Deploying Python Flask: the example of IEO CTD Checker

In this guide, we will be setting up IEO CTD Checker, a Python application using the Flask micro-framework on Ubuntu 16.04. The bulk of this article will be about how to set up the Gunicorn application server to launch the application and Nginx to act as a front-end reverse proxy. The systemd will allow Ubuntu's init system to automatically start everything whenever the server boots.

Prerequisites

Before starting on this guide, you should have a non-root user configured on your server. This user needs to have `sudo` privileges so that it can perform administrative functions. To learn how to set this up, follow this initial server setup guide.

Install the components

Install Conda

Miniconda is a free minimal installer for conda. It is a small, bootstrap version of Anaconda that includes only conda, Python, the packages they depend on, and a small number of other useful packages, including pip, zlib and a few others.

In our case, Miniconda Linux 64-bits was selected (Python version 3.7). To know more about your operating system type in your command window `uname -a`

Install nginx

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

> NOTE: In an initial stage of this project, the preinstalled version of Python was used in combination with pip and virtualenv. However, a problem with the Python Cartopy library was found. By installing conda, all problems dissapeared.

Create a virtual environment

With conda, we can create, export, list, remove, and update environments that have different versions of Python and/or packages installed in them. In our case, we use it to isolate our Flask application from the other Python files on the system.

To create an environment with a specific version of Python:

```
conda create --name env_ctdcheck python=3.7
```

Here, `env_ctdcheck` is the name chosen for our virtual environment. When conda asks you to proceed, type `y`. This creates the env_ctdcheck environment in `/envs/`.

Verify that the new environment was installed correctly:

```
conda info --envs
```

To activate the environment:

```
conda activate env_ctdcheck
```

Your prompt will change to indicate that you are now operating within the virtual environment. It will look something like this `(env_ctdcheck) user@host:~/ctdcheck$ .`

To deactivate the environment:

```
conda deactivate env_ctdcheck
```

If you want to eliminate the environment:

```
conda env remove --name env_ctdcheck
```

Install all the components after activate the environment:

```
conda install flask pandas_flavor matplotlib cartopy lxml pillow fpdf
```

To save all dependencies to a file text:

```
conda env export > environment.yml
```

> Install all the programs that you want in this environment at the same time to avoid conflicts.

In case you want create the environment from the `environment.yml` (for example, to recreate in other computer):

```
conda env create -f environment.yml
```

After that, activate the new environment (the first line of the `yml` file sets the new environment's name) and verify that the new environment was installed correctly.

Create a simple Python Flask app

It's time to create a simply Python Flask application that we'll call `app.py` : and we'll locate the file inside a directory call `ctdcheck`

```
(env_ctdcheck) $ vi ~/ctdcheck/app.py
```

Within this file, we'll place our application code. Basically, we need to import flask and instantiate a Flask object. We can use this to define the functions that should be run when a specific route is requested:

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def hello():
        return "<h1 style='color:blue'>Hello There!</h1>"

if __name__ == "__main__":        app.run(host='0.0.0.0')
```

We test the app by typing:

```
(env_ctdcheck) $ python app.py
```

The app should be accesible through the browser: `http://localhost:5000` or `http://127.0.0.1:5000` You should see something like this:

When you are finished, hit CTRL-C in your terminal window a few times to stop the Flask development server.

> If the port is not accesible try to open up it: `(env_ctdcheck) $ sudo ufw allow 5000`

> New versions of Flask may run in a different way. Check:
>
> https://flask.palletsprojects.com/en/1.1.x/quickstart/#a-minimal-application

Create the WSGI entry point

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

We will call the file `wsgi.py`:

`(env_ctdcheck) $ vi ~/ctdcheck/wsgi.py`

The file is incredibly simple, we can simply import the Flask instance from our application and then run it:

```
from myproject import app
if __name__ == "__main__":    app.run()
```

Save and close the file when you are finished.

Testing Gunicorn

Before moving on, we should check that Gunicorn is up and running correctly.

We can do this by simply passing it the name of our entry point. This is constructed by the name of the module (minus the `.py` extension, as usual) plus the name of the callable within the application. In our case, this would be `wsgi:app`.

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

```
(env_ctdcheck) $ cd ~/ctdcheck

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

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

```
http:// server_domain_or_IP
:5000
```

You should see your application's output again. We're now done with our virtual environment, so we can deactivate it: `conda deactivate env_ctdcheck` Any Python commands will now use the system's Python environment again.

Create a systemd unit file

The next piece we need to take care of is the systemd service unit file. Creating a systemd unit file will allow Ubuntu's init system to automatically start Gunicorn and serve our Flask application whenever the server boots. To begin, create a unit file ending in .service within the /etc/systemd/system directory:

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

The file content:

```
[Unit]
Description=uWSGI instance to serve myproject
After=network.target
[Service]
User=myuser
Group=www-dataWorkingDirectory=/home/myuser/ctdcheck
Environment="PATH=/home/myuser/miniconda3/envs/env_ctdcheck/bin"
ExecStart=/home/myuser/miniconda3/envs/env_ctdcheck/bin/gunicorn --timeout 300 --bind
unix:myproject.sock -m 007 wsgi:app
[Install]WantedBy=multi-user.target
```

We start with the `[Unit]` section, which is used to specify metadata and dependencies. We'll put

a description of our service here and tell the init system to only start this after the networking target has been reached.

Next, we'll open up the `[Service]` section. We'll specify the user and group that we want the process to run under. We will give our regular user account ownership of the process since it owns all of the relevant files. We'll give group ownership to the `www-data` group so that Nginx can communicate easily with the Gunicorn processes. We'll then map out the working directory so that the init system knows where our the executables for the process are located (within our virtual environment). We'll then specify the commanded to start the service. Systemd requires that we give the full path to the Gunicorn executable, which is installed within our virtual environment. We will tell it to start 3 worker processes (adjust this as necessary). We will also tell it to create and bind to a Unix socket file within our project directory called `myproject.sock`. We'll set a umask value of `007` so that the socket file is created giving access to the owner and group, while restricting other access. Finally, we need to pass in the WSGI entry point file name and the Python callable within. In our specific case, a timeout of 300s was set-up to allow upload of multiple files.

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

With that, our systemd service file is complete. Save and close it now. We can now start the Gunicorn service we created and enable it so that it starts at boot:

```
sudo systemctl start myprojectsudo systemctl enable myproject
```

If we want to stop the service just type:

```
sudo systemctl stop myproject
```

Configuring Nginx to Proxy Requests

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

Begin by creating a new server block configuration file in Nginx's `sites-available` directory. We'll

simply call this `ctdcheck` to keep in line with the rest of the guide:
`sudo vi /etc/nginx/sites-available/` **ctdcheck**

Open up a server block and tell Nginx to listen on the default port 80. We also need to tell it to use this block for requests for our server's domain name or IP address:

/etc/nginx/sites-available/ctdcheck

```
server {
    listen 80;
    listen [::]:80;

    ctdcheck.ieo.es www.ctdcheck.ieo.es;
}
```

The only other thing that we need to add is a location block that matches every request. Within this block, we'll include the `proxy_params` file that specifies some general proxying parameters that need to be set. We'll then pass the requests to the socket we defined using the `proxy_pass` directive:

/etc/nginx/sites-available/ctdcheck

```
server {
    listen 80;
    listen [::]:80;
    server_name ctdcheck.ieo.es www.ctdcheck.ieo.es;
    location / {
        include proxy_params;        proxy_pass
http://unix:/home/myuser/ctdcheck/myproject.sock;
    }
}
```

That's actually all we need to serve our application. Save and close the file when you're finished. To enable the Nginx server block configuration we've just created, link the file to the `sites-enabled` directory:

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

With the file in that directory, we can test for syntax errors by typing: `sudo nginx -t`

If this returns without indicating any issues, we can restart the Nginx process to read the our new config: `sudo systemctl restart nginx`

The last thing we need to do is adjust our firewall again. We no longer need access through port 5000, so we can remove that rule. We can then allow access to the Nginx server:

```
sudo ufw delete allow 5000sudo ufw allow 'Nginx Full'
```

You should now be able to go to your server's domain name or IP address in your web browser. In this case, by typing: `http://ctdcheck.ieo.es`

You should see your application's output:

Create a more compex Python Flask app: the real case of IEO CTD Checker

In this final section, we show technical issues about a specific and more complex app.

It's time to upload to the server our real code and test the app. The code of IEO CTD Checker can be found here: https://github.com/PabloOtero/CTDChecker

This app needs an updated file containing the list of Cruise Summary. This file is mantained by our colleagues of BSH: `http://seadata.bsh.de/isoCodelists/sdnCodelists/csrCodeList.xml`

The easiest way to download the file everyday at midnight and detele the previous file is create a task in the crontab:

`crontab -e`

And type inside:

```
0 0 * * * cd /home/myuser/ctdcheck && wget -N
```

http://seadata.bsh.de/isoCodelists/sdnCodelists/csrCodeList.xml

---

Revision #11

Created Fri, Feb 14, 2020 12:17 PM by Pablo

Updated Thu, Sep 17, 2020 6:47 AM by Pablo