teclado

Deploying Python Apps with uWSGI and nginx

This tutorial covers the basic steps of deploying a Python application onto a public server using MongoDB, uWSGI, and nginx. We will be using a sample project, a Flask web server, for demonstration, however, the deployment process should remain similar for any other Python applications. Our sample project can be found here: https://github.com/tecladocode/price-of-chair-deployment.

In this tutorial, we will not cover how to set up a server on any hosting platforms, however, if you are looking for such a tutorial, you may take a look at this one: DigitalOcean Tutorial, in which you will learn the basics on setting up a server from the beginning on a cloud hosting platform. The procedure should be similar for setting up server on other platforms as well, such as AWS (Amazon Web Service).

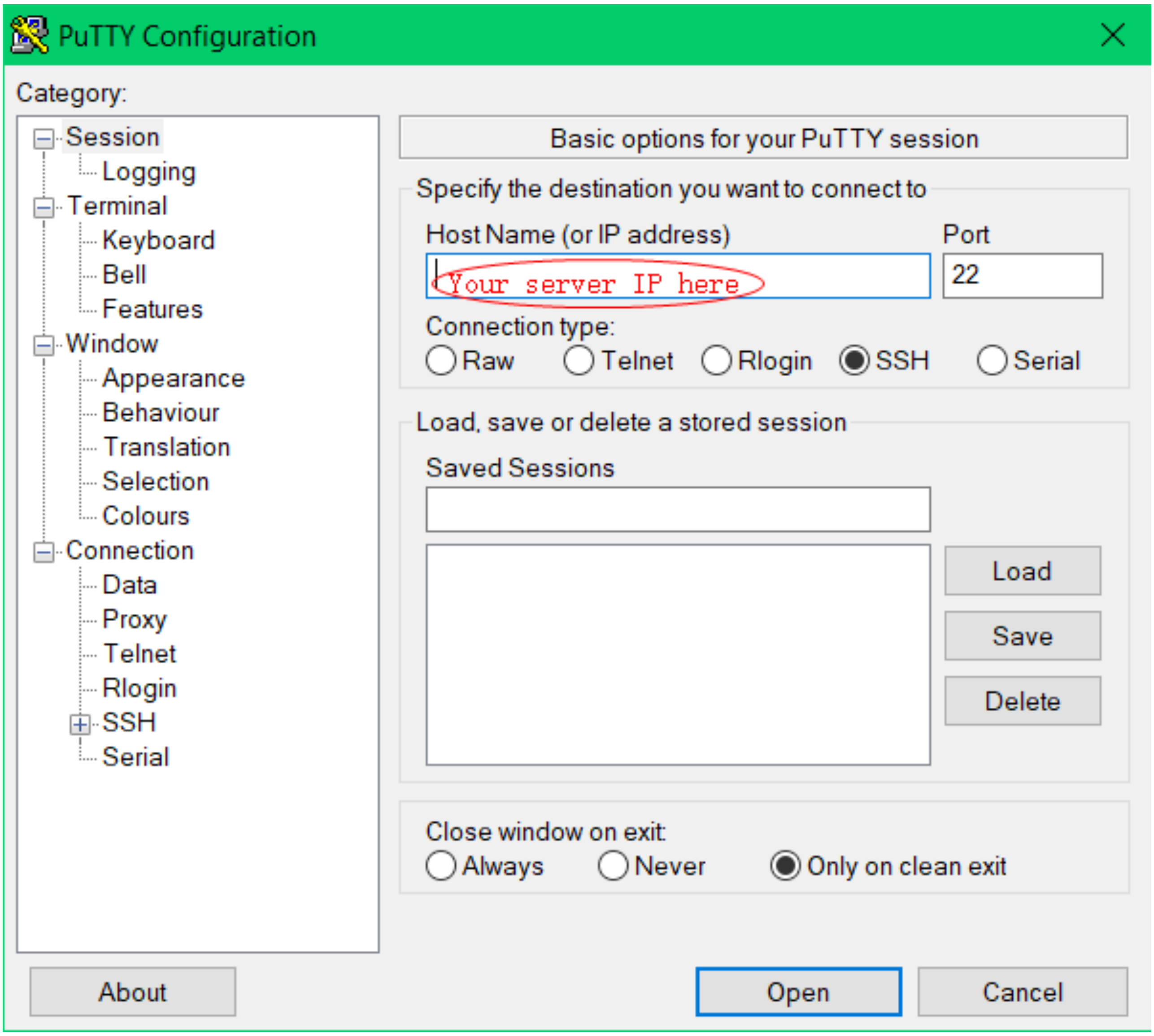
If you are a first time learner, we highly recommend you to follow through the whole tutorial so that you can get familiar with it and may be less likely to run into problems.

Connecting to our server

In order to connect to our server, we need to use a tool called `SSH` (Secure Shell). We can SSH our server using the command:

```
ssh root@<your server ip>
```

You will be asked for the root password (or the SSH key if you have set it up previously). Beware that `SSH` command only works on `UNIX` , not on `Windows` . However, there are plenty of software that you can use to SSH from Windows, PuTTy is a popular choice:



Host Name (or IP address): Your server IP here

Port: 22

Connection type: ○ Raw ○ Telnet ○ Rlogin ● SSH ○ Serial

Close window on exit: ○ Always ○ Never ● Only on clean exit

After connecting to our server and logging in as the `root` user, it is recommended to run the below command first to get all the available updates:

```
apt-get update
```

We can use the following command to install packages:

```
apt-get install <package1> <package2>
```

Note that this is a just an example to install different packages using one command, we will see real use cases in the following sections.

Creating another user

Since the `root` user is the most powerful, essentially a root user can do everything on the server, so we may want to limit access to it to improve security. So in this section, we will create a new user and configure it to "act like" a `root` user but with certain limitations, and we will login as this user from then on. It is highly recommended to do so, but if you choose not to follow this practice and simply want to login as the `root` user anyway, you may click here to skip to the next section.

Hello John Doe

In this section, we will create a user named `johndoe`. You may choose any name you want, just remember to swap `johndoe` with your username for each command and configuration. We can create a new user `johndoe` with the following command:

```
adduser johndoe
```

You will be asked to enter and confirm the password for this user, and then provide some info about this user. Notice that you can leave the info sections blank if you want to. And if you entered unmatching passwords, just complete the info section and we can change the password later by using the command:

```
passwd johndoe
```

Providing user with additional privilege

Since we will be logging in as `johndoe` for most of the time in the future, we will want it to have some "extra power", that is, temporarily acting as a super user. To do this, we need to run the command:

```
visudo
```

first, and we will see a text file popping up. Then we navigate to the lines containing:

```
# User privilege specification
root ALL=(ALL:ALL) ALL
```

You can do this with the arrow keys. We need to add a new line for our user in this section:

```
# User privilege specification
root ALL=(ALL:ALL) ALL
johndoe ALL=(ALL:ALL) ALL
```

Remember that the `ALL` has to be **all uppercase**, otherwise it will raise syntax error.

After adding this line, use `ctrl + o` to save and press `ENTER` to overwrite, then press `ctrl + x` to quit.

Enable SSH for our new user

Next, we want to allow us to login as `johndoe` using SSH, and we may also want to disable login as `root` from SSH to make our server more secure. To do this, use the command:

```
vi /etc/ssh/sshd_config
```

And we will be prompted with another text file. Navigate to the section which contains:

```
# Authentication
PermitRootLogin yes
```

Press `i` on your keyboard to enter insert mode and change the `yes` to `no` to disallow login as root.

Next, look for a configuration called `PasswordAuthentication` :

```
# Change to no to disable tunnelled clear text passwords
PasswordAuthentication yes
```

Important: make sure to set it as `yes` so that you can use your password to login in the future. It should be set to `yes` already, however, there are some platforms that enforces SSH key authentication and set it to `no` instead.

Then go to the bottom of the file and add the following lines:

```
AllowUsers johndoe
```

For this section, if you already have other users on the server, make sure to include them as well. On AWS, for instance, a user named `ubuntu` is initialized and used to login for the first time. If you choose to create a new user, say `johndoe`, then you will need to add them together into `AllowUsers`:

```
AllowUsers johndoe ubuntu
```

Otherwise, you will no longer be able to login as `ubuntu` in the future.

Next, press `Esc` to quit insert mode, press `:` (colon) to enable the command function and enter `wq` to write and quit (after hitting `ENTER` to confirm).

Some other useful vi commands are: `:q` to quit without modification and `:q!` to force quit and discard changes.

Finally, we use the command:

```
service sshd reload
```

to enable our modifications.

Now we've created a new user `johndoe` and enabled both its super user privilege and SSH access.

MongoDB

Installing MongoDB

```
sudo apt-get install -y mongodb
```

Checking MongoDB is running

To check whether MongoDB is running, type:

```
sudo systemctl status mongodb
```

If it's not running, you can start it by typing:

```
sudo systemctl start mongodb
```

And you can always stop it with:

```
sudo systemctl stop mongodb
```

MongoDB will start when your server restarts. You can prevent that default behaviour by doing:

```
sudo systemctl disable mongodb
```

Getting code from GitHub

In this section, we will pull our code from `GitHub` , which integrates a popular VCS (Version Control System) called `Git` . `Git` is a very good tool to manage and access your code both locally and remotely.

Setting up our app folder

First, we create a folder called `pricing-service` for our app, since our sample project is a REST API which manages items of stores. We create this folder using the following command:

```
sudo mkdir -p /var/www/html/pricing-service
```

The folder is owned by the `root` user since we used `sudo` to create it. We need to transfer ownership to our current user:

```
sudo chown johndoe:johndoe /var/www/html/pricing-service
```

Remember that `johndoe` is the username in our tutorial, make sure you change it to yours accordingly. The same goes for `pricing-service` .

Next, we get our app from `Git` :

```
cd /var/www/html/pricing-service/
git clone https://github.com/tecladocode/price-of-chair-deployment .
```

Note that there's a trailing space and period (`.`) at the end, which tells `Git` the destination is the current folder. If you're not in this folder `/var/www/html/pricing-service/`, remember to switch to it or explicitly specify it in the `Git` command. And for the following commands in this section, we all assume that we are inside the folder `/var/www/html/pricing-service/` unless specified otherwise.

In order to store logs, we need to create a log folder, (under `/var/www/html/pricing-service/`). We also need to create a file called `emperor.log`, where the uWSGI logs will go:

```
mkdir log
touch log/emperor.log
```

Then we will install Python:

```
sudo apt install software-properties-common
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt install build-essential python3.7-dev python3-pip python3.7
```

Next, we will install `pipenv`, which is a python library used to create virtual environment. Since we may want to deploy several services on one server in the future, using Pipenv allows us to create independent environment for each project so that their dependencies won't affect each other. We may install `pipenv` using the following command:

```
pip3 install --user pipenv
echo "PATH=$HOME/.local/bin:$PATH" >> ~/.bashrc
source ~/.bashrc
```

After it is installed, we can use it to create our environment and install the requirements:

```
pipenv install
```

uWSGI

In this section, we will be using `uWSGI` to run the app for us, in this way, we can run our app in multiple threads within multiple processes. It also allow us to log more easily. More details on `uWSGI` can be found here.

First, we define a `uWSGI` service in the system by:

```
sudo vi /etc/systemd/system/uwsgi_pricing_service.service
```

And the content we are going to input is shown below:

```
[Unit]
Description=uWSGI Pricing Service

[Service]
User=johndoe
Group=johndoe
WorkingDirectory=/var/www/html/pricing-service
Environment=MONGODB_URI=mongodb://127.0.0.1:27017/fullstack
ExecStart=/home/johndoe/.local/bin/pipenv run uwsgi --master --emperor /var/
www/html/pricing-service/uwsgi.ini --die-on-term --uid johndoe --gid johndoe
 --logto /var/www/html/pricing-service/log/emperor.log
Restart=always
KillSignal=SIGQUIT
Type=notify
NotifyAccess=all

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

We will explain the basic idea of these configs. Each pair of square brackets `[]` defines a `section` which can contain some properties.

The `Unit` section simply provides some basic description and can be helpful when looking at the logs.

The `Service` section contains several properties related to our app.
The `Environment` properties defines all the environment variables we need in our code. In our sample code, we want to retrieve the `MONGODB_URI` from system environment. And this is the place where you should keep all your secrets, such as secret keys and credentials.

If we want to add multiple environment variables, we just need to add multiple lines of the `Environment` entry following the syntax:

```
Environment=key=value
Environment=key=value
Environment=key=value
...
```

The `ExecStart` property informs `uWSGI` on how to run our app as well as log it.

At last, the `WantedBy` property in `Install` section allows the service to run as soon as the server boots up.

Hint: after editing the above file, press `ESC` to quit insert mode and use `:wq` to write and quit.

Configuring uWSGI

Before creating or editing the configuration file, run this command and remember the path it tells you. You'll need it to tell uwsgi where the Python home is:

```
pipenv --venv
```

Our next step is to configure `uWSGI` to run our app. To do so, we need to create a file named `uwsgi.ini` with the following content:

```
[uwsgi]
base = /var/www/html/pricing-service
app = app
module = %(app)

home = /path/to/your/venv
pythonpath = %(base)

socket = %(base)/socket.sock

chmod-socket = 777

processes = 8

threads = 8

harakiri = 15

callable = app

logto = %(base)/log/%n.log
```

Note that you should change the `base` folder accordingly in your own app. For the second entry, `run` is referred to the `run.py` in our sample app, which serves as the entry point of our app, so you may need to change it accordingly in your own project as well. We defined the `socket.sock` file here which will be required by the `nginx` later. The socket file will serve as the connection point between `nginx` and our `uWSGI` service.

We asked for 8 processes with 8 threads each, but you may adjust them according to your server capacity and data volume. The `harakiri` is a Japanese word for suicide, so in here it means after how long (in seconds) will the `emperor` kill the thread if it has failed. This is also an advantage we have with `uWSGI` , it allows our service to be resilient to minor failures. And it also specifies the log location.

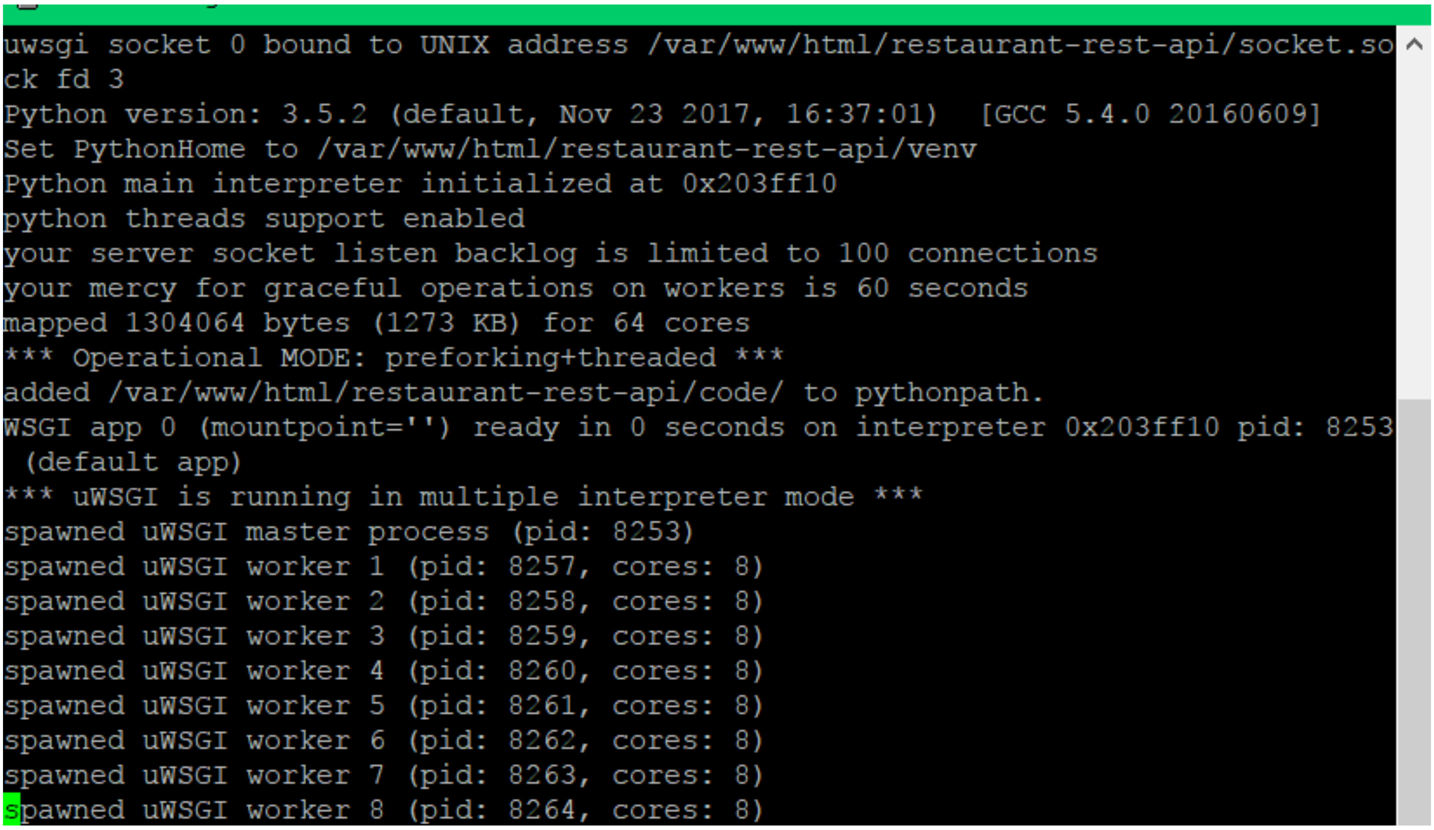
And at last, after saving the above file, we use the command below to run the `uWSGI` service we defined earlier:

```
sudo systemctl start uwsgi_pricing_service
```

And we should be able to check the `uWSGI` logs immediately to make sure it's running by using the command:

```
vi /log/uwsgi.log
```

If anything is running normally, we should be seeing something like this:



```
uwsgi socket 0 bound to UNIX address /var/www/html/restaurant-rest-api/socket.so
ck fd 3
Python version: 3.5.2 (default, Nov 23 2017, 16:37:01)  [GCC 5.4.0 20160609]
Set PythonHome to /var/www/html/restaurant-rest-api/venv
Python main interpreter initialized at 0x203ff10
python threads support enabled
your server socket listen backlog is limited to 100 connections
your mercy for graceful operations on workers is 60 seconds
mapped 1304064 bytes (1273 KB) for 64 cores
*** Operational MODE: preforking+threaded ***
added /var/www/html/restaurant-rest-api/code/ to pythonpath.
WSGI app 0 (mountpoint='') ready in 0 seconds on interpreter 0x203ff10 pid: 8253
 (default app)
*** uWSGI is running in multiple interpreter mode ***
spawned uWSGI master process (pid: 8253)
spawned uWSGI worker 1 (pid: 8257, cores: 8)
spawned uWSGI worker 2 (pid: 8258, cores: 8)
spawned uWSGI worker 3 (pid: 8259, cores: 8)
spawned uWSGI worker 4 (pid: 8260, cores: 8)
spawned uWSGI worker 5 (pid: 8261, cores: 8)
spawned uWSGI worker 6 (pid: 8262, cores: 8)
spawned uWSGI worker 7 (pid: 8263, cores: 8)
spawned uWSGI worker 8 (pid: 8264, cores: 8)
```

But if there are any error sin our code, it will also be reflected in the log.

Nginx

`Nginx` (engine x) is an HTTP and reverse proxy server, a mail proxy server, and a generic TCP/UDP proxy server. In this tutorial, we use `nginx` to direct traffic to our application. `Nginx` can be really helpful in scenarios like running our app on multiple threads, and it performs very well so we don't need to worry about it slowing down our app. More details about `nginx` can be found here.

Installing nginx

```
sudo apt-get install nginx
```

Configure firewall to grant access to nginx

First, check if the firewall is active:

```
sudo ufw status
```

If not, we will enable it later. Before that, let's add some new rules:

```
sudo ufw allow 'Nginx HTTP'
sudo ufw allow ssh
```

Important: the second line, adding SSH rules, is not related to `nginx` configuration, but since we're activating the firewall, we don't want to get blocked out of the server!

If the UFW (Ubuntu Firewall) is inactive, use the command below to activate it:

```
sudo ufw enable
```

To check if `nginx` is running, use the command:

```
systemctl status nginx
```

Some other helpful command options for system controller are:

```
systemctl start <service_name>
systemctl restart <service_name>
systemctl reload <service_name>
systemctl stop <service_name>
```

Configure nginx for our app

Before deploying our app onto the server, we need to configure `nginx` for our app. Use the below command to create a config file for our app:

```
sudo vi /etc/nginx/sites-available/pricing-service.conf
```

Note that `pricing-service` is what we named our service, you may change it accordingly, but remember to remain consistent throughout the configurations.

Next, we input the below text into `pricing-service.conf` file. **Remember to change your service name accordingly in this file as well**.

```
server {
  listen 80;
  real_ip_header X-Forwarded-For;
  set_real_ip_from 127.0.0.1;
  server_name localhost;

  location / {
    include uwsgi_params;
    uwsgi_pass unix:/var/www/html/pricing-service/socket.sock;
  }

  error_page 404 /404.html;
  location = /404.html {
    root /usr/share/nginx/html;
  }

  error_page 500 502 503 504 /50x.html;
  location = /50x.html {
    root /usr/share/nginx/html;
  }
}
```

The above config allows `nginx` to send the request coming from our user's browser to our app. It also sets up some error pages for our service using `nginx` predefined pages.

And at last, in order to enable our configuration, we need to do something like this:

```
sudo rm /etc/nginx/sites-enabled/default
sudo ln -s /etc/nginx/sites-available/pricing-service.conf /etc/nginx/sites-enabled/
```

We need to remove the default file since `nginx` will look at this file by default. We want `nginx` to look at our config file instead, thus we added a soft link between our config file and the `sites-enabled` folder.

Running our app

Finally, we can launch our app! We can do so by starting the `nginx` and `uWSGI` services we defined (we already started the `uWSGI` service in the previous section).

```
sudo systemctl start nginx
```

If any of these services is already running, you may use the below commands (taking `nginx` for example) to reload and restart it so that it has the latest changes:

```
sudo systemctl reload nginx
sudo systemctl restart nginx
```

Creating a cron job

In this section we'll create a cron job that runs hourly.

To do this, modify the `crontab` file:

```
sudo vi /etc/crontab
```

And in it, place the script you want to run every hour. We will be running our `alert_updater.py` file from our code. Do not delete any existing lines, so we have just added the one for our alert updater.

```
# /etc/crontab: system-wide crontab
# Unlike any other crontab you don't have to run the `crontab'
# command to install the new version when you edit this file
# and files in /etc/cron.d. These files also have username fields,
# that none of the other crontabs do.

SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# m h dom mon dow user  command
11 *    * * *   johndoe    cd /var/www/html/pricing-service && /home/johndoe
/.local/bin/pipenv run python alert_updater.py
17 *    * * *   root    cd / && run-parts --report /etc/cron.hourly
25 6    * * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --r
eport /etc/cron.daily )
47 6    * * 7   root    test -x /usr/sbin/anacron || ( cd / && run-parts --r
eport /etc/cron.weekly )
52 6    1 * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --r
eport /etc/cron.monthly )
```

Deployment wrap-up

As the tutorial is very detailed, you may find it a bit hard to put the pieces together. Here's a quick wrap-up that may help you sort things out.

- We created a `UNIX` user and granted him some privilege.
- We set up a MongoDB database and connected our app to it.
- We used `uWSGI` to run our app multi-processly and multi-threadly.
- We used `nginx` to direct requests to our `uWSGI` service.
- We added a cron job that runs every hour to update our alerts and send e-mails if necessary.

Thanks for reading!