How to Deploy Python WSGI Applications Using a CherryPy Web Server Behind Nginx

Authored by: ***ASPHostServer Administrator*** *[asphostserver@gmail.com]*
Saved From: *http://faq.asphosthelpdesk.com/article.php?id=153*

Introduction

Chances are you found yourself asking one of these questions after reading this tutorial's title:

- Why should I use CherryPy's pure-Python web server instead of a "real" stand-alone (perhaps C based) one for my WSGI application?
- CherryPy...Isn't that a framework? What's that got to do with the deployment of my Bottle, Flask etc. based application?

There are many good answers and reasons for both. In this DigitalOcean article, we are going to find them. We will begin with talking about what CherryPy exactly is and the advantages of utilizing it for your web application. We will continue with explaining "why" and most importantly "how" you can deploy a Python application using CherryPy's Web Server.

Glossary

1. Understanding CherryPy and Using Nginx

1. CherryPy WSGI Web Server In Brief
2. Why Deploy With CherryPy's WSGI Web Server?
3. Using Nginx as Reverse-Proxy In Front of CherryPy

2. Preparing Production

1. Updating the default operating system
2. Setting up Python, pip and virtualenv
3. Creating a Virtual (Python) Environment
4. Downloading and installing CherryPy
5. Downloading and installing Nginx

3. Serving Python Web Applications with CherryPy Web Server

1. WSGI
2. WSGI Application Object (Callable): wsgi.py
3. Creating a script to use CherryPy Web Server: server.py
4. Running the server

4. Configuring Nginx

5. Miscellaneous Tips and Suggestions

Understanding CherryPy and Using Nginx

CherryPy as a whole is a minimalist Python Web Framework. What minimalist here means is it is not shipped with too many components out-of-the-box, whether you like (or need) them or not. Minimalism is basically refraining from imposing things on the developers without giving them a choice. CherryPy - and other such frameworks for that matter - usually handle the core necessities one would expect (e.g. sessions, caching, file uploads et al.) and leave the rest - and the choice - of what to use and how to use to be decided by you.

What separates CherryPy from other Python frameworks (including some "fully fledged" ones) is its developers' ambition to provide it ready to work (i.e. develop) with and ready to deploy as a self contained application bundle (package). In order to achieve this task, for web applications, a solid web server is a must. This is exactly where CherryPy excels, with its very own HTTP/1.1-compliant, WSGI thread-pooled Web Server. It is production-ready and just fantastic.

CherryPy WSGI Web Server In Brief

CherryPy's pure Python web server is a compact solution which comes with the namesake framework. Defined by the [CherryPy] project as a high-speed, production ready, thread pooled, generic HTTP server. It is a modularized component which can be used to serve any Python WSGI web application.

CherryPy Web Server's Highlights:

- A very compact and simple to use pure-Python solution
- Easy to configure, easy to use
- Thread-pooled and fast
- Allows scaling
- Supports SSL

Why Deploy With CherryPy's WSGI Web Server?

As we mentioned at the beginning, you are probably wondering about the reasons to use this solution instead of another hyped and famous one that you might have heard of or even tried. The truth of the matter is, thanks to the excellent specification of WSGI, it has been fairly easy to create a web server. Over the years, this gave birth to many of them, with some reaching a certain level of popularity and most remaining hidden inside its developer's machine.

The amount of choices are vast, and they all (mostly) do the same thing under-the-hood to a large extent.

Why exactly should you use CherryPy web server for your application's deployment?

The answer is rather simple: it is a joy to work with. The ease of use to serve your WSGI web application using CherryPy's server is exceptional. It will save you tons of headaches as you get up and running within a minute or two. It is customizable to a certain degree, giving you the ability to run both multi-process and multi-threaded instances the simplest way possible through a single file (e.g. server.py).

Remember: For multi-process setups, you will need more than one "server" object instance configured.

Unless you are certain that within minutes of your application going online it will be getting tens of thousands of requests per second and you cannot simply use more servers to balance the load, then you are better off using your time to continue developing the application than hassling with libraries, CPU optimizations, dealing with crashes, etc.

CherryPy Web Server, coupled with (ease) using Nginx as a front-facing reverse-proxy is a truly rock solid way to serve Python WSGI based web applications, whether it be developed on top of Bottle, CherryPy, Django, Flask, Pyramid or any other framework.

Remember: The above mentioned architecture (explained below) enables you to easily scale horizontally (more servers) and even vertically (more capacity per server). Your "bottleneck" is likely to be the backend (database). That is why it"s not worth your while to try to optimize things to death before even having a promise of extreme loads. Even then, a sane caching mechanism, if introduced, will probably solve most of your problems.

Using Nginx as Reverse-Proxy In Front of CherryPy

Nginx is a very high performant web server /* (reverse)-proxy. It has reached its popularity due to being light weight, relatively easy to work with and easy to extend (with add-ons / plug-ins). Thanks to its architecture, it is capable of handling *a lotof requests (virtually unlimited), which - depending on your application or website load - could be really hard to tackle using some other, older alternatives.

Remember:* "Handling" connections technically means not dropping them and being able to serve them with *something. You still need your application and database functioning well in order to have Nginx serve clients responses that are not error messages.

Why exactly should you use Nginx as a reverse-proxy in front of an application server?

Although your application server - CherryPy WSGI web server in our case - can serve your application and its static files (e.g. javascript, css, images etc.), it is a very good idea to make use of a reverse-proxy, set up in front, such as Nginx. This relieves a lot of the load [from the application servers] as it handles the client requests (and overheads) and various other tasks, granting you a much better overall performance.

As your application grows, you will want to optimize it and when the time comes, distribute it across servers to be able to handle more connections simultaneously and have a generally more robust architecture. Having a reverse-proxy in front of your application server(s) helps you with this from very beginning as well.

Its extensibility (e.g. native caching along with failover and other mechanisms) is also a great feat that benefits web applications unlike (simpler) application servers.

Example of a Basic Server Architecture:

Client Request ----> Nginx (Reverse-Proxy)

|

/|\

| | `-> App. Server I. 127.0.0.1:8081

| `--> App. Server II. 127.0.0.1:8082

`----> App. Server III. 127.0.0.1:8083

Note:* When an application is set to listen for incoming connections on 127.0.0.1, it will only be possible to access it locally. If you use 0.0.0.0, however, it will accept connections from *the outsideas well.

Preparing Production

In this section, we are going to prepare for production (i.e. for deploying our application).

We will begin with:

- updating the default operating system
- downloading and installing common Python tools (i.e. pip, virtualenv)
- and creating a virtual environment to contain the application (inside which its dependencies such as CherryPy reside).

Updating the default operating system

To ensure that we have the latest available versions of default applications, we need to update our system.

For Debian Based Systems (i.e. Ubuntu, Debian), run the following:

aptitude update

aptitude -y upgrade

For RHEL Based Systems (i.e. CentOS), run the following:

yum -y update

Setting up Python, pip and virtualenv

Note for CentOS / RHEL Users:

CentOS / RHEL, by default, comes as a very lean server. Its toolset, which is likely to be dated for your needs, isnot there to run your applications but to power the server's system tools (e.g. YUM).

In order to prepare your CentOS system, Python needs to be set up (i.e. compiled from the source) and pip/ virtualenv need installing using that interpreter.

On Ubuntu and Debian, a recent version of Python interpreter which you can use comes by default. It leaves us with only a limited number of additional packages to install:

- python-dev (development tools),
- pip (to manage packages),
- virtualenv (to create isolated, virtual environments).

python-dev:

python-dev is an operating-system level package which contains extended development tools for building Python modules.

Run the following command to install python-dev using aptitude:

aptitude install python-dev

pip:

pip is a package manager which will help us to install the application packages that we need.

Run the following commands to install pip:

curl https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py | python -

curl https://raw.github.com/pypa/pip/master/contrib/get-pip.py | python -

export PATH="/usr/local/bin:$PATH"

You might need sudo privileges.

virtualenv:

It is best to contain a Python application within its own environment, together with all of its dependencies. An environment can be best described (in simple terms) as an isolated location (a directory) where everything resides. For this purpose, a tool called virtualenvis used.

Run the following to install virtualenv using pip:

sudo pip install virtualenv

Creating a self-contained Virtual (Python) Environment

Having all the necessary tools ready at our disposal, we can create an environment deploy our application.

Remember: If you haven't got a virtualenv on your development (local) machine for your project, you should consider creating one and moving your application (and its dependencies) inside.

Let's begin with creating a folder which will contain both the virtual environment and your application module:

You can use any name here to suit your needs.

mkdir my_app

We can continue with entering this folder and creating a new virtual environment inside:

You can also choose any name you like for your virtual environment.

cd my_app

virtualenv my_app_venv

Let's create a new folder there to contain your Python application module as well:

This is the folder where your application module will reside.

mkdir app

And activate the interpreter inside the virtual environment to use it:

Please make sure to use the name you chose for your virtual environment if you went for something other than "my_app_venv".

source my_app_venv/bin/activate

In the end, this is how your main application deployment directory should look like:

my_app # Main Folder to Contain Everything Together

|

|=== my_app_venv # V. Env. folder with the Python Int.

|=== app # Your application module

|..

|.

Downloading and installing CherryPy

In order to use CherryPy's WSGI Web Server, we first need to have it downloaded and installed.

To install CherryPy using pip, run the following:

pip install cherrypy

Note: **If you are working inside an environment, CherryPy will be installed there. Otherwise, the installation will be *globally available(i.e. systemwide). Global installation is not recommended. Always opt for usingvirtualenv both on your server and your development machine.***

Downloading and installing Nginx

Note for CentOS / RHEL Users:

The below instructions will not work on CentOS systems.

Run the following command to use the default system package manager aptitude install Nginx:

sudo aptitude install nginx

To run Nginx, you can use the following:

sudo service nginx start

To stop Nginx, you can use the following:

sudo service nginx stop

To restart Nginx, you can use the following:

After each time you reconfigure Nginx, a restart or reload is needed for the new settings to come into effect.

sudo service nginx restart

Serving Python Web Applications with CherryPy Web Server

In this section, we will see how a WSGI application works with CherryPy web server. This process consists of providing the server with a WSGI application callable (e.g. application = (..)) as the point of entry.

WSGI

WSGI in a nutshell is an interface between a web server and the application itself. It exists to ensure a standardized way between various servers and applications (frameworks) to work with each other, allowing interchangeability when necessary (e.g. switching from development to production environment), which is a must-have need nowadays.

WSGI Application Object (Callable): wsgi.py

As mentioned above, web servers running on WSGI need an application object (i.e. your application's).

With most frameworks and applications, this consists of:

- A wsgi.py to contain and provide an application object (a callable) to be used by the server.
- A wsgi.py to contain and provide an application object (a callable) to be used by the server.

We will begin with creating an exemplary "wsgi.pyâ€• and continue with creating a common "server.pyâ€• for CherryPy.

You can choose any name instead of wsgi.py and server.py. However, these are the ones that are commonly used (e.g. by Django).

Let's begin with creating a wsgi.py file to contain a basic WSGI application.

Run the following command to create a wsgi.py using the text editor nano:

nano wsgi.py

Let's continue with moving (copy/paste) the basic WSGI application code inside (which should be replaced with your own application's callable for production):

```
def application(env, start_response):

start_response('200 OK', [('Content-Type', 'text/html')])

return ["Hello!"]
```

This is the file that is included by the server and each time a request comes, the server uses this application callable to run the application's request handlers (i.e. controllers) upon parsing the URL (e.g. mysite.tld/controller/method/variable).

After placing the application code in, press CTRL+X and then confirm with Y to save this file inside the my_appfolder alongside the virtual environment and the app module containing your actual application.

Note: This WSGI application is the most basic example to its kind. You will need to replace this code

PDF Generated by PHPKB Knowledge Base Script

block to include your own application object from the application module.

Once we are done, this is how your main application deployment directory should look like:

my_app # Main Folder to Contain Everything Together

|

|=== my_app_venv # V. Env. folder with the Python Int.

|=== app # Your application module

|

|--- wsgi.py # File containing application callable

|..

|.

Creating a script to use CherryPy Web Server:server.py

Inside our current working directory (e.g. my_app) folder, we now need to create a Python script which:

- Includes the application
- Creates one or more CherryPy web server instances
- Configures these server instances
- Starts and stops the server engine(s)

There are multiple ways to obtain such script and in the below example, we will use a simple but easy to configure one.

Create a server.py file to contain the web server launch script using nano:

nano server.py

Inside this file, write (copy/paste) the following server snippet:

Import your application as:

from wsgi import application

Example:

from wsgi import application

Import CherryPy

import cherrypy

PDF Generated by PHPKB Knowledge Base Script

```
if __name__ == '__main__':

# Mount the application

cherrypy.tree.graft(application, "/")

# Unsubscribe the default server

cherrypy.server.unsubscribe()

# Instantiate a new server object

server = cherrypy._cpserver.Server()

# Configure the server object

server.socket_host = "0.0.0.0"

server.socket_port = 8080

server.thread_pool = 30

# For SSL Support

# server.ssl_module = 'pyopenssl'

# server.ssl_certificate = 'ssl/certificate.crt'

# server.ssl_private_key = 'ssl/private.key'

# server.ssl_certificate_chain = 'ssl/bundle.crt'

# Subscribe this server

server.subscribe()

# Example for a 2nd server (same steps as above):

# Remember to use a different port

# server2 = cherrypy._cpserver.Server()

# server2.socket_host = "0.0.0.0"
```


PDF Generated by PHPKB Knowledge Base Script

server2.socket_port = 8081

server2.thread_pool = 30

server2.subscribe()

Start the server engine (Option 1 *and* 2)

cherrypy.engine.start()

cherrypy.engine.block()

Save and exit again by pressing CTRL+X and confirming with Y.

Finally, this is how your main application deployment directory should look like:

my_app # Main Folder to Contain Everything Together

|

|=== my_app_venv # V. Env. folder with the Python Int.

|=== app # Your application module

|

|--- wsgi.py # File containing application callable

|--- server.py # Python executable to launch the server

|..

|.

Running the server

To start serving your application, you just need to execute server.py using your Python installation.

Run the following to start the server as configured:

Note: **This will execute the script, using the activated Python interpreter because we are still working within the virtual environment. If it is not activated, you will need to state the path:** ***my_app/bin/pythonserver.py.***

python server.py

This will run the server on the foreground. If you would like to stop it, press CTRL+C.

To run the server in the background, run the following:

python server.py &

PDF Generated by PHPKB Knowledge Base Script

To return to the command line, just press enter. The app will still be running.

When you run an application in the background, you will need to use a process manager (e.g. htop) to kill (or stop) it.

Configuring Nginx

After setting up CherryPy to run our application, we now need to do the same with Nginx for it to talk with the CherryPy server(s). For this, we need to modify Nginx's configuration file: "nginx.confâ€•

Run the following command to open up nginx.conf and edit it using nano text editor:

sudo nano /etc/nginx/nginx.conf

Afterwards, you can replace the file with the following example configuration to get Nginx work as a reverse-proxy, talking to your application.

Example configuration for web applications:

worker_processes 1;

events {

worker_connections 1024;

}

http {

sendfile on;

gzip on;

gzip_http_version 1.0;

gzip_proxied any;

gzip_min_length 500;

gzip_disable "MSIE [1-6]\.";

gzip_types text/plain text/xml text/css

text/comma-separated-values

text/javascript

```
application/x-javascript

application/atom+xml;

# Configuration containing list of application servers

upstream app_servers {

server 127.0.0.1:8080;

# server 127.0.0.1:8081;

# ..

# .

}

# Configuration for Nginx

server {

# Running port

listen 80;

# Settings to serve static files

location ^~ /static/ {

# Example:

# root /full/path/to/application/static/file/dir;

root /app/static/;

}

# Serve a static file (ex. favico)

# outside /static directory

location = /favico.ico {
```


PDF Generated by PHPKB Knowledge Base Script

```
root /app/favico.ico;

}

# Proxy connections to the application servers

# app_servers

location / {

proxy_pass http://app_servers;

proxy_redirect off;

proxy_set_header Host $host;

proxy_set_header X-Real-IP $remote_addr;

proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

proxy_set_header X-Forwarded-Host $server_name;

}

}

}
```

When you are done modifying the configuration, press CTRL+X and confirm with Y to save and exit. You will need to restart Nginx for changes to come into effect.

Run the following to restart Nginx:

```
sudo service nginx stop

sudo service nginx start
```