muellermarkus.com. I am looking forward to it!

Read more

Presenting Socket.IO: Building a chat in 70 lines May 30 2020

A short historic overview: Building JavaScript apps that receive server events in real-time May 25 2020

My Favorite Shortcuts for JetBrains' IDEs Apr 11 2020

Exploring Observable's interactive JavaScript Notebooks Apr 5 2020

Why HTTPS is important & how to make your website support it Jul 6 2019

7 Steps to Better JavaScript Development Mar 27 2018

Coding Interview Tips Mar 10 2018

Userscripts: Nuclino Sidebar Toggle Shortcut Feb 9 2018