Automating Flask applications deployments using Ansible, Gunicorn and Nginx

In this article we will explore the purpose, the tools and the steps necessary to automate the deployment of flask web applications using Ansible:

Content

Why it might be a good idea to automate your deployment processes

The value of deployment automation may vary depending on the size, complexity or volume of your processess but here are some key bennefits that automated deplyments can bring whether you are a regular user or a big business or organization:

- Faster deployments: once you have your automated deploying process set up, it can be performed in seconds. This might be quite handy either if you are making changes to your application, deploying to a new server or restoring your network services after a crash.
- Minimizing errors: deployments involve multiple steps, when doing it manually essential steps can be missed and small mistakes might be overlooked. Once your

deployment automation has been correctly configured the proccess is set and you can rely on it to aways work the same way
- Scalability: If you are working with a single application, without continuous update releases and only one server, deploying it manually doesn't seen to be a problem. Scale this up to more than three applications with diferent dependencies or multiply the number of servers you are deploying to and it can get really messy very fast. Automated deployments allows you to configure and manage multiple servers and applications in an considerably more controlled manner, making it not only less repetitive and time consuming but less prone to human errors as well.

An overview on Ansible

Ansible is an open source automation tool witten in Python. It can configure systems, deploy software, and orchestrate advanced workflows to support application deployment, system updates, and more. The instructions to the assignments that should be executed are passed through a series of written tasks called playbooks. Ansible Playbooks offer a repeatable, re-usable, simple configuration management and multi-machine deployment system, one that is well suited to deploying from simple applications to very complex system environments.

What this article will not cover

The main goal of this article is to give you a basic understanding on how the deployment of a flask application can be made in an automated way using Ansible. We expect you to have a basic understanding on how web applications work and we will not provide instructions on how to buy and configure a domain nor on how to connect or set up a server. For the purposes of this tutorial we assume you have a domain name with the proper DNS/CNAME configuration in place, which associates your domain name with the IP address of the server that you'll be deploying this application to.

Getting started

We will start by creating and moving into a folder where we will set up a project locally.

```
$ mkdir myproject
$ cd myproject
```

Creating a requirements file

the next step is to create a requirements.txt file where we will include all the needed dependencies that should be installed to our virtual enviroment

```
touch requirements.txt
```

In this case we included only Flask, which we will be using to develop our application, and gunicorn, that will be responsible to manage the comunication between the application and the web services, but it might include a lot more modules and dependencies depending on the complexity of your application:

```
Flask
gunicorn
```

Setting up a virtual enviroment

now let's start a virtual enviroment where we will install the dependecies necessary to run our Flask application. Type:

```
python3 -m venv myprojectenv
```

to create the virtual enviroment, and then

```
source myprojectenv/bin/activate
```

to activate it

Finally we will install all the needed dependencies within our virtual enviroment by executing:

```
$ pip install -r requirements.txt
```

Creating a flask application

After that we will create a very simple Flask aplication inside our project folder

```
$ touch myproject.py
```

```
from flask import Flask,

app = Flask(__name__)

@app.route("/")
def index():
  return "<h1>Hello World!</h1>"
```

At this point, all the files created should be committed to a git repo. Here's a link with more information on [how to create a github repo and how to add/commit files to it](). From now on we assume this is done and that the github link to the repo is - whenever you see this, replace by your git URL

Once we have our application set up locally it is time to start our deployment process. As this article's main purpose is to show how to automate the procces using Ansible, we will not be diving deep into the specifics of the manual deployment itself but will give you an overview on how it works along the steps of the automation proccess.

Installing Ansible

Installing Ansible is a pretty straightforward proccess, just run:

```
$ brew install ansible            # on OSX
```

or

```
$ [sudo] pip install ansible      # elsewhere
```

-- Take notice that even though it is possible to manage Windows hosts with Ansible, it cannot run on a Windows host natively. An alternative for Windows users is to run it under the Windows Subsystem for Linux (WSL), something we will not cover in this tutorial but can be found on Ansible Official Documentation

Creating an inventory file and a playbook

After installing Ansible, the next thing we will be doing is creating a folder where we will be storing all the files related to the deployment procces

```
$ mkdir deploy
$ cd deploy
```

inside this folder, we will start by creating two files:

An inventory file called "hosts" with a list of hosts you want to manage:

```
$ touch hosts
```

```
[sites]
<YOUR_SERVER_IP>
```

and a playbook.yml file, it's in this file where we will be declaring the configurations and launching the tasks we wish to implement to our proccess:

```
$ touch playbook.yml
```

Building your Ansible playbook:

Setting up and updating your remote server

The first thing we wanna do once we created out playbook file is setting up the server that will be receiving our application. This first two lines of our playbook define the server where we will be deploying our application and the user that will be running the commands to the server. (For the purposes of this tutorial we will be running all the tasks as root. While it might be possible to do it as another user types, it will require further adjustments in privileges and authorizations, something that is out of the scope of this article):

```
- hosts: <YOUR_SERVER_IP>
  user: root
```

After that we will finally start writing the tasks that we want ansible to execute when we run our playbook.

```
- hosts: <YOUR_SERVER_IP>
  user: root
  tasks:
```

The first thing we wanna do is to download and update all the packages in the server system, for that we will be using the ansible.builtin.apt module which manages apt packages (such as for Debian/Ubuntu).

```
- hosts: <YOUR_SERVER_IP>
  user: root
  tasks:
    - name: Update and upgrade apt packages
      become: true
      apt:
        upgrade: yes
        update_cache: yes
```

After that we want to install the python3.8-venv package, this is a dependency needed to set up the virtual environment we will be using later in our server

```
- hosts: <YOUR_SERVER_IP>
  user: root
  tasks:
    [...]

    - name: Install the package "python3.8-venv"
      ansible.builtin.apt:
        name: python3.8-venv
```

Downloading your project to your remote server

Now that we have our server updated and ready to receive our application, we will clone our project from our git repository to the server using the ansible.builtin.git module:

```
- hosts: <YOUR_SERVER_IP>
  user: root
  tasks:
    [...]

    - name: Get repository from git and update locally
      ansible.builtin.git:
        repo: "<GITHUB_REPO_URL>"
        dest: "/home/myproject/"
        clone: yes
        update: yes
```

Setting up a virtual enviroment in your server and installing dependencies

Once our project folder is downloaded to our server, is time to set up the enviroment we will be running our application. For that we will use the ansible.builtin.pip module which allow us to manage Python library dependencies. This task will do the equivalent of what we did locally in the "Setting up a virtual enviroment" step but now inside our deployment server.

```
- hosts: <YOUR_SERVER_IP>
  user: root
  tasks:
    [...]

    - name: Install specified python requirements in indicated virtualenv
      ansible.builtin.pip:
        virtualenv_command: python3 -m venv
        requirements: /home/myproject/requirements.txt
        virtualenv: /home/myproject/venv
```

Configuring Gunicorn

Now that we have our application up and running with all it's needed dependencies is time to serve the application, for that we will be using Gunicorn. We will start by creating a systemd service unit file. This file will allow Ubuntu’s init system to automatically start Gunicorn and serve the Flask application whenever the server boots.

```
$ touch myProjectServices.service
```

we’ll start with the [Unit] section, which is used to specify metadata and dependencies. Add a description of your service here and tell the init system to only start this after the networking target has been reached:

```
[Unit]
Description=Gunicorn instance to serve myproject
After=network.target
```

Next, add a [Service] section. This will specify the user and group that you want the process to run under, in this case we will run it all as root:

```
[Unit]
Description=Gunicorn instance to serve myproject
After=network.target

[Service]
User=root
Group=root
```

Now, we will map out the working directory and set the PATH environmental variable so that the init system knows that the executables for the process are located within our virtual environment.

Also specify the command to start the service. This command will do the following: Provide the full path to the Gunicorn executable, which is installed within our virtual environment. Start 3 worker processes Create and bind to a Unix socket file: "myproject.sock" within your project directory. Tells Gunicorn to load your Flask application by binding the module import name of your application file alongside with the Python callable within that file (myproject:app)

```
[Unit]
Description=Gunicorn instance to serve mysite
After=network.target

[Service]
User=root
Group=root
WorkingDirectory=/home/myproject
Environment="PATH=/home/myproject/venv/bin"
ExecStart=/home/myproject/venv/bin/gunicorn --workers 3 --bind unix:mysite.sock
myproject:app
```

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

```
[Unit]
Description=Gunicorn instance to serve mysite
After=network.target

[Service]
User=root
Group=root
WorkingDirectory=/home/myproject
Environment="PATH=/home/myproject/venv/bin"
ExecStart=/home/myproject/venv/bin/gunicorn --workers 3 --bind unix:mysite.sock
myproject:app

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

Save it in your deployment folder and close it.

Now that we have our service file set up we want to upload it to our server system. To do this we will add another task to our playbook using the ansible.builtin.copy module, which allow us to copy a file from the local machine to a location on the remote machine.

```
- hosts: <YOUR_SERVER_IP>
  user: root
  tasks:
    [...]

    - name: Copy service file with owner and permissions to server
      ansible.builtin.copy:
        src: myProjectServices.services
        dest: /etc/systemd/system/
        owner: root
        group: root
        mode: '0777'
```

We can now start the Gunicorn service that we created and enable it so that it starts at boot. To do this we will use the ansible.builtin.systemd module that is used to control systemd units on remote hosts

```
- hosts: <YOUR_SERVER_IP>
  user: root
  tasks:
    [...]

    - name: Start gunicorn service
      systemd:
        name: myproject
        state: restarted
        daemon_reload: yes

    - name: Enable gunicorn service
      systemd:
        name: myproject
        enabled: true
```

Take notice that we are setting the state to "restarted" instead of "started" and adding the "daemond_reload" rule, this is necessary because idempotence is offered as a built-in feature of many of the Ansible modules, this means that, once a task has been completed and the service is running, reprompting the task will not make any changes to the state of the target node, something that is not desireable when we are making adjustments in our services and need Gunicorn to start over to reflect this changes.

Installing and Configuring Nginx to receive Proxy Requests

Our Gunicorn application server should now be up and running, waiting for requests on the socket file in the project directory. To pass web requests to that socket we will be using Nginx. The first thing we wanna do is to install Nginx to our server, to do that we will once again use the ansible.builtin.apt module and add the following task

```
- hosts: <YOUR_SERVER_IP>
  user: root
  tasks:
    [...]

    - name: Install nginx
      ansible.builtin.apt:
        name: nginx
```

Now we want to allow the acces from Nginx to our ports, to do that we will set up some firewall permissions using the community.general.ufw module. Notice that we set up the services we want to allow before enabling the firewall and that we enabled OpenSSH alongside Nginx. We do this because if we enable the firewall without adding the rules before, it will be set to its original configuration which denies everything by default.

Another interesting thing to notice here is the usage of a loop to do similar tasks within the same module in Ansibe (in this case the rule "allow" both to "OpenSSH" and to "Nginx Full"), this is an simple and effective way to save you some time and avoid repetition when building your playbook. You can read more about using loops in Ansible here

```
- hosts: <YOUR_SERVER_IP>
  user: root
  tasks:
    [...]

    - name: alow ssh
      community.general.ufw:
        name: "{{item}}"
        rule: allow
      loop:
        - OpenSSH
        - Nginx Full

    - name: enable UFW
      community.general.ufw:
        state: reloaded
```

Before we start Nginx we want to configure it to pass web requests to the sock. We will start by creating a configuration file. We will call this myprojectRequests to keep in line with the rest of the guide:

```
$ touch myProjectRequests
```

In this first part we will set up a server block and tell Nginx to listen on the default port 80. It also tell it to use this block for requests for our server's domain name:

```
server {
    listen 80;
    server_name <your_domain> <www.your_domain>;
}
```

Next, add a location block that matches every request. Within this block, you'll include the proxy_params file that specifies some general proxying parameters that need to be set. You'll then pass the requests to the socket you defined using the proxy_pass directive:

```
server {
    listen 80;
    server_name <your_domain> <www.your_domain>;

    location / {
        include proxy_params;
        proxy_pass http://unix:/home/myproject/myproject.sock;
    }
}
```

Save it in your deployment folder and close it.

Now that we have our server block configuration file set up we want to copy it to Nginx's sites-available directory in our server. We do this the same way we did with our services file using ansible.builtin.copy module

```
- hosts: <YOUR_SERVER_IP>
  user: root
  tasks:
    [...]

     - name: Copy server block configuration file to server
      ansible.builtin.copy:
        src: myProjectRequests
        dest: /etc/nginx/sites-available/myproject
        owner: root
        group: root
        mode: '0777'
```

To enable the Nginx server block configuration you’ve just created, link the file to the sites-enabled directory. Do this by creating a symlink between the sites-available to sites-enabled directory using the ansible.builtin.file module that allows you to set attributes to files, directories, or symlinks and their targets.

```
- hosts: <YOUR_SERVER_IP>
  user: root
  tasks:
    [...]

    - name: Creating a symlink between sites-available and sites-enabled directories
      ansible.builtin.file:
        src: /etc/nginx/sites-available/myproject
        dest: /etc/nginx/sites-enabled/myproject
        state: link
```

Now we will start nginx using the ansible.builtin.systemd module. Similarly to Gunicorn we want to set the state to "restarted" instead of "started" as we want it to reflect eventual changes we do in it's configuration every time we run our deployment playbook.

```
- hosts: <YOUR_SERVER_IP>
  user: root
  tasks:
    [...]

    - name: Start nginx
      systemd:
        name: nginx
        state: restarted
```

Defining variables

When dealing with local files and paths we do not want them hard coded in our playbook tasks, as it can gets considerably hard and confusing to replace one by one, either if in the future we are using the same playbook to deploy a different applicattion or if we are redeploying the same application but changed the path to a file or directory or altered the name of a specific file. For that reason it is good practice to define them as variables at the begining of our playbook and replace them on the body of our tasks.

After doing that our playbook should be looking something like this:

```
- hosts: <YOUR_SERVER_IP>
  vars:
    git_repo: '<GITHUB_REPO_URL>'
    folder_project_destination: /home/myproject/
    requirements_path: /home/myproject/requirements.txt
    venv_path: /home/myproject/venv
    service_file: myProjectServices.service
    block_configuration_file: myProjectRequests
  user: root
  tasks:
    - name: Update and upgrade apt packages
      become: true
      apt:
        upgrade: yes
        update_cache: yes

    - name: Install the package "python3.8-venv"
      ansible.builtin.apt:
        name: python3.8-venv

    - name: Get repository from git and update locally
      ansible.builtin.git:
        repo: "{{git_repo}}"
        dest: "{{folder_project_destination}}"
        clone: yes
        update: yes

    - name: Install specified python requirements in indicated virtualenv
      ansible.builtin.pip:
        virtualenv_command: python3 -m venv
        requirements: "{{requirements_path}}"
        virtualenv: "{{venv_path}}"

    - name: Copy service file with owner and permissions to server
      ansible.builtin.copy:
        src: "{{service_file}}"
        dest: /etc/systemd/system/
        owner: root
        group: root
        mode: '0777'

    - name: Start gunicorn service
      systemd:
        name: "{{service_file}}"
        state: restarted
        daemon_reload: yes

    - name: Enable gunicorn service
      systemd:
        name: "{{service_file}}"
        enabled: true

    - name: Install nginx
      ansible.builtin.apt:
        name: nginx

    - name: alow ssh
      community.general.ufw:
        name: "{{item}}"
        rule: allow
      loop:
        - OpenSSH
        - Nginx Full

    - name: enable UFW
      community.general.ufw:
        state: reloaded

    - name: Copy server block configuration file to server
      ansible.builtin.copy:
        src: "{{block_configuration_file}}"
        dest: /etc/nginx/sites-available/{{block_configuration_file}}
        owner: root
        group: root
        mode: '0777'

    - name: Creating a symlink between sites-available and sites-enabled directories
      ansible.builtin.file:
        src: /etc/nginx/sites-available/{{block_configuration_file}}
        dest: /etc/nginx/sites-enabled/{{block_configuration_file}}
        state: link

    - name: Start nginx
      systemd:
        name: nginx
        state: restarted
```

Runing your Ansible playbook

Now that we have our playbook set up with all the tasks necessary to make our deployment we will run it from inside our deploy folder with the command:

```
$ ansible-playbook -i hosts playbook.yml
```

You will se a log from the tasks that are being executed:

```
PLAY [5.161.90.22]
****************************************************************************************



TASK [Gathering Facts]
****************************************************************************************

ok: [5.161.90.22]

TASK [Update and upgrade apt packages]
****************************************************************************************

ok: [5.161.90.22]

```

```
TASK [Install the package "python3.8-venv"]
****************************************************************************************

ok: [5.161.90.22]

TASK [Get repository from git and update locally]
****************************************************************************************

ok: [5.161.90.22]

TASK [Install specified python requirements in indicated virtualenv]
****************************************************************************************

ok: [5.161.90.22]

TASK [Copy service file with owner and permissions to server]
****************************************************************************************

ok: [5.161.90.22]

TASK [Start gunicorn service]
****************************************************************************************

changed: [5.161.90.22]

TASK [Enable gunicorn service]
****************************************************************************************

ok: [5.161.90.22]

TASK [Install nginx]
****************************************************************************************

ok: [5.161.90.22]

TASK [alow ssh]
****************************************************************************************

ok: [5.161.90.22] => (item=OpenSSH)
ok: [5.161.90.22] => (item=Nginx Full)

TASK [enable UFW]
****************************************************************************************

changed: [5.161.90.22]

TASK [Copy server block configuration file to server]
****************************************************************************************

ok: [5.161.90.22]

TASK [Creating a symlink between sites-available and sites-enabled directories]
```

```
***************************************************************************************

ok: [5.161.90.22]

TASK [Start nginx]
***************************************************************************************

*changed: [5.161.90.22]

PLAY RECAP
***************************************************************************************

5.161.90.22                : ok=14   changed=3    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

If it was able to run through all the taks until the end without errors, your application should be deployed and you should now be able to navigate to your server’s domain name in your web browser:

```
http://your_domain
```

Conclusion

In this guide, we created a simple Flask application within a Python virtual environment, then we configured the Gunicorn app to serve this application. Afterwards, we created a systemd service file to automatically launch this application server on boot. We then created an Nginx server block that passes web client traffic to the application server, relaying external requests to our domain. All those steps were set up and saved as a series of tasks in an Ansible playbook that now can be executed as many times as needed and might be reused for future deployments. This shows how useful, simple and practical might be automating your deployment proccess using Ansible.