**Docker Compose - A gentle introduction with WordPress**

Docker_Icon.png

Overview

[Docker Compose](https://docs.docker.com/compose/overview/) is a orchestration tool for multi-container Docker applications.

We can configure our application's services via a single command to create and start all the services from our configuration using a Compose file. In other words, by using Docker Compose we can orchestrate processes of Docker containers such as starting up, shutting down, and setting up intra-container linking and volumes.

Using Compose is basically a three-step process:

1. Define app's environment with a **Dockerfile** so it can be reproduced anywhere.
2. Define the services that make up an app in **docker-compose.yml** so they can be run together in an isolated environment.
3. Run **docker-compose up** and Compose will start and run the entire app.

Throughout this, we'll work on Ubuntu 14.04.

Docker compose with "Hello World image"

In this section, we'll play with a simple test using Docker Compose just to get the taste of it.

Because we'll launch just one container, the test won't demonstrate one of the main features of Docker Compose: up and down a group of Docker containers all at the same time.

The docker-compose command works on a per-directory basis. So, we just make one directory for each container and one **docker-compose.yml** file for each container inside its directory. In that way, we can have multiple groups of Docker containers running on one machine.

We'll be working on a virtual environment (python 2.7):

$ virtualenv venv

$ source venv/bin/activate

(venv) $

Install python-pip:

$ sudo apt-get -y install python-pip

Install Docker Compose:

$ sudo pip install docker-compose

**~/hello-world/docker-compose.yml**:

my-test:

image: hello-world

The first line is the container name while the second line specifies the container image which will be downloaded from the Docker Hub repository.

$ docker-compose up

Creating helloworld\_my-test\_1

Attaching to helloworld\_my-test\_1

my-test\_1 |

my-test\_1 | Hello from Docker.

my-test\_1 | This message shows that your installation appears to be working correctly.

...

The output then explains what Docker is doing with Docker Hub, Docker daemon, and Docker client:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the executable that produces the output above.
4. The Docker daemon streamed that output to the Docker client, which sent it to our terminal.

Wordpress and MariaDB with Docker Compose

In the following sections, we'll deal with a real-world example of using Docker Compose to install an WordPress application. We'll install MariaDB as well.

Since WordPress app normally runs on a LAMP stack, we need Linux, Apache, MySQL/MariaDB, and PHP.

The official WordPress Docker image provides Apache and PHP for us but not MariaDB which we need to take care of by ourselves.

Install Wordpress

We need to tell Docker Compose to start a new container called **wordpress** and download the **wordpress** image from the Docker Hub.

**~/wordpress/docker-compose.yml**:

wordpress:

image: wordpress

To bring up the image:

(venv)~/wordpress$ docker-compose up

At the end of the download process, we get complains like this:

...

wordpress\_1 | error: missing required WORDPRESS\_DB\_PASSWORD environment variable

wordpress\_1 | Did you forget to -e WORDPRESS\_DB\_PASSWORD=... ?

wordpress\_1 |

wordpress\_1 | (Also of interest might be WORDPRESS\_DB\_USER and WORDPRESS\_DB\_NAME.)

wordpress\_wordpress\_1 exited with code 1

It tells us our wordpress needs db.

Install MariaDB

We should define a new container called **wordpress\_db** and tell Docker to use the **mariadb** image from the Docker Hub.

To add the MariaDB image to the group, let's modify our **~/wordpress/docker-compose.yml** :

wordpress:

image: wordpress

links:

- wordpress\_db:mysql

wordpress\_db:

image: mariadb

With the modification, we'll have two containers: **wordpress** container and **wordpress\_db** container.

We're telling our **wordpress** container to link our **wordpress\_db** container into the **wordpress** container and call it **mysql**.

So, inside the **wordpress** container the hostname **mysql** will be forwarded to our **wordpress\_db** container.

Let's run **docker-compose up** again:

(venv)~/wordpress$ docker-compose up

...

ERROR: for wordpress Cannot start service wordpress: Cannot start container d14451708c365e5c0862194e6585620569f06caaf0fc8c226a0a44a332adf70a: Cannot link to a non running container: /wordpress\_wordpress\_db\_1 AS /wordpress\_wordpress\_1/mysql

ERROR: Encountered errors while bringing up the project.

Still we see that we're not quite there yet though. More work needs to be done.

It looks like just linking the two containers isn't quite enough: we need to set the MYSQL\_ROOT\_PASSWORD variable.

wordpress:

image: wordpress

links:

- wordpress\_db:mysql

ports:

- 8888:80

wordpress\_db:

image: mariadb

environment:

MYSQL\_ROOT\_PASSWORD: mysql-root-password

In 'port' section, the first port number is the host port number, and the second port number is the container port. The port configuration forwards requests on port 8888 of the host to the default web server port 80 inside the container.

(venv)~/wordpress$ docker-compose up -d

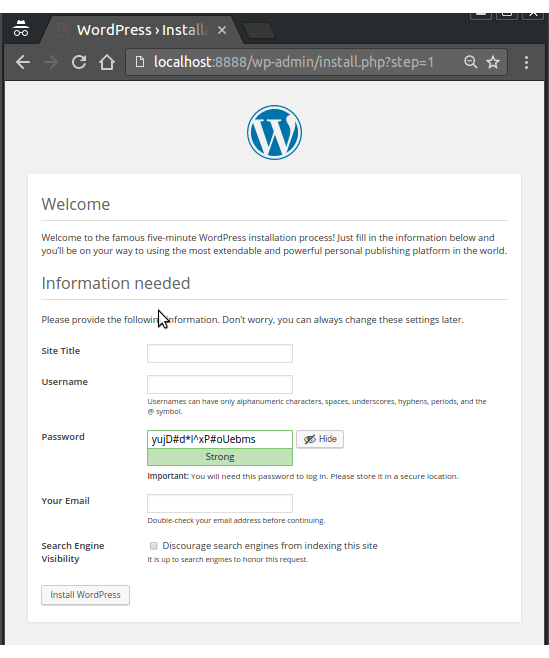
Starting wordpress\_wordpress\_db\_1

Starting wordpress\_wordpress\_1

(venv)~/wordpress$

This time, we run it with the **-d** option, which will tell **docker-compose** to run the containers in the background as a service.

Open up a web browser (**localhost:8888**):



Stop the containers

Stop all the Docker containers with the following command:

$ docker-compose stop

Docker-compose using bridge network

Instead of using the legacy **link** for communications between containers, we can use **networks** as shown in [Docker Networks - Bridge](https://www.bogotobogo.com/DevOps/Docker/Docker-Networks.php).

