Docker Explained: How To Containerize Python Web Applications

Authored by: ***ASPHostServer Administrator*** *[asphostserver@gmail.com]*
Saved From: http://faq.asphosthelpdesk.com/article.php?id=260

Docker in Brief

The docker project **offers higher-level tools, working together, which are built on top of some Linux kernel features. The goal is to help developers and system administrators port applications - with all of their dependencies conjointly - and get them running across systems and machines -*headache free.***

Docker achieves this by creating safe, LXC (i.e. Linux Containers) based environments for applications called docker containers. These containers are created using docker images, which can be built either by executing commands manually or automatically through Dockerfiles.

Installing Docker on Ubuntu (Latest)

With its most recent release (0.7.1. dating 5 Dec.), docker can be deployed on various Linux operating systems including Ubuntu / Debian and CentOS / RHEL.

We will quickly go over the installation process for Ubuntu (Latest).

Installation Instructions for Ubuntu

Update:

```
sudo aptitude    update
sudo aptitude -y upgrade
```

Make sure aufs support is available:

```
sudo aptitude install linux-image-extra-`uname -r`
```

Add docker repository key to apt-key for package verification:

```
sudo sh -c "wget -qO- https://get.docker.io/gpg | apt-key add -"
```

Add the docker repository to aptitude sources:

```
sudo sh -c "echo deb http://get.docker.io/ubuntu docker main\
> /etc/apt/sources.list.d/docker.list"
```

Update the repository with the new addition:

```
sudo aptitude    update
```

Finally, download and install docker:

```
sudo aptitude install lxc-docker
```

Ubuntu's default firewall (UFW: Uncomplicated Firewall) denies all forwarding traffic by default, which is needed by docker.

Enable forwarding with UFW:

Edit UFW configuration using the nano text editor.

```
sudo nano /etc/default/ufw
```

Scroll down and find the line beginning with DEFAULT_FORWARD_POLICY.

Replace:

```
DEFAULT_FORWARD_POLICY="DROP"
```

With:

```
DEFAULT_FORWARD_POLICY="ACCEPT"
```

Press CTRL+X and approve with Y to save and close.

Finally, reload the UFW:

```
sudo ufw reload
```

Basic Docker Commands

Running the docker daemon and CLI Usage

Upon installation, the docker daemon should be running in the background, ready to accept commands sent by the docker CLI. For certain situation where it might be necessary to manually run docker, use the following.

Running the docker daemon:

```
sudo docker -d &
```

docker CLI Usage:

```
sudo docker [option] [command] [arguments]
```

Note: docker needs sudo privileges in order to work.

Docker Commands

Here is a summary of currently available (version 0.7.1) docker commands:

attach

Attach to a running container

build

Build a container from a Dockerfile

commit

Create a new image from a container's changes

cp

Copy files/folders from the containers filesystem to the host path

diff

Inspect changes on a container's filesystem

events

Get real time events from the server

export

Stream the contents of a container as a tar archive

history

Show the history of an image

images

List images

import

Create a new filesystem image from the contents of a tarball

info

Display system-wide information

insert

Insert a file in an image

inspect

Return low-level information on a container

kill

Kill a running container

load

Load an image from a tar archive

login

Register or Login to the docker registry server

logs

Fetch the logs of a container

port

Lookup the public-facing port which is NAT-ed to PRIVATE_PORT

ps

List containers

pull

Pull an image or a repository from the docker registry server

push

Push an image or a repository to the docker registry server

restart

Restart a running container

rm

Remove one or more containers

rmi

Remove one or more images

run

Run a command in a new container

save

Save an image to a tar archive

search

Search for an image in the docker index

start

Start a stopped container

stop

Stop a running container

tag

Tag an image into a repository

top

Lookup the running processes of a container

version

Show the docker version information

Building a Docker Container To Sandbox Python WSGI Apps

After having installed docker on our server and having quickly gone over its commands, we are ready to start with the actual work to create our docker container running a Python WSGI Application.

Note: **The following section will enable you to have a dockerized (containerized) Python WSGI web application. However, it is definitely not the recommended method *due to its complexity and impracticability. It is here to offer you a chance to learn how to work with a live container and get familiar with the commands we will need to define later in the next section to automate the process.***

Let's begin!

Creating a Base Docker Container From Ubuntu

Using docker's RUN command, we will begin with creating a new container based on the Ubuntu image. We are going to attach a terminal to it using the -t flag and will have bash as the running process.

We are going to expose port 80 so that our application will be accessible from the outside. In future, you might want to load-balance multiple instances and "link" containers to each other to access them using a reverse-proxy running container, for example.

```
sudo docker run -i -t -p 80:80 ubuntu /bin/bash
```

Note: **After executing this command, docker might need to *pull the Ubuntu image before creating a new container for you.***

Remember: You will be attached to the container you create. In order to detach yourself and go back to your main terminal access point, run the escape sequence: CTRL+P followed by CTRL+Q. Being

attached to a docker container is like being connected to a new server instance from inside another.

To attach yourself back to this container:

1. List all running containers using "sudo docker ps"
- Find its ID
- Use "sudo docker attach [id]" to attach back to its terminal

Important:* Please do not forget that since we are *in a container, all the following commands will be executed there, without affecting the host it resides.

Preparing the Base Container for the Installation

In order to deploy Python WSGI web applications inside a container - and the tools we need for the process - the relevant application repositories must be available for the downloads. Unfortunately (and intentionally to keep things simple) this is not the case with the default Ubuntu image that comes with docker.

Let's append Ubuntu's universe repository to the default list of application sources list of the base image.

```
echo "deb http://archive.ubuntu.com/ubuntu/ raring main universe" >>
/etc/apt/sources.list
```

Update the list with the newly added source.

```
apt-get update
```

Before we proceed with setting up Python WSGI applications, there are some tools we should have such as nano, tar, curl, etc. -just in case.

Let's download some useful tools inside our container.

```
apt-get install -y tar \
                   git \
                   curl \
                   nano \
                   wget \
                   dialog \
                   net-tools
                   build-essential
```

Installing Common Python Tools for Deployment

For our tutorial (as an example), we are going to create a very basic Flask application. After following

this article, you can use and deploy your favorite framework instead, the same way you would deploy it on a virtual server.

Remember: All the commands and instructions below still take place inside a container, which acts almost as if it is a brand new server instance of its own.

Let's begin our deployment process with installing Python and pip the Python package manager:

```
# Install pip's dependency: setuptools:
apt-get install -y python python-dev python-distribute python-pip
```

Installing The Web Application and Its Dependencies

Before we begin with creating a sample application, we better make sure that everything - i.e. all the dependencies - are there. First and foremost, you are likely to have your Web Application Framework (WAF) as your application's dependency (i.e. Flask).

As we have pip installed and ready to work, we can use it to pull all the dependencies and have them set up inside our container:

```
# Download and install Flask framework:
pip install flask
```

After installing pip, let's create a basic, sample Flask application inside a "my_application" folder which is to contain everything.

```
# Make a my_application folder
mkdir my_application
# Enter the folder
cd my_application
```

Note: If you are interested in deploying your application instead of this simple-sample example, see the "Quick Tip**" mid-section below.**

Let's create a single, one page flask "Hello World!" application using nano.

```
# Create a sample (app.py) with nano:
nano app.py
```

And copy-and-paste the contents below for this small application we have just mentioned:

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def hello():
    return "Hello World!"
if __name__ == "__main__":
    app.run()
```

Press CTRL+X and approve with Y to save and close.

Alternatively, you can use a "requirements.txt" to contain your application's dependencies such as Flask.

To create a requirements.txt using nano text editor:

```
nano requirements.txt
```

And enter the following inside, alongside all your dependencies:

```
flask
cherrypy
```

Press CTRL+X and approve with Y to save and close.

Our final application folder structure:

```
/my_application
    |
    |- requirements.txt  # File containing list of dependencies
    |- /app              # Application module (which should have your app)
    |- app.py            # WSGI file containing the "app" callable
    |- server.py         # Optional: To run the app servers (CherryPy)
```

Note:Please see the following section regarding the "server.py" -Configuring your Python WSGI Application.

Remember: This application folder will be created inside the container. When you are automatically building images (see the following section on Dockerfiles), you will need to make sure to have this structure on the host, alongside the Dockerfile.

__ * Quick tip for actual deployments * __

How to get your application repository and its requirements inside a container

---

In the above example, we created the application directory inside the container. However, you will not be doing that to deploy your application. You are rather likely to pull its source from a repository.

There are several ways to copy your repository inside a container.

Below explained are two of them:

```
# Example [1]
# Download the application using git:
# Usage: git clone [application repository URL]
# Example:
git clone https://github.com/mitsuhiko/flask/tree/master/examples/flaskr
# Example [2]
# Download the application tarball:
# Usage: wget [application repository tarball URL]
# Example: (make sure to use an actual, working URL)
```

```
wget http://www.github.com/example_usr/application/tarball/v.v.x
# Expand the tarball and extract its contents:
# Usage: tar vxzf [tarball filename .tar (.gz)]
# Example: (make sure to use an actual, working URL)
tar vxzf application.tar.gz
# Download and install your application dependencies with pip.
# Download the requirements.txt (pip freeze output) and use pip to install them
all:
# Usage: curl [URL for requirements.txt] | pip install -r -
# Example: (make sure to use an actual, working URL)
curl http://www.github.com/example_usr/application/requirements.txt | pip
install -r -
```

Configuring your Python WSGI Application

To serve this application, you will need a web server. The web server, which powers the WSGI app, needs to be installed in the same container as the application's other resources. In fact, it will be the process that docker runs.

Note: In this example, we will use CherryPy's built-in production ready HTTP web server due to its simplicity. You can use Gunicorn, CherryPy or even uWSGI (and set them up behind Nginx) by following our tutorials on the subject.

Download and install CherryPy with pip:

```
pip install cherrypy
```

Create a "server.py" to serve the web application from "app.py":

```
nano server.py
```

Copy and paste the contents from below for the server to import your application and start serving it:

```
# Import your application as:
# from app import application
# Example:
from app import app
# Import CherryPy
import cherrypy
if __name__ == '__main__':
    # Mount the application
    cherrypy.tree.graft(app, "/")
    # 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 = 80
    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()
    # Start the server engine (Option 1 *and* 2)
    cherrypy.engine.start()
    cherrypy.engine.block()
```

And that's it! Now you can have a "dockerized" Python web application securely kept in its sandbox, ready to serve thousands and thousands of client requests by simply running:

```
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 &
```

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

To test that everything is running smoothly, which they should given that all the port allocations are already taken care of, you can visit http://[your IP] with your browser to see the "Hello World!**" message.**

Creating the Dockerfile to Automatically Build the Image

As we have mentioned in the previous step, it is certainly not the recommended way to create containers this way for a scalable production deployment. The right way to do can be considered as using Dockerfiles to automate the build process in a structured way.

After having gone through the necessary commands for downloading and installing inside a container, we can use the same knowledge to compose a Dockerfile that docker can use to build an image from, which then can be used to run a Python WSGI application container easily.

Before we start working on the Dockerfile, let's quickly go over the basics.

Dockerfile Basics

Dockerfiles are scripts containing commands declared successively, which are to be executed in that order by docker to automatically create a new docker image. They help greatly with deployments.

These files always begin with defining an base image using the FROM command. From there on, thebuild processstarts and each following action taken forms the final image which will be committed on the host.

Usage:

```
# Build an image using the Dockerfile at current location
# Tag the final image with [name] (e.g. *nginx*)
# Example: sudo docker build -t [name] .
sudo docker build -t nginx_img .
```

Dockerfile Commands Overview

Add

Copy a file from the host into the container

CMD

Set default commands to be executed, or passed to the ENTRYPOINT

ENTRYPOINT

Set the default entrypoint application inside the container

ENV

Set environment variable (e.g. "key = value")

EXPOSE

Expose a port to outside

FROM

Set the base image to use

MAINTAINER

Set the author / owner data of the Dockerfile

RUN

Run a command and commit the ending result (container) image

USER

Set the user to run the containers from the image

VOLUME

Mount a directory from the host to the container

WORKDIR

Set the directory for the directives of CMD to be executed

Creating the Dockerfile

To create a Dockerfile at the current location using the nano text editor, execute the following command:

```
sudo nano Dockerfile
```

Note:Append all the following lines one after the other to form the Dockerfile.

Defining the Fundamentals

Let's begin our Dockerfile by defining the basics (fundamentals) such as the FROM image (i.e. Ubuntu) and the MAINTAINER.

Append the following:

```
##############################################################
# Dockerfile to build Python WSGI Application Containers
# Based on Ubuntu
##############################################################
# Set the base image to Ubuntu
FROM ubuntu
# File Author / Maintainer
MAINTAINER Maintaner Name
```

Updating the Default Application Repository for Installations

---

Run the following to update the`apt-get`repository with additional applications just as we did in the previous section.

Append the following:

```
# Add the application resources URL
RUN echo "deb http://archive.ubuntu.com/ubuntu/ raring main universe" >>
/etc/apt/sources.list
# Update the sources list
RUN apt-get update
```

Installing the Basic Tools

After updating the default application repository sources list, we can begin our deployment process by getting the basic applications we will need.

Append the following:

```
# Install basic applications
apt-get install -y tar git curl nano wget dialog net-tools build-essential
```

Note: **Although you are unlikely to ever need some of the tools above, we are getting them nonetheless *-just-in-case.***

Base Installation Instructions for Python and Basic Python Tools

---

For deploying Python WSGI applications, you are extremely likely to need some of the tools which we worked with before (e.g.pip). Let's install them now before proceeding with setting up the framework (i.e. your WAF) and the your web application server (WAS) of choice.

Append the following:

```
# Install Python and Basic Python Tools
RUN apt-get install -y python python-dev python-distribute python-pip
```

Application Deployment

Given that we are building docker images to deploy Python web applications, we can very all take advantage of docker's ADD command to copy the application repository, preferably with a REQUIREMENTS file to quickly get running in one single step.

Note: To package everything together in a single file and not to repeat ourselves, an application folder, structured similarly to the one below might be a good way to go.

Example application folder structure:

```
/my_application
    |
    |- requirements.txt  # File containing list of dependencies
```

```
    |- /app              # Application module
    |- app.py            # WSGI file containing the "app" callable
    |- server.py         # Optional: To run the app servers (CherryPy)
```

Note: To see about creating this structure, please roll back up and refer to the section Installing The Web Application and Its Dependencies.

Append the following:

```
# Copy the application folder inside the container
ADD /my_application /my_application
```

Note: If you want to deploy from an online host git repository, you can use the following command to clone:

```
RUN git clone [application repository URL]
```

Please do not forget to replace the URL placeholder with your actual one.

Bootstrapping Everything

After adding the instructions for copying the application, let's finish off with final configurations such as pulling the dependencies from the requirements.txt.

```
# Get pip to download and install requirements:
RUN pip install -r /my_application/requirements.txt
# Expose ports
EXPOSE 80
# Set the default directory where CMD will execute
WORKDIR /my_application
# Set the default command to execute
# when creating a new container
# i.e. using CherryPy to serve the application
CMD python server.py
```

Final Dockerfile

In the end, this is what the Dockerfile should look like:

```
################################################################
# Dockerfile to build Python WSGI Application Containers
# Based on Ubuntu
################################################################
# Set the base image to Ubuntu
FROM ubuntu
# File Author / Maintainer
MAINTAINER Maintaner Name
# Add the application resources URL
RUN echo "deb http://archive.ubuntu.com/ubuntu/ raring main universe" >>
```

```
/etc/apt/sources.list
# Update the sources list
RUN apt-get update
# Install basic applications
apt-get install -y tar git curl nano wget dialog net-tools build-essential
# Install Python and Basic Python Tools
RUN apt-get install -y python python-dev python-distribute python-pip
# Copy the application folder inside the container
ADD /my_application /my_application
# Get pip to download and install requirements:
RUN pip install -r /my_application/requirements.txt
# Expose ports
EXPOSE 80
# Set the default directory where CMD will execute
WORKDIR /my_application
# Set the default command to execute
# when creating a new container
# i.e. using CherryPy to serve the application
CMD python server.py
```

Again save and exit the file by pressing CTRL+X and confirming with Y.

Using the Dockerfile to Automatically Build Containers

As we first went over in the "basics" section, Dockerfiles' usage consists of calling them with docker build command.

Since we are instructing docker to copy an application folder (i.e. /my_application) from the current directory, we need to make sure to have it alongside this Dockerfile before starting the build process.

This docker image will allow us to quickly create containers running Python WSGI applications with a single command.

To start using it, build a new container image with the following:

```
sudo docker build -t my_application_img .
```

And using that image - which we tagged my_application_img - we can run a new container running the application with:

```
sudo docker run -name my_application_instance -p 80:80 -i -t my_application_img
```

Now you can visit the IP address of your server instance, and your application will be running via a docker container.

Example:

```
# Usage: Visit http://[my server instance's ip]
http://95.85.10.236/
```

Sample Response:

```
Hello World!
```