NiviGroup

QUANTIC RESEARCH

Automated deploy of a Python 3/Django project with Ansible

Marco Santamaria
Python Developer @ Quantic Research - Nivi Group
marco.santamaria@gmail.com

“There should be one-- and preferably only one --obvious way to do it” (from The zen of Python)

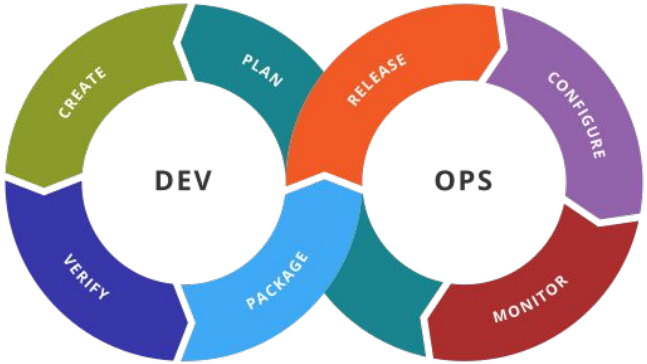
Is that true for Python web application deployment? Really?

In my experience deployment is a critical subject for the Python ecosystem and is one of the activity that scares more who comes to Python for the first time.

Furthermore, many professional Django developer are used to deploy with one of the good PaaS solution available today (Heroku, PythonAnywhere,...), without a clear understanding of what is going under the hood.

But in many real world projects happens that you need to manage directly a traditional server infrastructure.

DevOps mindset



Developers often see deployment as a duty of a different person like a system administrator or a member of the operations department.

If you want to embrace modern DevOps culture, is better to think about deployment together with development and testing as parts of the same process.

In this whole process developers are fully involved in each step and must have a good understanding about deployment, server configuration and how to automate them.

Packaging your app

I am going to use Django in the next slides, but the same applies to other Python web frameworks. I will try to be framework agnostic in the approach.

The first step to properly deploy a web app is to properly package it. There are many options to do the job, but for our purposes a good choice is to use the standard tool provided by the PyPA: setuptools.

One important thing is to craft a **`setup.py`** file at the root of your project. It is also recommended to include a single Python package for your project.

Let's say we have a Django project named **`pycon8`**.

Packaging your app - setup.py

```
from pycon8 import __version__
from setuptools import setup,
find_packages

setup(
    name='pycon8',

packages=find_packages(exclude=['tests']
),
    include_package_data=True,
    version=__version__,
    install_requires=[]
)
```

This is a minimal setup.py file for it.

It is also a good idea to set in the root `__init__.py` file the package version.

In this way you should be able to install the app in the current Python environment with the command:

```
$ pip install .
```

Packaging your app - concrete requirements

Another important thing to do is freezing your Python dependencies in a pip requirements file.

For an application the production requirements file should list all the Python packages needed, including second level dependencies, with all the versions freezed to make the deploy repeatable.

The setup.py file also includes an install_requires list, but they should be considered like “abstract requirements” for a library as opposed to the “concrete requirements” of an application listed in pip requirements files, as explained in this excellent article by Donald Stufft: https://caremad.io/posts/2013/07/setup-vs-requirement/.

Packaging your app - requirements.txt

```
appdirs==1.4.3
decorator==4.0.11
Django==1.10.6
packaging==16.8
pexpect==4.2.1
pickleshare==0.7.4
prompt-toolkit==1.0.14
ptyprocess==0.5.1
Pygments==2.2.0
pyparsing==2.2.0
simplegeneric==0.8.1
six==1.10.0
traitlets==4.3.2
wcwidth==0.1.7
.
```

Also the application we want to deploy should be included in the requirements file, using a dot (last item on the left).

In this way we can install with a single command both the app and its requirements:

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

Configure your app - the problem

We have a single codebase where we store the app source code, but different environments where that code will (hopefully) run. There are several parameters that are likely to vary between environments: credentials to databases or external services, log level, etc. But we don’t want to change the code every time we deploy to a different environment.

Creating a Python settings file excluded from version control is risky because it can be accidentally committed. Also it can easily introduce error when created for a particular environment. If you include it under version control with some sample values, then you need to change it in place when you deploy, and this is error prone, again.

Configure your app - twelve-factor approach

Adam Wiggins, cofounder of Heroku, maintains an on line document that tries to outline the 12 most important “factors” that make a modern web application cleanly configurable and deployable.

Without being religious about it, I truly believe that it is worth following most of the principles stated there.

About dependency management, I already discussed the use of PIP and requirements files.

About app configuration, the advice is to rely on environment variables.

The twelve factors

- **I. Codebase**: *One codebase tracked in revision control, many deploys*
- **II. Dependencies**: *Explicitly declare and isolate dependencies*
- **III. Config**: *Store config in the environment*
- **IV. Backing services**: *Treat backing services as attached resources*
- **V. Build, release, run**: *Strictly separate build and run stages*
- **VI. Processes**: *Execute the app as one or more stateless processes*
- **VII. Port binding**: Export services via port binding
- **VIII. Concurrency**: Scale out via the process model
- **IX. Disposability**: Maximize robustness with fast startup and graceful shutdown
- **X. Dev/prod parity**: Keep development, staging, and production as similar as possible
- **XI. Logs**: Treat logs as event streams
- **XII. Admin processes**: Run admin/management tasks as one-off processes

Configure your app - dot env files

It is possible to store several environment variables in a single ini-like file (without sections) that let you bundle in a single artifact all the configuration for a given environment. This kind of file usually has the “.env” extension.

In Python it is easy to read an environment variable using the standard library, but there are a bunch of excellent Python packages that let you read dot env files: python-dotenv, django-environ (django specific), honcho, envparse.

I will use python-dotenv because I like the motto “Do one thing, do it well!”.

Configure your app - dot env file

```
DJANGO_SETTINGS_MODULE=pycon8.settings
DEBUG=False
LOG_LEVEL=INFO
SECRET_KEY=6n(eg$)4^^)1p3jyn)4_s2f)&&%x!
DB_HOST=192.168.33.11
DB_NAME=pycon8
DB_USER=pycon8
DB_PASSWORD=Py.C0n.8
DB_PORT=5432
STATIC_ROOT=/opt/pycon8/static
```

A sample .env file

```
import os
from pathlib import Path
from dotenv import load_dotenv, find_dotenv
load_dotenv(find_dotenv())

root_path = Path(__file__).parent.parent

SECRET_KEY = os.getenv('SECRET_KEY',
'S3cr3t.Key')
DEBUG = bool(os.getenv('DEBUG', 'False'))

...
```

How to use python-dotenv in your settings

WSGI

Django developers are familiar with the runserver command, that starts a local development web server, but, as stated in the official documentation, it is not suitable for production.

To interact with a real web server there is the WSGI (Web Server Gateway Interface) standard, which defines an interface between the web server and a Python web application. Django is compatible with WSGI out of the box and every Django project has a **`wsgi.py`** file where is defined an **`application`** object.

There are several implementation of the WSGI specification: uWSGI, Gunicorn, Apache mod_wsgi are the most popular choices.

WSGI - Django wsgi.py file

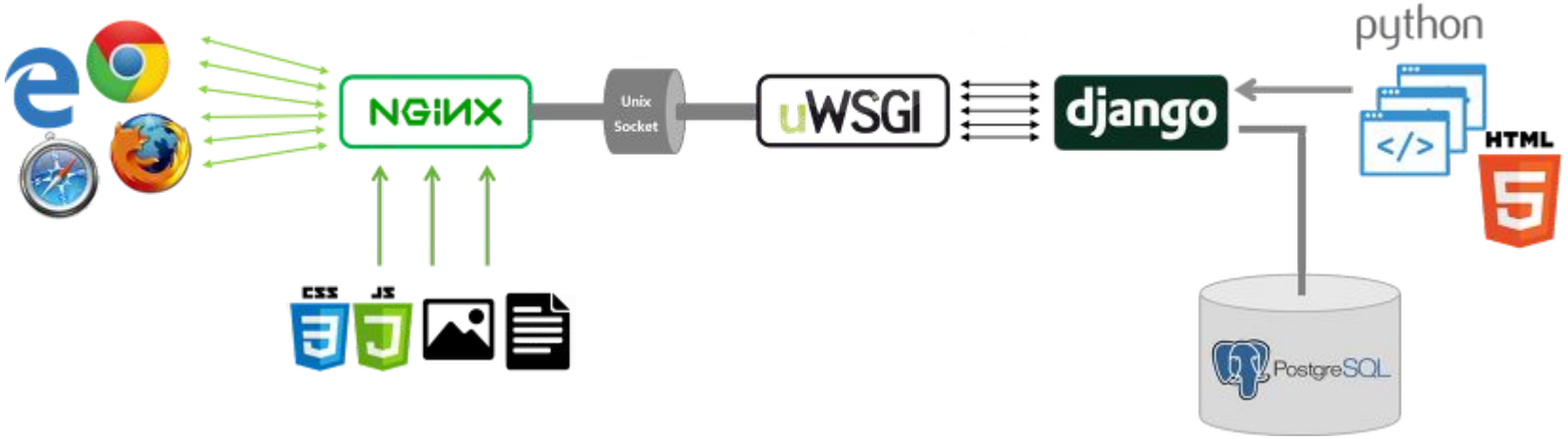
```
"""
WSGI config for pycon8 project.

It exposes the WSGI callable as a module-level variable named
``application``.
"""
import os
from django.core.wsgi import get_wsgi_application

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "pycon8.settings")

application = get_wsgi_application()
```

WSGI - architecture



This is the system architecture we want to implement. I'm going to use CentOS 7 as a Linux distribution, but is easy to replicate the same approach on Debian.

uWSGI

uWSGI is a good option to connect your application with a web server:

- Is written in C by a cool italian development team
- Is fast, stable, reliable
- Support multiple applications with the emperor mode
- It has an option for everything

It can be installed in several ways, but under the major Linux distribution (CentOS, Debian, etc.) it comes prepackaged in a recent version with a nice emperor configuration.

uWSGI - configuration

```
[uwsgi]
plugin = python36u
home = /opt/pycon8
module = pycon8.wsgi:application
socket = /opt/pycon8/uwsgi.sock
chmod-socket = 660
daemonize =
/var/log/pycon8/uwsgi.log

harakiri = 30
enable-threads = true
vacuum = true

# Read environment variables
for-readline = /opt/pycon8/.env
  env = %(_)
endfor =
```

To make uWSGI aware of your application in CentOS 7 the only thing you need to take care of is creating a vassal configuration file:

`/etc/uwsgi.d/pycon8.ini`

There's no need to change the main configuration file.

Then you restart the uwsgi service with the command:

```
$ systemctl restart uwsgi
```

NGiNX

Nginx is a lightweight and fast web server written in C that plays nicely with uWSGI.

It can be installed easily under CentOS with yum.

Then you can create a configuration file **`/etc/nginx/conf.d/pycon8.conf`** for your app and restart the service:

```
$ systemctl restart nginx
```

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    server_name "";

    location = favicon.ico {
        access_log off;
        log_not_found off; }

    location /static/ {
        root /opt/pycon8;
    }

    location / {
        include uwsgi_params;
        uwsgi_pass unix:/opt/pycon8/uwsgi.sock;
        uwsgi_read_timeout 30s;
        uwsgi_send_timeout 30s;
    }
}
```

Ansible - Introduction

Ansible is an automation tool written in Python that lets you execute tasks on remote machines using the YAML declarative format.

It can be used to automate the configuration of a server and the deployment of a Python web application. It shines in orchestrating multiple application deployment on multiple machines.

It is agentless: doesn’t require an agent to be installed on the target machines, only OpenSSH (and WinRM on Windows) and Python 2.6 or later.

Ansible - inventories

Ansible let you perform operations on a set of servers defined in an **inventory** file.

An inventory file is an ini file where you define the properties of your machines. Every machine has a label, an ip, ssh credentials, and several other optional properties.

It is possible to define a group of hosts using an ini file section.

A good practice is to have a separate inventory file for each environment: staging, development, production, etc.

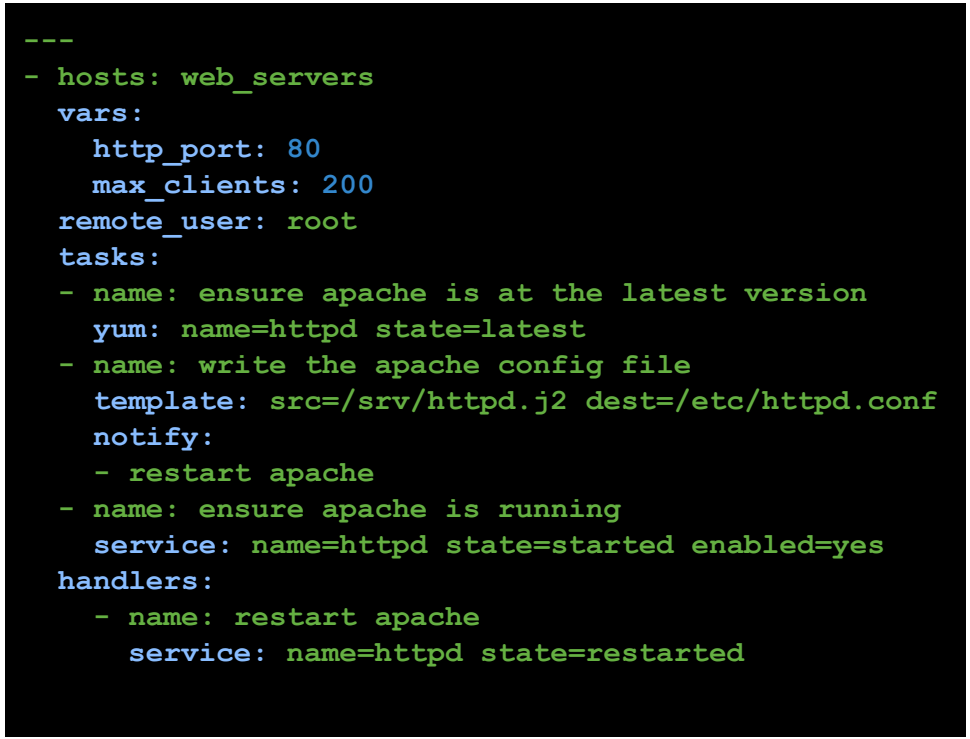
Ansible - playbooks, tasks and modules

Ansible **palybooks** are YAML files that define what **tasks** are going to be performed. They are organized as a list of **plays**, where each play lists some tasks to be executed against a set of machines from an inventory.

A task is essentially a call to a **module** with a name and some parameters. Ansible ships with a large number of modules that let you perform many different actions on remote machines: check in the official documentation for all modules.

Modules try to be **idempotent** (when possible): the result of performing a module once is exactly the same as the result of performing it repeatedly without any intervening actions.

Ansible - a playbook example



```
---
- hosts: web_servers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum: name=httpd state=latest
  - name: write the apache config file
    template: src=/srv/httpd.j2 dest=/etc/httpd.conf
    notify:
    - restart apache
  - name: ensure apache is running
    service: name=httpd state=started enabled=yes
  handlers:
    - name: restart apache
      service: name=httpd state=restarted
```

This is a playbook with a single play that executes a list of three tasks against the web_servers inventory group.

It also defines an **handler**, that is notified by the second task.

An handler will be executed at the end of the task list.

Ansible - playbook execution

To execute a playbook named all_serves.yml against the staging inventory, you can use the following command:

```
$ ansible-playbook -i hosts/staging all_servers.yml
```

It is also possible to pass additional variables using the extra-vars argument:

```
$ ansible-playbook -i hosts/staging all_servers.yml \

   --extra-vars "pycon8=0.1.0"
```

Ansible - roles and dependencies

An Ansible **role** is a way to bundle together a set of related tasks and make them available for playbooks. They are normally defined in a separate **`roles`** folder. Each role contains also related handlers, files and variables.

It is also possible to define **dependencies** between two roles. Let's say that we defined a **`python_app`** role. If we want to impose that it requires the execution of the **`python_server`** role we can add a meta/main.yml file contained within the role directory:

```
---

allow_duplicates: yes
dependencies:
- { role: python_server }
```

Ansible - variables and templates

In Ansible **variables** are key that store some value and can be defined at various levels: inventory files, playbooks, roles, tasks, extra_vars. There are specific precedence rules that can be found in the documentation.

Ansible allows you to reference variables in your playbooks using the Jinja2 templating system. Using the Jinja2 curly bracket syntax it is possible to inject the value of a variable in playbook, roles and inventory files.

Roles may also include a `templates` folder where you can store Jinja2 template for configuration files that can be filled with appropriate values.

Ansible - template example


```
[uwsgi]
plugin = python36u
home = /opt/{{ app_name }}
module = {{ app_name
}}.wsgi:application
socket = /opt/{{ app_name }}/uwsgi.sock
chmod-socket = 660
daemonize = /var/log/{{ app_name
}}/uwsgi.log

harakiri = 30
enable-threads = true
vacuum = true

# Read environment variables
for-readline = /opt/{{ app_name }}/.env
  env = %(_)
endfor =
```

This is the uWSGI configuration file template where the **app_name** variable will be filled during the playbook execution with the name of the app to be deployed.

Ansible - layout


```
hosts/
   production              # inventory file for production servers
   staging                 # inventory file for staging environment
all_servers.yml            # master playbook
web_servers.yml            # playbook for web_server tier
db_servers.yml             # playbook for db_server tier
roles/
    my_role/               # this hierarchy represents a "role"
        tasks/             #
            main.yml       #  tasks file can include smaller files if warranted
        handlers/          #
            main.yml       #  handlers file
        templates/         #  files for use with the template resource
            ntp.conf.j2    #  templates end in .j2
        defaults/          #
            main.yml       #  default lower priority variables for this role
        meta/              #
            main.yml       #  role dependencies
```

Ansible - example playbook for Django

Now let's put together all the concepts and technologies described until now.

At the following URL you can find an Ansible playbook that performs the configuration of two machines (database server and web server) and the deployment of a single Django app named pycon8:

https://github.com/marco-santamaria/django-ansible-deploy

The master playbook, named **`all_servers.yml`** can be found in the deploy folder and can be executed on local Vagrant machines with the command:

```
$ ansible-playbook -i hosts/local all_servers.yml --extra-vars "pycon8=0.1.0"
```

Ansible - example playbook for Django

In the playbook you can find four roles:

- **`postrgresql_server`** (configures a CentOS 7/PostgreSQL database server)
- **`postgresql_db`** (creates a PostgreSQL database and depends on postgresql_server)
- **`python_server`** (configures a CentOS 7/Python 3.6/uWSGI/NGiNX web server)
- **`python_app`** (deploys a Django project and depends on python_server)

The app and the playbook do not include the full complexity of a real world project, but they aim to be a good starting point to deploy automatically a Python app.

Any questions?

Thanks!