How To Install and Use Docker on CentOS 7

Posted May 24, 2016

Top of Form

Bottom of Form

1

 3.6kviews [Docker](https://www.digitalocean.com/community/tags/docker?type=tutorials) [CentOS](https://www.digitalocean.com/community/tags/centos?type=tutorials)

**Introduction**

Docker is an application that makes it simple and easy to run application processes in a container, which are like virtual machines, only more portable, more resource-friendly, and more dependent on the host operating system. For a detailed introduction to the different components of a Docker container, check out[The Docker Ecosystem: An Introduction to Common Components](https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-an-introduction-to-common-components).

There are two methods for installing Docker on CentOS 7. One method involves installing it on an existing installation of the operating system. The other involves spinning up a server with a tool called [Docker Machine](https://www.digitalocean.com/community/tutorials/how-to-provision-and-manage-remote-docker-hosts-with-docker-machine-on-centos-7) that auto-installs Docker on it.

In this tutorial, you'll learn how to install and use it on an existing installation of CentOS 7.

Prerequisites

* 64-bit CentOS 7 Droplet
* Non-root user with sudo privileges. A CentOS 7 server set up using [Initial Setup Guide for CentOS 7](https://www.digitalocean.com/community/tutorials/initial-server-setup-with-centos-7)explains how to set this up.

**Note:** Docker requires a 64-bit version of CentOS 7 as well as a kernel version equal to or greater than 3.10. The default 64-bit CentOS 7 Droplet meets these requirements.

All the commands in this tutorial should be run as a non-root user. If root access is required for the command, it will be preceded by sudo. [Initial Setup Guide for CentOS 7](https://www.digitalocean.com/community/tutorials/initial-server-setup-with-centos-7) explains how to add users and give them sudo access.

Step 1 — Installing Docker

The Docker installation package available in the official CentOS 7 repository may not be the latest version. To get the latest and greatest version, install Docker from the official Docker repository. This section shows you how to do just that.

But first, let's update the package database:

* sudo yum check-update

Now run this command. It will add the official Docker repository, download the latest version of Docker, and install it:

* curl -fsSL https://get.docker.com/ | sh

After installation has completed, start the Docker daemon:

* sudo systemctl start docker

Verify that it's running:

* sudo systemctl status docker

The output should be similar to the following, showing that the service is active and running:

Output

● docker.service - Docker Application Container Engine

Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)

Active: active (running) since Sun 2016-05-01 06:53:52 CDT; 1 weeks 3 days ago

Docs: https://docs.docker.com

Main PID: 749 (docker)

Lastly, make sure it starts at every server reboot:

* sudo systemctl enable docker

Installing Docker now gives you not just the Docker service (daemon) but also the docker command line utility, or the Docker client. We'll explore how to use the docker command later in this tutorial.

Step 2 — Executing Docker Command Without Sudo (Optional)

By default, running the docker command requires root privileges — that is, you have to prefix the command with sudo. It can also be run by a user in the **docker** group, which is automatically created during the installation of Docker. If you attempt to run the docker command without prefixing it with sudoor without being in the docker group, you'll get an output like this:

Output

docker: Cannot connect to the Docker daemon. Is the docker daemon running on this host?.

See 'docker run --help'.

If you want to avoid typing sudo whenever you run the docker command, add your username to the docker group:

* sudo usermod -aG docker $(whoami)

You will need to log out of the Droplet and back in as the same user to enable this change.

If you need to add a user to the docker group that you're not logged in as, declare that username explicitly using:

* sudo usermod -aG docker username

The rest of this article assumes you are running the docker command as a user in the docker user group. If you choose not to, please prepend the commands with sudo.

Step 3 — Using the Docker Command

With Docker installed and working, now's the time to become familiar with the command line utility. Usingdocker consists of passing it a chain of options and subcommands followed by arguments. The syntax takes this form:

* docker [option] [command] [arguments]

To view all available subcommands, type:

* docker

As of Docker 1.11.1, the complete list of available subcommands includes:

Output

attach Attach to a running container

build Build an image from a Dockerfile

commit Create a new image from a container's changes

cp Copy files/folders between a container and the local filesystem

create Create a new container

diff Inspect changes on a container's filesystem

events Get real time events from the server

exec Run a command in a running container

export Export a container's filesystem as a tar archive

history Show the history of an image

images List images

import Import the contents from a tarball to create a filesystem image

info Display system-wide information

inspect Return low-level information on a container or image

kill Kill a running container

load Load an image from a tar archive or STDIN

login Log in to a Docker registry

logout Log out from a Docker registry

logs Fetch the logs of a container

network Manage Docker networks

pause Pause all processes within a container

port List port mappings or a specific mapping for the CONTAINER

ps List containers

pull Pull an image or a repository from a registry

push Push an image or a repository to a registry

rename Rename a container

restart Restart a container

rm Remove one or more containers

rmi Remove one or more images

run Run a command in a new container

save Save one or more images to a tar archive

search Search the Docker Hub for images

start Start one or more stopped containers

stats Display a live stream of container(s) resource usage statistics

stop Stop a running container

tag Tag an image into a repository

top Display the running processes of a container

unpause Unpause all processes within a container

update Update configuration of one or more containers

version Show the Docker version information

volume Manage Docker volumes

wait Block until a container stops, then print its exit code

To view the switches available to a specific command, type:

* docker docker-subcommand --help

To view system-wide information, use:

* docker info

Step 4 — Working with Docker Images

Docker containers are run from Docker images. By default, it pulls these images from Docker Hub, a Docker registry managed by Docker, the company behind the Docker project. Anybody can build and host their Docker images on Docker Hub, so most applications and Linux distributions you'll need to run Docker containers have images that are hosted on Docker Hub.

To check whether you can access and download images from Docker Hub, type:

* docker run hello-world

The output, which should include the following, should indicate that Docker in working correctly:

Output

Hello from Docker.

This message shows that your installation appears to be working correctly.

...

You can search for images available on Docker Hub by using the docker command with the searchsubcommand. For example, to search for the CentOS image, type:

* docker search centos

The script will crawl Docker Hub and return a listing of all images whose name match the search string. In this case, the output will be similar to this:

Output

NAME DESCRIPTION STARS OFFICIAL AUTOMATED

centos The official build of CentOS. 2224 [OK]

jdeathe/centos-ssh CentOS-6 6.7 x86\_64 / CentOS-7 7.2.1511 x8... 22 [OK]

jdeathe/centos-ssh-apache-php CentOS-6 6.7 x86\_64 / Apache / PHP / PHP M... 17 [OK]

million12/centos-supervisor Base CentOS-7 with supervisord launcher, h... 11 [OK]

nimmis/java-centos This is docker images of CentOS 7 with dif... 10 [OK]

torusware/speedus-centos Always updated official CentOS docker imag... 8 [OK]

nickistre/centos-lamp LAMP on centos setup 3 [OK]

...

In the **OFFICIAL** column, **OK** indicates an image built and supported by the company behind the project. Once you've identifed the image that you would like to use, you can download it to your computer using the pull subcommand, like so:

* docker pull centos

After an image has been downloaded, you may then run a container using the downloaded image with therun subcommand. If an image has not been downloaded when docker is executed with the runsubcommand, the Docker client will first download the image, then run a container using it:

* docker run centos

To see the images that have been downloaded to your computer, type:

* docker images

The output should look similar to the following:

[secondary\_lable Output]

REPOSITORY TAG IMAGE ID CREATED SIZE

centos latest 778a53015523 5 weeks ago 196.7 MB

hello-world latest 94df4f0ce8a4 2 weeks ago 967 B

As you'll see later in this tutorial, images that you use to run containers can be modified and used to generate new images, which may then be uploaded (*pushed* is the technical term) to Docker Hub or other Docker registries.

Step 5 — Running a Docker Container

The hello-world container you ran in the previous step is an example of a container that runs and exits, after emitting a test message. Containers, however, can be much more useful than that, and they can be interactive. After all, they are similar to virtual machines, only more resource-friendly.

As an example, let's run a container using the latest image of CentOS. The combination of the **-i** and **-t**switches gives you interactive shell access into the container:

* docker run -it centos

Your command prompt should change to reflect the fact that you're now working inside the container and should take this form:

Output

[root@59839a1b7de2 /]#

**Important:** Note the container id in the command prompt. In the above example, it is 59839a1b7de2.

Now you may run any command inside the container. For example, let's install MariaDB server in the running container. No need to prefix any command with sudo, because you're operating inside the container with root privileges:

* yum install mariadb-server

Step 6 — Committing Changes in a Container to a Docker Image

Docker filesystems are temporary by default. If you start up a Docker image, you can create, modify, and delete files just like a virtual machine. However, if you stop the container and start it up again, all your changes will be lost: any files you previously deleted will now be back, and any new files or edits you made won't be present. This is because Docker images are more like templates than like images in the standard virtualization world.

To learn how to retain these changes **within the container** so that they are persistent through restarts of the container, you need to use Docker Data Volumes. Refer to [How To Work with Docker Data Volumes on Ubuntu 14.04](https://www.digitalocean.com/community/tutorials/how-to-work-with-docker-data-volumes-on-ubuntu-14-04) for details — the prerequisites to this article include Ubuntu 14.04, but the content about Docker Data Volumes applies to CentOS as well.

This section shows you how to save the state of a container as a new Docker image.

After installing MariaDB server inside the CentOS container, you now have a container running off an image, but the container is different from the image you used to create it.

To save the state of the container as a new image, first exit from it:

* exit

Then commit the changes to a new Docker image instance using the following command. The **-m** switch is for the commit message that helps you and others know what changes you made, while **-a** is used to specify the author. The container ID is the one you noted earlier in the tutorial when you started the interactive docker session. Unless you created additional repositories on Docker Hub, the repository is usually your Docker Hub username:

* docker commit -m "What did you do to the image" -a "Author Name" container-id repository/new\_image\_name

For example:

* docker commit -m "added mariadb-server" -a "Sunday Ogwu-Chinuwa" 59839a1b7de2 finid/centos-mariadb

**Note:** When you *commit* an image, the new image is saved locally, that is, on your computer. Later in this tutorial, you'll learn how to push an image to a Docker registry like Docker Hub so that it may be assessed and used by you and others.

After that operation has completed, listing the Docker images now on your computer should show the new image, as well as the old one that it was derived from:

* docker images

The output should be of this sort:

Output

REPOSITORY TAG IMAGE ID CREATED SIZE

finid/centos-mariadb latest 23390430ec73 6 seconds ago 424.6 MB

centos latest 778a53015523 5 weeks ago 196.7 MB

hello-world latest 94df4f0ce8a4 2 weeks ago 967 B

In the above example, **centos-mariadb** is the new image, which was derived from the existing CentOS image from Docker Hub. The size difference reflects the changes that were made. And in this example, the change was that MariaDB server was installed. So next time you need to run a container using CentOS with MariaDB server pre-installed, you can just use the new image. Images may also be built from what's called a Dockerfile. But that's a very involved process that's well outside the scope of this article. We'll explore that in a future article.

Step 7 — Listing Docker Containers

After using Docker for a while, you'll have many active (running) and inactive containers on your computer. To view the active ones, use:

* docker ps

You will see output similar to the following:

Output

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

f7c79cc556dd centos "/bin/bash" 3 hours ago Up 3 hours silly\_spence

To view all containers — active and inactive, pass it the -a switch:

* docker ps -a

To view the latest container you created, pass it the -l switch:

* docker ps -l

Stopping a running or active container is as simple as typing:

* docker stop container-id

The container-id can be found in the output from the docker ps command.

Step 8 — Pushing Docker Images to a Docker Repository

The next logical step after creating a new image from an existing image is to share it with a select few of your friends, the whole world on Docker Hub, or other Docker registry that you have access to. To push an image to Docker Hub or any other Docker registry, you must have an account there.

This section shows you how to push a Docker image to Docker Hub.

To create an account on Docker Hub, register at [Docker Hub](https://hub.docker.com/). Afterwards, to push your image, first log into Docker Hub. You'll be prompted to authenticate:

* docker login -u docker-registry-username

If you specified the correct password, authentication should succeed. Then you may push your own image using:

* docker push docker-registry-username/docker-image-name

It will take sometime to complete, and when completed, the output will be of this sort:

Output

The push refers to a repository [docker.io/finid/centos-mariadb]

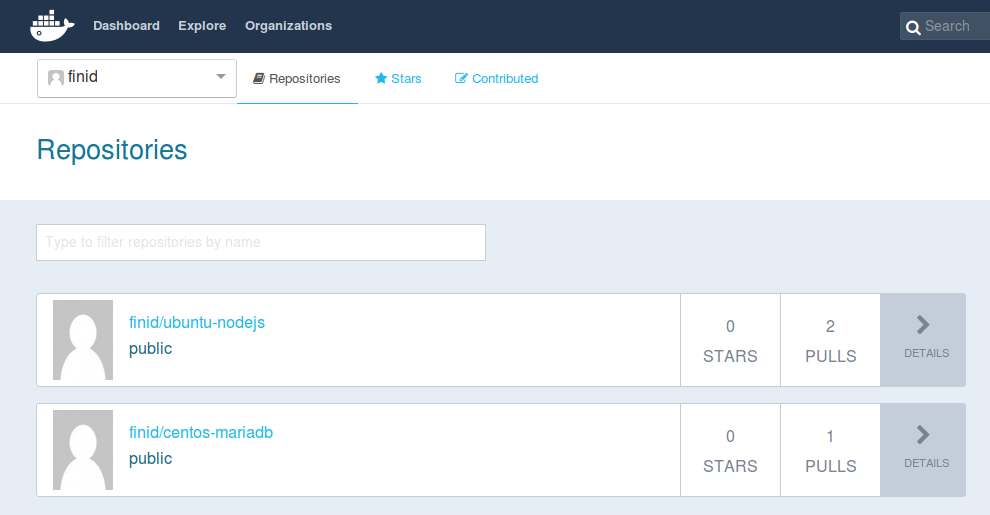
670194edfaf5: Pushed

5f70bf18a086: Mounted from library/centos

6a6c96337be1: Mounted from library/centos

...

After pushing an image to a registry, it should be listed on your account's dashboard, like that show in the image below.



If a push attempt results in an error of this sort, then you likely did not log in:

Output

The push refers to a repository [docker.io/finid/centos-mariadb]

e3fbbfb44187: Preparing

5f70bf18a086: Preparing

a3b5c80a4eba: Preparing

7f18b442972b: Preparing

3ce512daaf78: Preparing

7aae4540b42d: Waiting

unauthorized: authentication required

Log in, then repeat the push attempt.

Conclusion

There's a whole lot more to Docker than has been given in this article, but this should be enough to getting you started working with it on CentOS 7. Like most open source projects, Docker is built from a fast-developing codebase, so make a habit of visiting the project's [blog page](https://blog.docker.com/) for the latest information.

Also check out the [other Docker tutorials](https://www.digitalocean.com/community/tags/docker?type=tutorials) in the DO Community.

How To Provision and Manage Remote Docker Hosts with Docker Machine on CentOS 7

Posted May 24, 2016

Top of Form

Bottom of Form

2

 3.6kviews [Docker](https://www.digitalocean.com/community/tags/docker?type=tutorials) [CentOS](https://www.digitalocean.com/community/tags/centos?type=tutorials)

**Introduction**

Docker Machine is a tool that makes it easy to provision and manage multiple Docker hosts remotely from your personal computer. Such servers are commonly referred to as Dockerized hosts, and as a matter of course, can be used to run Docker containers.

While Docker Machine can be installed on a local or a remote system, the most common approach is to install it on your local computer (native installation or virtual machine) and use it to provision Dockerized remote servers.

Though Docker Machine can be installed on most Linux distribution as well as on Mac OS X and Windows, in this tutorial, we'll install it on your local machine running CentOS 7 and use it to provision Dockerized DigitalOcean Droplets.

Prerequisites

To follow this tutorial, you will need the following:

* A local machine running CentOS 7 with Docker installed. (see [How To Install and Use Docker on CentOS 7](https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-centos-7) for instructions)
* A DigitalOcean API token. If you don't have one, generate it using [this guide](https://www.digitalocean.com/community/tutorials/how-to-use-the-digitalocean-api-v2). When you generate a token, be sure that it has read-write scope. That is the default, so if you do not change any option while generating it, it will have read-write capabilities. And to make it easier to use on the command line, be sure to assign the token to a variable as given in that article.

Step 1 — Installing Docker Machine on Your Local Computer

n this step, we'll work through the process of installing Docker Machine on your local computer running CentOS 7.

To download and install the binary, type:

* wget https://github.com/docker/machine/releases/download/v0.6.0/docker-machine-$(uname -s)-$(uname -m)

The name of the file should be docker-machine-Linux-x86\_64. Rename it to docker-machine to make it easier to work with:

* mv docker-machine-Linux-x86\_64 docker-machine

Make it executable:

chmod +x docker-machine

Move or copy it to the usr/local/bin directory so that it will be available as a system command.

sudo mv docker-machine /usr/local/bin

Check the version, which will indicate that it's properly installed:

* docker-machine version

The output should be similar to

Output

docker-machine version 0.6.0, build e27fb87

Step 2 — Installing Additional Docker Machine Scripts

There are three bash scripts in the Docker Machine GitHub repository designed to facilitate the usage of the docker and docker-machine commands. They provide command completion and bash-prompt customization.

In this step, we'll install the three scripts on your local machine. They will be downloaded and installed into the /etc/bash\_completion.d directory.

The first script makes it possible to see the active machine in your bash prompt. This comes in handy when you working with and switching between multiple Dockerized machines. The script is calleddocker-machine-prompt.bash. To download it, type:

* sudo wget https://raw.githubusercontent.com/docker/machine/master/contrib/completion/bash/docker-machine-prompt.bash -O /etc/bash\_completion.d/docker-machine-prompt.bash

To complete the installation of the above file, you'll have to set a custom value for the PS1 variable in your.bashrc file. So open it using nano (PS1 is a special shell variable used to modify the bash command prompt):

* nano ~/.bashrc

Copy and paste the following line at the end of that file:

~/.bashrc

export PS1='[\u@\h \W$(\_\_docker\_machine\_ps1 " [%s]")]\$ '

With this modification to your ~/.bashrc file, when you activate a machine, its name will be inserted into the shell prompt.

Save and close the file.

The second script is called docker-machine-wrapper.bash. It adds a use subcommand to the docker-machine command, making it easy to switch between Dockerized Machines. To download it, type

* sudo wget https://raw.githubusercontent.com/docker/machine/master/contrib/completion/bash/docker-machine-wrapper.bash -O /etc/bash\_completion.d/docker-machine-wrapper.bash

The third script is called docker-machine.bash. It adds bash completion for docker-machine commands. Download it using:

* sudo wget https://raw.githubusercontent.com/docker/machine/master/contrib/completion/bash/docker-machine.bash -O /etc/bash\_completion.d/docker-machine.bash

To apply the changes you've made so far, close, then reopen your terminal. If you're logged into the machine via SSH, exit the session and log in again. Command completion for the docker and docker-machine commands should now be working. If it does not work, it's likely that the bash-completionpackage is not installed. If that's the case, install it using:

* sudo yum install bash-completion

That should fix it.

Step 3 — Provisioning a Dockerized Host Using Docker Machine

Now that you have Docker and Docker Machine running on your local machine, you can now provision a Dockerized Droplet on your DigitalOcean account using Docker Machine's docker-machine createcommand. If you've not done so already, assign your DigitalOcean API token to a bash variable using:

* export shell-variable=your-api-token

**NOTE:** This tutorial uses DOTOKEN as the bash variable for the DO API token. The variable name does not have to be DOTOKEN, and it does not have to be in all caps.

To make the variable permanent, put it in your ~/.bashrc file. This step is optional, but it is necessary if you want to the value to persist across terminal sessions.

* nano ~/.bashrc

Add a line similar to this anywhere:

~/.bashrc

export DOTOKEN=your-api-token

To activate the variable in the current terminal session, type:

* source ~/.bashrc

To call the docker-machine create command successfully you must specify (at a minimum) the driver, the API token (or the variable that evaluates to it), and a unique name for the remote host. To create your first Dockerized host, type:

* docker-machine create --driver digitalocean --digitalocean-access-token $DOTOKEN machine-name

Partial output as the host is being created follows. In this output, the name of the host is centos-docker:

Output

...

Installing Docker...

Copying certs to the local machine directory...

Copying certs to the remote machine...

Setting Docker configuration on the remote daemon...

Checking connection to Docker...

Docker i up and running!

To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run: docker-machine env centos-docker

An SSH key pair is created for the new host so that docker-machine can access it remotely. The Droplet is provisioned with the desired operating system, and Docker is installed on the system. When the command is complete, your Docker Droplet is up and running.

To see the newly create host from the command line, type:

* docker-machine ls

The output should be similar to this:

Output

NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS

centos-docker - digitalocean Running tcp://104.236.120.71:2376 v1.11.1

Step 4 — Specify the Base OS When Creating a Dockerized Host

This step shows you how to specify a base OS and version for the Dockerized Host being created with the--digitalocean-image flag.

For example, to create a Dockerized host using CentOS 7, type:

* docker-machine create --driver digitalocean --digitalocean-image centos-7-x64 --digitalocean-access-token $DOTOKEN machine-name

What if you would like to run Debian 8 on your server? You would type the following command:

* docker-machine create --driver digitalocean --digitalocean-image debian-8-x64 --digitalocean-access-token $DOTOKEN machine-name

By default, the base operating system used when creating a Dockerized host with Docker Machine is*supposed* to be the latest version (or the latest LTS version for Ubuntu). However, at the time of this publication, the docker-machine create command is still using Ubuntu 15.10 as the base operating system, even though Ubuntu 16.04 is the latest LTS edition. So if you need to run Ubuntu 16.04, you'll have to specify Ubuntu along with the desired version by passing the --digitalocean-image flag to thedocker-machine create command:

* docker-machine create --driver digitalocean --digitalocean-image ubuntu-16-04-x64 --digitalocean-access-token $DOTOKEN machine-name

The base operating system is not the only choice you have. You can also specify the size of the Droplet. by default, it is the smallest Droplet, which has 512 MB of RAM and a 20 GB SSD. This next example provisions a host with 1 GB of RAM:

* docker-machine create --driver digitalocean --digitalocean-size 1gb --digitalocean-access-token $DOTOKEN machine-name

To see all the flags specific to creating a Docker Machine using the DigitalOcean driver, type:

* docker-machine create --driver digitalocean -h

**Tip:** If you refresh the Droplet page of your DigitalOcean dashboard, you should see the new hosts you created using the docker-machine command.

Step 5 — Executing Other Docker Machine Commands

You've seen how to provision a Dockerized host using the create subcommand. You also seen how to list the hosts available to Docker Machine using the ls subcommand. In this step, you'll learn about a few more of the other docker-machine subcommands.

To obtain detailed information about a Dockerized host, use the inspect subcommand, like so:

* docker-machine inspect machine-name

The output should include lines like these. The **Image** line reveals the version of the Linux distribution used:

Output

...

"DropletName": "",

"Image": "centos-7-0-x64",

"Region": "nyc3",

"SSHKeyID": 1912961,

"Size": "512mb",

"IPv6": false,

"Backups": false,

"PrivateNetworking": false,

---

To print the connection configuration for a host, type:

* docker-machine config machine-name

The output should be similar to this:

Output

--tlsverify

--tlscacert="/home/kamit/.docker/machine/certs/ca.pem"

--tlscert="/home/kamit/.docker/machine/certs/cert.pem"

--tlskey="/home/kamit/.docker/machine/certs/key.pem"

-H=tcp://45.55.183.145:2376

The last line in the output of the docker-machine config command reveals the IP address of the host, but you can also get that piece of information by typing:

* docker-machine ip machine-name

If you need to power down a remote host, you can use docker-machine to stop it:

* docker-machine stop machine-name

Verify that it is stopped.

* docker-machine ls

The status of the host has changed:

Ouput

NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS

centos-docker digitalocean Timeout

To start it again:

* docker-machine start machine-name

Verify that it is started:

* docker-machine ls

You will see that the STATE is now set Running for the host:

Ouput

NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS

centos-docker - digitalocean Running tcp://159.203.117.16:2376 v1.11.1

Finally, if you want to remove a host:

* docker-machine rm machine-name

Step 6 — Executing Commands on a Dockerized Host via SSH

At this point, you've been getting information about your hosts, but you can do more than that. For example, you can execute native Linux commands on a Docker host by using the ssh subcommand ofdocker-machine from your local system. This section explains how to perform ssh commands viadocker-machine as well as how to open an SSH session to a Dockerized host.

Assuming that you've provisioned a host with CentOS as the operating system, execute the following command from your local system to update the packages on the Docker host:

* docker-machine ssh machine-name yum update

Not sure what kernel your remote Docker host is using? Type the following:

* docker-machine ssh machine-name uname -r

Besides using the ssh subcommand to execute commands on the remote Docker host, you can also use it to log into the Dockerized host itself. That's as easy as typing:

* docker-machine ssh machine-name

Your command prompt will change to reflect the fact that you're logged into the remote host:

root@machine-name#

To exit from the remote host, simply type:

exit

Step 7 — Activating a Dockerized Host

Activating a Docker host connects your local Docker client to that system, which makes it possible to run normal docker commands on the remote system. To activate a Docker host, type the following command:

* eval $(docker-machine env machine-name)

Alternatively, you can activate it by using this command:

* docker-machine use machine-name

**Tip** When working with multiple Docker hosts, the docker-machine use command is the easiest method of switching from one to the other.

After typing any of the above commands, your bash prompt should change to indicate that your Docker client is pointing to the remote Docker host. It will take this form. The name of the host will be at the end of the prompt:

username@localmachine:~ [machine-name]$

Now any docker command you type at this command prompt will be executed on that remote host.

If a host is active on the terminal that the docker-machine ls command is run, the asterisk under the**ACTIVE** column shows that it is the active one.

Output

NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS

centos-docker \* digitalocean Running tcp://45.55.183.145:2376 v1.11.1

**NOTE:** When a host is active, any docker command you type on the terminal you're using will be executed on the remote host. However, all normal Linux commands are executed on the local computer.

To exit from the remote Docker host, type the following:

docker-machine use -u

You will be returned to the prompt for your local system.

Step 8 — Creating Docker Containers on a Remote Dockerized Host

So far, you have provisioned a Dockerized Droplet on your DigitalOcean account and you've activated it — that is, your Docker client is pointing to it. The next logical step is to spin up containers on it. As an example, let's try running the official Nginx container.

While still pointing to your Docker host (as indicated by the changed prompt), execute this command to run an Nginx container:

docker run -d -p 8080:80 --name httpserver nginx

In this command, we're mapping port 80 in the Nginx container to port 8080 on the Dockerized host so that we can access the default Nginx page from anywhere.

If the command executed successfully, you will be able to access the default Nginx page by pointing your Web browser to http://machine-ip:8080.

While the Docker host is still activated (as seen by its name in the prompt), you should be able to list the images on that host:

docker images

The output should include the Nginx image you just used, plus others you downloaded before:

Output

REPOSITORY TAG IMAGE ID CREATED SIZE

nginx latest 3edcc5de5a79 14 minutes ago 182.7 MB

mariadb latest f7ef3a80dc89 2 days ago 382.2 MB

hello-world latest 94df4f0ce8a4 3 days ago 967 B

You should also be able to list the active or running containers on the host:

docker ps

If the Nginx container you ran in this step is the only active container, the output should look like this:

Output

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

c30e8c1bb03f nginx "nginx -g 'daemon off" 14 minutes ago Up 14 minutes 443/tcp, 0.0.0.0:8080->80/tcp httpserver

To exit the prompt for the remote host, type exit. This will close the terminal as well:

exit

**Tip:** If you intend to create containers on a remote host, your Docker client *must* be pointing to it — that is, it must be the active host in the terminal that you're using. Otherwise you'll be creating the container on your local machine. Let your command prompt be your guide.

Step 9 — Disabling Crash Reporting (Optional)

By default, whenever an attempt to provision a Dockerized host using Docker Machine fails, or Docker Machine crashes, some diagnostic information is sent automatically to a Docker account on Bugsnag. If you're not comfortable with this, you can disable the reporting by creating an empty file called no-error-report under your installations .docker/machine directory.

To accomplish that, simply type:

* touch ~/.docker/machine/no-error-report

Check the file for error messages if provisioning fails or Docker Machine crashes.

Conclusion

This has been an introduction to installing and using Docker Machine to provision multiple Docker Droplets remotely from one local system. Now you should be able to quickly provision as many Dockerized hosts on your DigitalOcean account as you need.

For more on Docker Machines, visit the [official documentation page](https://docs.docker.com/machine/overview/). The three bash scripts downloaded in this tutorial are hosted on [this GitHub page](https://github.com/docker/machine/tree/master/contrib/completion/bash).

How To Install and Use Docker Compose on Ubuntu 14.04

Posted Nov 19, 2015

Top of Form

Bottom of Form

13

 72.2kviews [Docker](https://www.digitalocean.com/community/tags/docker?type=tutorials) [Ubuntu](https://www.digitalocean.com/community/tags/ubuntu?type=tutorials)

**Introduction**

[Docker](https://docs.docker.com/) is a great tool, but to really take full advantage of its potential it's best if each component of your application runs in its own container. For complex applications with a lot of components, orchestrating all the containers to start up and shut down together (not to mention talk to each other) can quickly become unwieldy.

The Docker community came up with a popular solution called [Fig](http://www.fig.sh/), which allowed you to use a single YAML file to orchestrate all your Docker containers and configurations. This became so popular that the Docker team eventually decided to make their own version based on the Fig source. They called it *Docker Compose*. In short, it makes dealing with the orchestration processes of Docker containers (such as starting up, shutting down, and setting up intra-container linking and volumes) really easy.

By the end of this article, you will have Docker and Docker Compose installed and have a basic understanding of how Docker Compose works.

Docker and Docker Compose Concepts

Using Docker Compose requires a combination of a bunch of different Docker concepts in one, so before we get started let's take a minute to review the various concepts involved. If you're already familiar with Docker concepts like volumes, links, and port forwarding then you might want to go ahead and skip on to the next section.

**Docker Images**

Each Docker container is a local instance of a Docker image. You can think of a Docker image as a complete Linux installation. Usually a minimal installation contains only the bare minimum of packages needed to run the image. These images use the kernel of the host system, but since they are running inside a Docker container and only see their own file system, it's perfectly possible to run a distribution like CentOS on an Ubuntu host (or vice-versa).

Most Docker images are distributed via the [Docker Hub](https://hub.docker.com/), which is maintained by the Docker team. Most popular open source projects have a corresponding image uploaded to the Docker Registry, which you can use to deploy the software. When possible it's best to grab "official" images, since they are guaranteed by the Docker team to follow Docker best practices.

**Communication Between Docker Images**

Docker containers are isolated from the host machine by default, meaning that by default the host machine has no access to the file system inside the Docker container, nor any means of communicating with it via the network. Needless to say, this makes configuring and working with the image running inside a Docker container difficult by default.

Docker has three primary ways to work around this. The first and most common is to have Docker specify environment variables that will be set inside the Docker container. The code running inside the Docker container will then check the values of these environment variables on startup and use them to configure itself properly.

Another commonly used method is a [Docker data volume](https://www.digitalocean.com/community/tutorials/how-to-work-with-docker-data-volumes-on-ubuntu-14-04). Docker volumes come in two flavors — internal and shared. Docker containers run ephemerally by default, which means that every time the container is shut down or restarted it doesn't save its data — it essentially reverts to the state it was in when the container started. This is good for testing and development since you're guaranteed to be working with the same environment on each run of your Docker container but not so good if you were hoping the blog posts you happily typed into your WordPress install would still be there next time you restart Docker. (Luckily the creators of the official Docker WordPress and MariaDB images have taken care of this issue for us. More on that in the [Deploying Wordpress and PHPMyAdmin with Docker Compose on Ubuntu 14.04](https://www.digitalocean.com/community/tutorials/how-to-install-wordpress-and-phpmyadmin-with-docker-compose-on-ubuntu-14-04)article.)

Specifying an internal volume just means that for a folder you specify for a particular Docker container, the data will be persisted between restarts. For example if you wanted to make sure your log files hung around between each restart you might specify an internal /var/log volume.

A shared volume maps a folder inside a Docker container onto a folder on the host machine. This allows you to easily share files between the Docker container and the host machine, which we'll explore in the[Docker data volume article](https://www.digitalocean.com/community/tutorials/how-to-work-with-docker-data-volumes-on-ubuntu-14-04).

The third way to communicate with a Docker container is via the network. Docker allows communication between different Docker containers via links, as well as port forwarding, allowing you to forward ports from inside the Docker container to ports on the host server. For example, you can create a link to allow your WordPress and MariaDB Docker containers to talk to each other and port-forwarding to expose WordPress to the outside world so that users can connect to it.

Prerequisites

To follow this article, you will need the following:

* Ubuntu 14.04 Droplet
* A non-root user with sudo privileges ([Initial Server Setup with Ubuntu 14.04](https://www.digitalocean.com/community/tutorials/initial-server-setup-with-ubuntu-14-04) explains how to set this up.)

Step 1 — Installing Docker

First, install Docker if you haven't already. The quickest way to install Docker is to download and install their installation script (you'll be prompted for a sudo password).

* wget -qO- https://get.docker.com/ | sh

The above command downloads and executes a small installation script written by the Docker team. If you don't trust third party scripts or want more details about what the script is doing check out the instructions in the [DigitalOcean Docker tutorial](https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-getting-started) or Docker's own [installation documentation](https://docs.docker.com/installation/ubuntulinux/).

Working with Docker is a pain if your user is not configured correctly, so add your user to the dockergroup with the following command.

**sudo** usermod -aG docker $(whoami)

Log out and log in from your server to activate your new groups.

**Note:** To learn more about how to use Docker, read the *How to Use Docker* section of [How To Install and Use Docker: Getting Started](https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-getting-started#how-to-use-docker).

Step 2 — Installing Docker Compose

Now that you have Docker installed, let's go ahead and install Docker Compose. First, install python-pipas prerequisite:

* sudo apt-get -y install python-pip

Then you can install Docker Compose:

* sudo pip install docker-compose

Step 3 — Running a Container with Docker Compose

The public Docker registry, Docker Hub, includes a simple *Hello World* image. Now that we have Docker Compose installed, let's test it with this really simple example.

First, create a directory for our YAML file:

* mkdir hello-world

Then change into the directory:

* cd hello-world

Now create the YAML file using your favorite text editor (we will use nano):

* nano docker-compose.yml

Put the following contents into the file, save the file, and exit the text editor:

docker-compose.yml

my-test:

image: hello-world

The first line will be used as part of the container name. The second line specifies which image to use to create the container. The image will be downloaded from the official Docker Hub repository.

While still in the ~/hello-world directory, execute the following command to create the container:

* docker-compose up

The output should start with the following:

Output of 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.

my-test\_1 |

The output then explains what Docker is doing:

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 you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

If the process doesn't exit on its own, press CTRL-C.

This simple test does not show one of the main benefits of Docker Compose — being able to bring a group of Docker containers up and down all at the same time. The [How To Install Wordpress and PhpMyAdmin with Docker Compose on Ubuntu 14.04](https://www.digitalocean.com/community/tutorials/how-to-install-wordpress-and-phpmyadmin-with-docker-compose-on-ubuntu-14-04) articles show how to use Docker Compose to run three containers as one application group.

Step 4 — Learning Docker Compose Commands

Let's go over the commands the docker-compose tool supports.

The docker-compose command works on a per-directory basis. You can have multiple groups of Docker containers running on one machine — just make one directory for each container and one docker-compose.yml file for each container inside its directory.

So far we've been running docker-compose up on our own and using CTRL-C to shut it down. This allows debug messages to be displayed in the terminal window. This isn't ideal though, when running in production you'll want to have docker-compose act more like a service. One simple way to do this is to just add the -d option when you up your session:

* docker-compose up -d

docker-compose will now fork to the background.

To show your group of Docker containers (both stopped and currently running), use the following command:

* docker-compose ps

For example, the following shows that the helloworld\_my-test\_1 container is stopped:

Output of `docker-compose ps`

Name Command State Ports

-----------------------------------------------

helloworld\_my-test\_1 /hello Exit 0

A running container will show the Up state:

Output of `docker-compose ps`

Name Command State Ports

---------------------------------------------------------------

nginx\_nginx\_1 nginx -g daemon off; Up 443/tcp, 80/tcp

To stop all running Docker containers for an application group, issue the following command in the same directory as the docker-compose.yml file used to start the Docker group:

* docker-compose stop

**Note:** docker-compose kill is also available if you need to shut things down more forcefully.

In some cases, Docker containers will store their old information in an internal volume. If you want to start from scratch you can use the rm command to fully delete all the containers that make up your container group:

* docker-compose rm

If you try any of these commands from a directory other than the directory that contains a Docker container and .yml file, it will complain and not show you your containers:

Output from wrong directory

Can't find a suitable configuration file in this directory or any parent. Are you in the right directory?

Supported filenames: docker-compose.yml, docker-compose.yaml, fig.yml, fig.yaml

Step 5 — Accessing the Docker Container Filesystem (Optional)

If you need to work on the command prompt inside a container, you can use the docker exec command.

The *Hello World!* example exits after it is run, so we need to start a container that will keep running so we can then use docker exec to access the filesystem for the container. Let's take a look at the [Nginx image](https://hub.docker.com/_/nginx/)from Docker Hub.

Create a new directory for it and change into it:

* mkdir ~/nginx && cd $\_

Create a docker-compose.yml file in our new directory:

* nano docker-compose.yml

and paste in the following:

~/nginx/docker-compose.yml

nginx:

image: nginx

Save the file and exit. We just need to start the Nginx container as a background process with the following command:

* docker-compose up -d

The Nginx image will be downloaded and then the container will be started in the background.

Now we need the CONTAINER ID for the container. List of all the containers that are running:

* docker ps

You will see something similar to the following:

Output of `docker ps`

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

e90e12f70418 nginx "nginx -g 'daemon off" 6 minutes ago Up 5 minutes 80/tcp, 443/tcp nginx\_nginx\_1

**Note:** Only *running* containers are listed with the docker ps command.

If we wanted to make a change to the filesystem inside this container, we'd take its ID (in this examplee90e12f70418) and use docker exec to start a shell inside the container:

* docker exec -it e90e12f70418 /bin/bash

The -t option opens up a terminal, and the -i option makes it interactive. The /bin/bash options opens a bash shell to the running container. Be sure to use the ID for your container.

You will see a bash prompt for the container similar to:

root@e90e12f70418:/#

From here, you can work from the command prompt. Keep in mind, however, that unless you are in a directory that is saved as part of a data volume, your changes will disappear as soon as the container is restarted. Another caveat is that most Docker images are created with very minimal Linux installs, so some of the command line utilities and tools you are used to may not be present.

Conclusion

Great, so that covers the basic concepts of Docker Compose and how to get it installed and running. Check out the [Deploying Wordpress and PHPMyAdmin with Docker Compose on Ubuntu 14.04](https://www.digitalocean.com/community/tutorials/how-to-install-wordpress-and-phpmyadmin-with-docker-compose-on-ubuntu-14-04) tutorial for a more complicated example of how to deploy an application with Docker Compose.

For a complete list of configuration options for the docker-compose.yml file refer to the [Compose file reference](https://docs.docker.com/compose/compose-file/).

How To Install Wordpress and PhpMyAdmin with Docker Compose on Ubuntu 14.04

Posted Nov 19, 2015

Top of Form

Bottom of Form

22

 43kviews [Docker](https://www.digitalocean.com/community/tags/docker?type=tutorials) [WordPress](https://www.digitalocean.com/community/tags/wordpress?type=tutorials) [MySQL](https://www.digitalocean.com/community/tags/mysql?type=tutorials) [MariaDB](https://www.digitalocean.com/community/tags/mariadb?type=tutorials) [PHP](https://www.digitalocean.com/community/tags/php?type=tutorials) [Ubuntu](https://www.digitalocean.com/community/tags/ubuntu?type=tutorials)

**Introduction**

[Docker Compose](https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-compose-on-ubuntu-14-04) makes dealing with the orchestration processes of Docker containers (such as starting up, shutting down, and setting up intra-container linking and volumes) really easy.

This article provides a real-world example of using Docker Compose to install an application, in this case WordPress with PHPMyAdmin as an extra. WordPress normally runs on a LAMP stack, which means Linux, Apache, MySQL/MariaDB, and PHP. The official WordPress Docker image includes Apache and PHP for us, so the only part we have to worry about is MariaDB.

Prerequisites

To follow this article, you will need the following:

* Ubuntu 14.04 Droplet
* A non-root user with sudo privileges ([Initial Server Setup with Ubuntu 14.04](https://www.digitalocean.com/community/tutorials/initial-server-setup-with-ubuntu-14-04) explains how to set this up.)
* Docker and Docker Compose installed from the instructions in [How To Install and Use Docker Compose on Ubuntu 14.04](https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-compose-on-ubuntu-14-04)

Step 1 — Installing WordPress

We'll be using the official [WordPress](https://hub.docker.com/_/wordpress/) and [MariaDB](https://hub.docker.com/_/mariadb/) Docker images. If you're curious, there's lots more info about these images and their configuration options on their respective GitHub and Docker Hub pages.

Let's start by making a folder where our data will live and creating a minimal docker-compose.yml file to run our WordPress container:

* mkdir ~/wordpress && cd $\_

Then create a ~/wordpress/docker-compose.yml with your favorite text editor (nano is easy if you don't have a preference):

* nano ~/wordpress/docker-compose.yml

and paste in the following:

~/wordpress/docker-compose.yml

wordpress:

image: wordpress

This just tells Docker Compose to start a new container called wordpress and download the wordpressimage from the Docker Hub.

We can bring the image up like so:

* docker-compose up

You'll see Docker download and extract the WordPress image from the Docker Hub, and after some time you'll get some error messages similar to the below:

Output

wordpress\_1 | error: missing WORDPRESS\_DB\_HOST and MYSQL\_PORT\_3306\_TCP environment variables

wordpress\_1 | Did you forget to --link some\_mysql\_container:mysql or set an external db

wordpress\_1 | with -e WORDPRESS\_DB\_HOST=hostname:port?

dockercompose\_wordpress\_1 exited with code 1

This is WordPress complaining that it can't find a database. Let's add a MariaDB image to the mix and link it up to fix that.

Step 2 — Installing MariaDB

To add the MariaDB image to the group, re-open docker-compose.yml with your text editor:

* nano ~/wordpress/docker-compose.yml

Change docker-compose.yml to match the below (be careful with the indentation, YAML files are white-space sensitive)

docker-compose.yml

wordpress:

image: wordpress

links:

- wordpress\_db:mysql

wordpress\_db:

image: mariadb

What we've done here is define a new container called wordpress\_db and told it to use the mariadbimage from the Docker Hub. We also told the our wordpress container to link our wordpress\_dbcontainer into the wordpress container and call it mysql (inside the wordpress container the hostnamemysql will be forwarded to our wordpress\_db container).

If you run docker-compose up again, you will see it download the MariaDB image, and you'll also see that we're not quite there yet though:

Output

wordpress\_db\_1 | error: database is uninitialized and MYSQL\_ROOT\_PASSWORD not set

wordpress\_db\_1 | Did you forget to add -e MYSQL\_ROOT\_PASSWORD=... ?

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\_db\_1 exited with code 1

wordpress\_wordpress\_1 exited with code 1

Gracefully stopping... (press Ctrl+C again to force)

WordPress is still complaining about being unable to find a database, and now we have a new complaint from MariaDB saying that no root password is set.

It appears that just linking the two containers isn't quite enough. Let's go ahead and set theMYSQL\_ROOT\_PASSWORD variable so that we can actually fire this thing up.

Edit the Docker Compose file yet again:

* nano ~/wordpress/docker-compose.yml

Add these two lines to the *end* of the wordpress\_db section, but **make sure to change examplepass to a more secure password!**

docker-compose.yml

wordpress\_db:

...

environment:

MYSQL\_ROOT\_PASSWORD: examplepass

...

This will set an environment variable inside the wordpress\_db container called MYSQL\_ROOT\_PASSWORDwith your desired password. The MariaDB Docker image is configured to check for this environment variable when it starts up and will take care of setting up the DB with a root account with the password defined as MYSQL\_ROOT\_PASSWORD.

While we're at it, let's also set up a port forward so that we can connect to our WordPress install once it actually loads up. Under the wordpress section add these two lines:

docker-compose.yml

wordpress:

...

ports:

- 8080:80

...

The first port number is the port number on the host, and the second port number is the port inside the container. So, this configuration forwards requests on port 8080 of the host to the default web server port 80 inside the container.

**Note:** If you would like Wordpress to run on the default web server port 80 on the host, change the previous line to 80:80 so that requests to port 80 on the host are forwarded to port 80 inside the Wordpress container.

Your complete docker-compose.yml file should now look like this:

docker-compose.yml

wordpress:

image: wordpress

links:

- wordpress\_db:mysql

ports:

- 8080:80

wordpress\_db:

image: mariadb

environment:

MYSQL\_ROOT\_PASSWORD: examplepass

With this configuration we can actually go ahead and fire up WordPress. This time, let's run it with the -doption, which will tell docker-compose to run the containers in the background so that you can keep using your terminal:

* docker-compose up -d

You'll see a whole bunch of text fly by your screen. Once it's calmed down, open up a web browser and browse to the IP  
of your DigitalOcean box on port 8080 (for example, if the IP address of your server is 123.456.789.123 you should type http://123.456.789.123:8080 into your browser.)

You should see a fresh WordPress installation page and be able to complete the install and blog as usual.

Because these are both official Docker images and are following all of Docker's best practices, each of these images have pre-defined, persistent volumes for you — meaning that if you restart the container, your blog posts will still be there. You can learn more about working with Docker volumes in the [Docker data volumes tutorial](https://www.digitalocean.com/community/tutorials/how-to-work-with-docker-data-volumes-on-ubuntu-14-04).

Step 3 — Adding a PhpMyAdmin Container

Great, that was relatively painless. Let's try getting a little fancy.

So far we've only been using official images, which the Docker team takes great pains to ensure are accurate. You may have noticed that we didn't have to give the WordPress container any environment variables to configure it. As soon as we linked it up to a properly configured MariaDB container everything just worked.

This is because there's a script inside the WordPress Docker container that actually grabs theMYSQL\_ROOT\_PASSWORD variable from our wordpress\_db container and uses that to connect to WordPress.

Let's venture out of the official image area a little bit and use a [community contributed PhpMyAdmin image](https://hub.docker.com/r/corbinu/docker-phpmyadmin/). Go ahead and edit docker-compose.yml one more time:

* nano docker-compose.yml

Paste the following at the end of the file:

docker-compose.yml

phpmyadmin:

image: corbinu/docker-phpmyadmin

links:

- wordpress\_db:mysql

ports:

- 8181:80

environment:

MYSQL\_USERNAME: root

MYSQL\_ROOT\_PASSWORD: examplepass

Be sure to replace examplepass with the exact same root password from the wordpress\_db container you setup earlier.

This grabs docker-phpmyadmin by community member corbinu, links it to our wordpress\_db container with the name mysql (meaning from inside the phpmyadmin container references to the hostname mysqlwill be forwarded to our wordpress\_db container), exposes its port 80 on port 8181 of the host system, and finally sets a couple of environment variables with our MariaDB username and password. This image does not automatically grab the MYSQL\_ROOT\_PASSWORD environment variable from the wordpress\_dbcontainer's environment the way the wordpress image does. We actually have to copy theMYSQL\_ROOT\_PASSWORD: examplepass line from the wordpress\_db container, and set the username toroot.

The complete docker-compose.yml file should now look like this:

docker-compose.yml

wordpress:

image: wordpress

links:

- wordpress\_db:mysql

ports:

- 8080:80

wordpress\_db:

image: mariadb

environment:

MYSQL\_ROOT\_PASSWORD: examplepass

phpmyadmin:

image: corbinu/docker-phpmyadmin

links:

- wordpress\_db:mysql

ports:

- 8181:80

environment:

MYSQL\_USERNAME: root

MYSQL\_ROOT\_PASSWORD: examplepass

Now start up the application group again:

* docker-compose up -d

You will see PhpMyAdmin being installed. Once it is finished, visit your server's IP address again (this time using port 8181, e.g. http://123.456.789.123:8181). You'll be greeted by the PhpMyAdmin login screen.

Go ahead and login using username root and password you set in the YAML file, and you'll be able to browse your database. You'll notice that the server includes a wordpress database, which contains all the data from your WordPress install.

You can add as many containers as you like this way and link them all up in any way you please. As you can see, the approach is quite powerful —instead of dealing with the configuration and prerequisites for each individual components and setting them all up on the same server, you get to plug the pieces together like Lego blocks and add components piecemeal. Using tools like [Docker Swarm](https://docs.docker.com/swarm/install-w-machine/) you can even transparently run these containers over multiple servers! That's a bitoutside the scope of this tutorial though. Docker provides some [documentation]((https://docs.docker.com/swarm/install-w-machine/)) on it if you are interested.

Step 4 — Creating the WordPress Site

Since all the files for your new WordPress site are stored inside your Docker container, what happens to your files when you stop the container and start it again?

By default, the document root for the WordPress container is persistent. This is because the WordPress image from the Docker Hub is configured this way. If you make a change to your WordPress site, stop the application group, and start it again, your website will still have the changes you made.

Let's try it.

Go to your WordPress from a web browser (e.g. http://123.456.789.123:8080). Edit the **Hello World!** post that already exists. Then, stop all the Docker containers with the following command:

* docker-compose stop

Try loading the WordPress site again. You will see that the website is down. Start the Docker containers again:

* docker-compose up -d

Again, load the WordPress site. You should see your blog site and the change you made earlier. This shows that the changes you make are saved even when the containers are stopped.

Step 5 — Storing the Document Root on the Host Filesystem (Optional)

It is possible to store the document root for WordPress on the host filesystem using a Docker data volume to share files between the host and the container.

**Note:** For more details on working with Docker data volumes, take a look at the [Docker data volumes tutorial](https://www.digitalocean.com/community/tutorials/how-to-work-with-docker-data-volumes-on-ubuntu-14-04). 

Let's give it a try. Open up your docker-compose.yml file one more time:

* nano ~/wordpress/docker-compose.yml

in the wordpress: section add the following lines:

~/wordpress/docker-compose.yml

wordpress:

...

volumes:

- ~/wordpress/wp\_html:/var/www/html

...

Stop your currently running docker-compose session:

* docker-compose stop

Remove the existing container so we can map the volume to the host filesystem:

* docker-compose rm wordpress

Start WordPress again:

* docker-compose -d

Once the prompt returns, WordPress should be up and running again — this time using the host filesystem to store the document root.

If you look in your ~/wordpress directory, you'll see that there is now a wp\_html directory in it:

* ls ~/wordpress

All of the WordPress source files are inside it. Changes you make will be picked up by the WordPress container in real time.

This experience was a little smoother than it normally would be — the WordPress Docker container is configured to check if /var/www/html is empty or not when it starts and copies files there appropriately. Usually you will have to do this step yourself.

Conclusion

You should have a full WordPress deploy up and running. You should be able to use the same method to deploy quite a wide variety of systems using the images available on the Docker Hub. Be sure to figure out which volumes are persistent and which are not for each container you create.