Contents

[Get Started, Part 1: Orientation and setup 6](#_Toc536784831)

[Docker concepts 7](#_Toc536784832)

[**Images and containers** 8](#_Toc536784833)

[**Containers and virtual machines** 9](#_Toc536784834)

[Prepare your Docker environment 12](#_Toc536784835)

[**Test Docker version** 12](#_Toc536784836)

[**Test Docker installation** 13](#_Toc536784837)

[Recap and cheat sheet 13](#_Toc536784838)

[Conclusion of part one 14](#_Toc536784839)

[Getting started with Docker 14](#_Toc536784840)

[Installation 14](#_Toc536784841)

[Start the Docker daemon at boot 15](#_Toc536784842)

[Why can’t I use docker command as a non root user, by default? 15](#_Toc536784843)

[Docker basics 16](#_Toc536784844)

[Getting started with images 16](#_Toc536784845)

[Running a container 16](#_Toc536784846)

[Creating an image 17](#_Toc536784847)

[Get Started, Part 2: Containers 17](#_Toc536784848)

[Prerequisites 17](#_Toc536784849)

[Introduction 18](#_Toc536784850)

[Your new development environment 18](#_Toc536784851)

[Define a container with Dockerfile 18](#_Toc536784852)

[**Dockerfile** 18](#_Toc536784853)

[The app itself 19](#_Toc536784854)

[**requirements.txt** 20](#_Toc536784855)

[**app.py** 20](#_Toc536784856)

[Build the app 21](#_Toc536784857)

[Run the app 22](#_Toc536784858)

[Share your image 24](#_Toc536784859)

[**Log in with your Docker ID** 24](#_Toc536784860)

[**Tag the image** 25](#_Toc536784861)

[**Publish the image** 25](#_Toc536784862)

[**Pull and run the image from the remote repository** 26](#_Toc536784863)

[Conclusion of part two 26](#_Toc536784864)

[Recap and cheat sheet (optional) 27](#_Toc536784865)

[Get Started, Part 3: Services 27](#_Toc536784866)

[Prerequisites 28](#_Toc536784867)

[Introduction 28](#_Toc536784868)

[About services 28](#_Toc536784869)

[Your first docker-compose.yml file 29](#_Toc536784870)

[Run your new load-balanced app 30](#_Toc536784871)

[Scale the app 32](#_Toc536784872)

[Take down the app and the swarm 32](#_Toc536784873)

[Recap and cheat sheet (optional) 32](#_Toc536784874)

[Get Started, Part 4: Swarms 33](#_Toc536784875)

[Prerequisites 33](#_Toc536784876)

[Introduction 34](#_Toc536784877)

[Understanding Swarm clusters 34](#_Toc536784878)

[Set up your swarm 35](#_Toc536784879)

[Getting started with swarm mode 35](#_Toc536784880)

[Set up 35](#_Toc536784881)

[**Three networked host machines** 35](#_Toc536784882)

[**Docker Engine 1.12 or newer** 36](#_Toc536784883)

[**The IP address of the manager machine** 37](#_Toc536784884)

[**Open protocols and ports between the hosts** 37](#_Toc536784885)

[What’s next? 37](#_Toc536784886)

[Create a swarm 37](#_Toc536784887)

[What’s next? 39](#_Toc536784888)

[Deploy your app on the swarm cluster 40](#_Toc536784889)

[Deploy the app on the swarm manager 40](#_Toc536784890)

[Lets make sure what we did previously has been stopped on all three docker hosts 40](#_Toc536784891)

[Take down the app and the swarm 40](#_Toc536784892)

[Accessing your cluster 42](#_Toc536784893)

[Iterating and scaling your app 43](#_Toc536784894)

[Cleanup and reboot 44](#_Toc536784895)

[Stacks and swarms 44](#_Toc536784896)

[Unsetting docker-machine shell variable settings 44](#_Toc536784897)

[Restarting Docker machines 44](#_Toc536784898)

[Recap and cheat sheet (optional) 45](#_Toc536784899)

[How to use Dockerfiles 47](#_Toc536784900)

[The basics 47](#_Toc536784901)

[Example to be used in file 49](#_Toc536784902)

[Building the image 49](#_Toc536784903)

[Find out more 50](#_Toc536784904)

[Introduction 50](#_Toc536784905)

[Glossary 51](#_Toc536784906)

[1. Docker in Brief 51](#_Toc536784907)

[2. Dockerfiles 51](#_Toc536784908)

[3. Dockerfile Syntax 51](#_Toc536784909)

[4. Dockerfile Commands 51](#_Toc536784910)

[5. How To Use Dockerfiles 51](#_Toc536784911)

[How to Setup DNS Server with Docker Container 51](#_Toc536784912)

[**Testing DNS Server** 52](#_Toc536784913)

[**Deployment and Maintenance** 53](#_Toc536784914)

[**Conclusion** 54](#_Toc536784915)

[6. Dockerfile Example: Creating an Image to Install MongoDB 54](#_Toc536784916)

[Docker in Brief 54](#_Toc536784917)

[Dockerfiles 54](#_Toc536784918)

[Dockerfile Syntax 55](#_Toc536784919)

[What is Syntax? 55](#_Toc536784920)

[Dockerfile Syntax Example 55](#_Toc536784921)

[Dockerfile Commands (Instructions) 55](#_Toc536784922)

[ADD 56](#_Toc536784923)

[CMD 56](#_Toc536784924)

[ENTRYPOINT 56](#_Toc536784925)

[ENV 57](#_Toc536784926)

[EXPOSE 57](#_Toc536784927)

[FROM 57](#_Toc536784928)

[RUN 58](#_Toc536784929)

[USER 58](#_Toc536784930)

[VOLUME 58](#_Toc536784931)

[WORKDIR 59](#_Toc536784932)

[How to Use Dockerfiles 59](#_Toc536784933)

[Dockerfile Example: Creating an Image to Install MongoDB 59](#_Toc536784934)

[Creating the Empty Dockerfile 59](#_Toc536784935)

[Defining Our File and Its Purpose 60](#_Toc536784936)

[Setting The Base Image to Use 60](#_Toc536784937)

[Defining The Maintainer (Author) 60](#_Toc536784938)

[Setting Arguments and Commands for Downloading MongoDB 60](#_Toc536784939)

[Setting The Default Port For MongoDB 61](#_Toc536784940)

[Saving The Dockerfile 61](#_Toc536784941)

[Building Our First Image 62](#_Toc536784942)

[Running A MongoDB Instance 62](#_Toc536784943)

[Writing a Dockerfile 63](#_Toc536784944)

[Building an image from Dockerfile 63](#_Toc536784945)

[The Docker Ecosystem: An Introduction to Common Components 63](#_Toc536784946)

[Introduction 63](#_Toc536784947)

[Docker and Containerization 64](#_Toc536784948)

[Service Discovery and Global Configuration Stores 65](#_Toc536784949)

[Networking Tools 67](#_Toc536784950)

[Scheduling, Cluster Management, and Orchestration 68](#_Toc536784951)

[Conclusion 70](#_Toc536784952)

[**The Docker Ecosystem** 70](#_Toc536784953)

[The Docker Ecosystem: An Overview of Containerization 71](#_Toc536784954)

[Introduction 71](#_Toc536784955)

[A Brief History of Linux Containerization 71](#_Toc536784956)

[What Containerization Brings to the Picture 72](#_Toc536784957)

[Abstraction of the host system away from the containerized application 72](#_Toc536784958)

[Easy Scalability 72](#_Toc536784959)

[Simple Dependency Management and Application Versioning 72](#_Toc536784960)

[Extremely lightweight, isolated execution environments 72](#_Toc536784961)

[Shared Layering 73](#_Toc536784962)

[Composability and Predictability 73](#_Toc536784963)

[Using Dockerfiles for Repeatable, Consistent Builds 73](#_Toc536784964)

[The Architecture of Containerized Applications 73](#_Toc536784965)

[Using a Docker Registry for Container Management 74](#_Toc536784966)

[Conclusion 75](#_Toc536784967)

[How To Install and Use Docker on Debian 9 75](#_Toc536784968)

[Introduction 75](#_Toc536784969)

[Initial Server Setup with Debian 9 75](#_Toc536784970)

[Introduction 75](#_Toc536784971)

[Step One — Logging in as Root 76](#_Toc536784972)

[About Root 76](#_Toc536784973)

[Step Two — Creating a New User 76](#_Toc536784974)

[Step Three — Granting Administrative Privileges 77](#_Toc536784975)

[Step Four — Setting Up a Basic Firewall 78](#_Toc536784976)

[Step Five — Enabling External Access for Your Regular User 79](#_Toc536784977)

[If the Root Account Uses Password Authentication 80](#_Toc536784978)

[If the Root Account Uses SSH Key Authentication 80](#_Toc536784979)

[Step Six — Completing Optional Configuration 81](#_Toc536784980)

[Installing man Pages 81](#_Toc536784981)

[Changing the Default Editor 82](#_Toc536784982)

[Where To Go From Here? 83](#_Toc536784983)

[Prerequisites 83](#_Toc536784984)

[Step 1 — Installing Docker 83](#_Toc536784985)

[Step 2 — Executing the Docker Command Without Sudo (Optional) 85](#_Toc536784986)

[Step 3 — Using the Docker Command 86](#_Toc536784987)

[Step 4 — Working with Docker Images 88](#_Toc536784988)

[Step 5 — Running a Docker Container 92](#_Toc536784989)

[Step 6 — Managing Docker Containers 93](#_Toc536784990)

[Step 7 — Committing Changes in a Container to a Docker Image 95](#_Toc536784991)

[Step 8 — Pushing Docker Images to a Docker Repository 96](#_Toc536784992)

[Conclusion 98](#_Toc536784993)

[How To Install Docker Compose on Debian 9 99](#_Toc536784994)

[Introduction 99](#_Toc536784995)

[Prerequisites 99](#_Toc536784996)

[Step 1 — Installing Docker Compose 99](#_Toc536784997)

[Step 2 — Running a Container with Docker Compose 100](#_Toc536784998)

[Step 3 — Removing the Image (Optional) 103](#_Toc536784999)

[Conclusion 103](#_Toc536785000)

[Installing Webmin Docker Container on Dply 104](#_Toc536785001)

[Step One 104](#_Toc536785002)

[Step Two 104](#_Toc536785003)

[Step Three 105](#_Toc536785004)

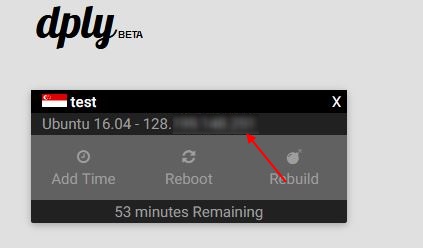
[**1-1 - Webmin Docker on Dply** 105](#_Toc536785005)

[Step Four 105](#_Toc536785006)

[Step Five 105](#_Toc536785007)

[Step Six 106](#_Toc536785008)

[Step Seven 106](#_Toc536785009)

[**** 106](#_Toc536785010)

[Step Eight 106](#_Toc536785011)

# Get Started, Part 1: Orientation and setup

*Estimated reading time: 4 minutes*

* [1: Orientation](https://docs.docker.com/get-started/part1)
* [2: Containers](https://docs.docker.com/get-started/part2)
* [3: Services](https://docs.docker.com/get-started/part3)
* [4: Swarms](https://docs.docker.com/get-started/part4)
* [5: Stacks](https://docs.docker.com/get-started/part5)
* [6: Deploy your app](https://docs.docker.com/get-started/part6)

Welcome! We are excited that you want to learn Docker. The Docker Get Started Tutorial teaches you how to:

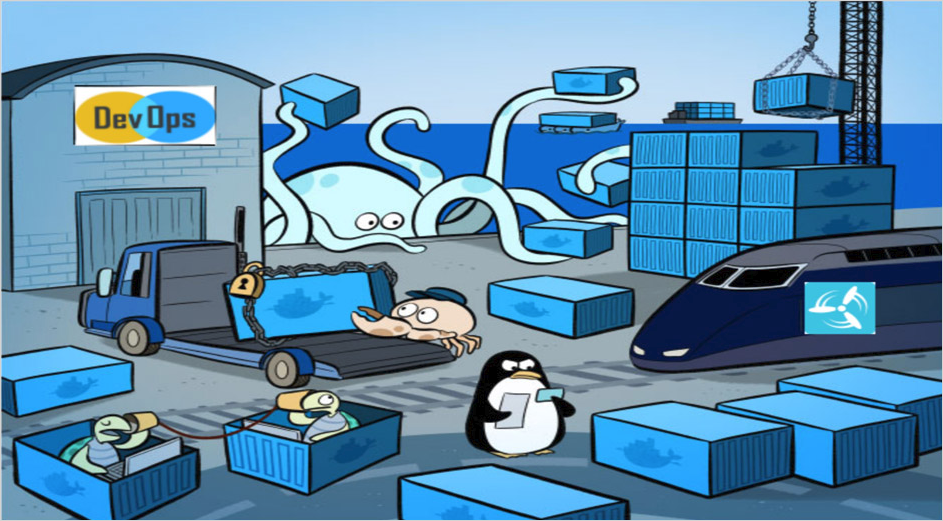
1. Set up your Docker environment (on this page)
2. [Build an image and run it as one container](https://docs.docker.com/get-started/part2/)
3. [Scale your app to run multiple containers](https://docs.docker.com/get-started/part3/)
4. [Distribute your app across a cluster](https://docs.docker.com/get-started/part4/)
5. [Stack services by adding a backend database](https://docs.docker.com/get-started/part5/)
6. [Deploy your app to production](https://docs.docker.com/get-started/part6/)

## Docker concepts

Docker is a platform for developers and sysadmins to **develop, deploy, and run** applications with containers. The use of Linux containers to deploy applications is called containerization. Containers are not new, but their use for easily deploying applications is.

Containerization is increasingly popular because containers are:

* Flexible: Even the most complex applications can be containerized.
* Lightweight: Containers leverage and share the host kernel.
* Interchangeable: You can deploy updates and upgrades on-the-fly.
* Portable: You can build locally, deploy to the cloud, and run anywhere.
* Scalable: You can increase and automatically distribute container replicas.
* Stackable: You can stack services vertically and on-the-fly.



### **Images and containers**

A container is launched by running an image. An **image** is an executable package that includes everything needed to run an application--the code, a runtime, libraries, environment variables, and configuration files.

A **container** is a runtime instance of an image--what the image becomes in memory when executed (that is, an image with state, or a user process). You can see a list of your running containers with the command, docker ps, just as you would in Linux.

### **Containers and virtual machines**

A **container** runs natively on Linux and shares the kernel of the host machine with other containers. It runs a discrete process, taking no more memory than any other executable, making it lightweight.

By contrast, a **virtual machine** (VM) runs a full-blown “guest” operating system with virtual access to host resources through a hypervisor. In general, VMs provide an environment with more resources than most applications need.

|  |  |
| --- | --- |
| Container stack example  Virtual machine stack example |  |

## Prepare your Docker environment

Install a [maintained version](https://docs.docker.com/engine/installation/#updates-and-patches) of Docker Community Edition (CE) or Enterprise Edition (EE) on a [supported platform](https://docs.docker.com/ee/supported-platforms/).

**For full Kubernetes Integration**

* [Kubernetes on Docker Desktop for Mac](https://docs.docker.com/docker-for-mac/kubernetes/) is available in [17.12 Edge (mac45)](https://docs.docker.com/docker-for-mac/edge-release-notes/#docker-community-edition-17120-ce-mac45-2018-01-05) or [17.12 Stable (mac46)](https://docs.docker.com/docker-for-mac/release-notes/#docker-community-edition-17120-ce-mac46-2018-01-09) and higher.
* [Kubernetes on Docker Desktop for Windows](https://docs.docker.com/docker-for-windows/kubernetes/) is available in [18.02 Edge (win50)](https://docs.docker.com/docker-for-windows/edge-release-notes/#docker-community-edition-18020-ce-rc1-win50-2018-01-26) and higher edge channels only.

[Install Docker](https://docs.docker.com/engine/installation/)

### **Test Docker version**

1. Run docker --version and ensure that you have a supported version of Docker:
2. docker --version
3. Docker version 17.12.0-ce, build c97c6d6
4. Run docker info or (docker version without --) to view even more details about your docker installation:
5. docker info
6. Containers: 0
7. Running: 0
8. Paused: 0
9. Stopped: 0
10. Images: 0
11. Server Version: 17.12.0-ce
12. Storage Driver: overlay2
13. ...

To avoid permission errors (and the use of sudo), add your user to the docker group. [Read more](https://docs.docker.com/engine/installation/linux/linux-postinstall/).

### **Test Docker installation**

1. Test that your installation works by running the simple Docker image, [hello-world](https://hub.docker.com/_/hello-world/):
2. docker run hello-world
3. Unable to find image 'hello-world:latest' locally
4. latest: Pulling from library/hello-world
5. ca4f61b1923c: Pull complete
6. Digest: sha256:ca0eeb6fb05351dfc8759c20733c91def84cb8007aa89a5bf606bc8b315b9fc7
7. Status: Downloaded newer image for hello-world:latest
8. Hello from Docker!
9. This message shows that your installation appears to be working correctly.
10. ...
11. List the hello-world image that was downloaded to your machine:
12. docker image ls
13. List the hello-world container (spawned by the image) which exits after displaying its message. If it were still running, you would not need the --all option:
14. docker container ls --all
15. CONTAINER ID IMAGE COMMAND CREATED STATUS
16. 54f4984ed6a8 hello-world "/hello" 20 seconds ago Exited (0) 19 seconds ago

## Recap and cheat sheet

## List Docker CLI commands

docker

docker container --help

## Display Docker version and info

docker --version

docker version

docker info

## Execute Docker image

docker run hello-world

## List Docker images

docker image ls

## List Docker containers (running, all, all in quiet mode)

docker container ls

docker container ls --all

docker container ls -aq

## Conclusion of part one

Containerization makes [CI/CD](https://www.docker.com/solutions/cicd) seamless. For example:

* applications have no system dependencies
* updates can be pushed to any part of a distributed application
* resource density can be optimized.

With Docker, scaling your application is a matter of spinning up new executables, not running heavy VM hosts.

# How To Install and Use Docker on Debian 9

### Introduction

[Docker](https://www.docker.com/) is an application that simplifies the process of managing application processes in containers. Containers let you run your applications in resource-isolated processes. They're similar to virtual machines, but containers are 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).

In this tutorial, you'll install and use Docker Community Edition (CE) on Debian 9. You'll install Docker itself, work with containers and images, and push an image to a Docker Repository.

# Initial Server Setup with Debian 9

### Introduction

When you first create a new Debian 9 server, there are a few configuration steps that you should take early on as part of the basic setup. This will increase the security and usability of your server and will give you a solid foundation for subsequent actions.

## Step One — Logging in as Root

To log into your server, you will need to know your **server's public IP address**. You will also need the password or, if you installed an SSH key for authentication, the private key for the **root** user's account. If you have not already logged into your server, you may want to follow our guide on [how to connect to your Droplet with SSH](https://www.digitalocean.com/community/tutorials/how-to-connect-to-your-droplet-with-ssh), which covers this process in detail.

If you are not already connected to your server, go ahead and log in as the **root** user using the following command (substitute the highlighted portion of the command with your server's public IP address):

* ssh root@your\_server\_ip

Accept the warning about host authenticity if it appears. If you are using password authentication, provide your **root** password to log in. If you are using an SSH key that is passphrase protected, you may be prompted to enter the passphrase the first time you use the key each session. If this is your first time logging into the server with a password, you may also be prompted to change the **root** password.

### About Root

The **root** user is the administrative user in a Linux environment that has very broad privileges. Because of the heightened privileges of the **root** account, you are discouraged from using it on a regular basis. This is because part of the power inherent with the **root** account is the ability to make very destructive changes, even by accident.

The next step is to set up an alternative user account with a reduced scope of influence for day-to-day work. We'll teach you how to gain increased privileges during the times when you need them.

## Step Two — Creating a New User

Once you are logged in as **root**, we're prepared to add the new user account that we will use to log in from now on.

#useradd asmith

Does not create a home folder with the above

#adduser asmith

**Note**: In some environments, a package called unscd may be installed by default in order to speed up requests to name servers like LDAP. The most recent version currently available in Debian contains [a bug](https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=844447) that causes certain commands (like the adduser command below) to produce additional output that looks like this:

sent invalidate(passwd) request, exiting

sent invalidate(group) request, exiting

These messages are harmless, but if you wish to avoid them, it is safe to remove the unscd package if you do not not plan on using systems like LDAP for user information:

* apt remove unscd

This example creates a new user called **sammy**, but you should replace it with a username that you like:

* adduser sammy

$su

#apt-get install open-vm-tools

#apt-get install openssh-server

You will be asked a few questions, starting with the account password.

Enter a strong password and, optionally, fill in any of the additional information if you would like. This is not required and you can just hit ENTER in any field you wish to skip.

## Step Three — Granting Administrative Privileges

Now, we have a new user account with regular account privileges. However, we may sometimes need to do administrative tasks.

To avoid having to log out of our normal user and log back in as the **root** account, we can set up what is known as "superuser" or **root** privileges for our normal account. This will allow our normal user to run commands with administrative privileges by putting the word sudo before each command.

To add these privileges to our new user, we need to add the new user to the **sudo** group. By default, on Debian 9, users who belong to the **sudo** group are allowed to use the sudo command.

As **root**, run this command to add your new user to the **sudo** group (substitute the highlighted word with your new user):

* #usermod -aG sudo asmith

Now, when logged in as your regular user, you can type sudo before commands to perform actions with superuser privileges.

## Step Four — Setting Up a Basic Firewall

Debian servers can use firewalls to make sure only connections to certain services are allowed. Although the iptables firewall is installed by default, Debian does not strongly recommend any specific firewall. In this guide, we will install and use the UFW firewall to help set policies and manage exceptions.

We can use the apt package manager to install UFW. Update the local index to retrieve the latest information about available packages and then install the firewall by typing:

* apt update
* apt install ufw

**Note:** If your servers are running on DigitalOcean, you can optionally use [DigitalOcean Cloud Firewalls](https://www.digitalocean.com/community/tutorials/an-introduction-to-digitalocean-cloud-firewalls)instead of the UFW firewall. We recommend using only one firewall at a time to avoid conflicting rules that may be difficult to debug.

Firewall profiles allow UFW to manage sets of firewall rules for applications by name. Profiles for some common software are bundled with UFW by default and packages can register additional profiles with UFW during the installation process. OpenSSH, the service allowing us to connect to our server now, has a firewall profile that we can use.

You can see this by typing:

* ufw app list

Output

Available applications:

. . .

OpenSSH

. . .

We need to make sure that the firewall allows SSH connections so that we can log back in next time. We can allow these connections by typing:

* ufw allow OpenSSH

Afterwards, we can enable the firewall by typing:

* ufw enable

Type "y" and press ENTER to proceed. You can see that SSH connections are still allowed by typing:

* ufw status

Output

Status: active

To Action From

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

OpenSSH ALLOW Anywhere

OpenSSH (v6) ALLOW Anywhere (v6)

As **the firewall is currently blocking all connections except for SSH**, if you install and configure additional services, you will need to adjust the firewall settings to allow acceptable traffic in. You can learn some common UFW operations in [this guide](https://www.digitalocean.com/community/tutorials/ufw-essentials-common-firewall-rules-and-commands).

## Step Five — Enabling External Access for Your Regular User

Now that we have a regular user for daily use, we need to make sure we can SSH into the account directly.

**Note:** Until verifying that you can log in and use sudo with your new user, we recommend staying logged in as **root**. This way, if you have problems, you can troubleshoot and make any necessary changes as **root**. If you are using a DigitalOcean Droplet and experience problems with your **root** SSH connection, you can [log into the Droplet using the DigitalOcean Console](https://www.digitalocean.com/community/tutorials/how-to-use-the-digitalocean-console-to-access-your-droplet).

The process for configuring SSH access for your new user depends on whether your server's **root** account uses a password or SSH keys for authentication.

### If the Root Account Uses Password Authentication

If you logged in to your **root** account using a password, then password authentication is enabled for SSH. You can SSH to your new user account by opening up a new terminal session and using SSH with your new username:

* ssh sammy@your\_server\_ip

After entering your regular user's password, you will be logged in. Remember, if you need to run a command with administrative privileges, type sudo before it like this:

* sudo command\_to\_run

You will be prompted for your regular user password when using sudo for the first time each session (and periodically afterwards).

To enhance your server's security, **we strongly recommend setting up SSH keys instead of using password authentication**. Follow our guide on [setting up SSH keys on Debian 9](https://www.digitalocean.com/community/tutorials/how-to-set-up-ssh-keys-on-debian-9) to learn how to configure key-based authentication.

### If the Root Account Uses SSH Key Authentication

If you logged in to your **root** account using SSH keys, then password authentication is disabled for SSH. You will need to add a copy of your local public key to the new user's ~/.ssh/authorized\_keys file to log in successfully.

Since your public key is already in the **root** account's ~/.ssh/authorized\_keys file on the server, we can copy that file and directory structure to our new user account in our existing session with the cpcommand. Afterwards, we can adjust ownership of the files using the chown command.

Make sure to change the highlighted portions of the command below to match your regular user's name:

* cp -r ~/.ssh /home/sammy
* chown -R sammy:sammy /home/sammy/.ssh

Now, open up a new terminal session and using SSH with your new username:

* ssh sammy@your\_server\_ip

You should be logged in to the new user account without using a password. Remember, if you need to run a command with administrative privileges, type sudo before it like this:

* sudo command\_to\_run

You will be prompted for your regular user password when using sudo for the first time each session (and periodically afterwards).

## Step Six — Completing Optional Configuration

Now that we have a strong baseline configuration, we can consider a few optional steps to make the system more accessible. The following sections cover a few additional tweaks focused on usability.

### Installing man Pages

Debian provides extensive manuals for most software in the form of man pages. However, the mancommand is not always included by default on minimal installations.

Install the man-db package to install the man command and the manual databases:

* sudo apt install man-db

Now, to view the manual for a component, you can type:

* man command

For example, to view the manual for the top command, type:

* man top

Most packages in the Debian repositories include manual pages as part of their installation.

### Changing the Default Editor

Debian offers a wide variety of text editors, some of which are included in the base system. Commands with integrated editor support, like visudo and systemctl edit, pass text to the editor command, which is mapped to the system default editor. Setting the default editor according to your preferences can help you configure your system more easily and avoid frustration.

If your preferred editor is not installed by default, use apt to install it first:

* sudo apt install your\_preferred\_editor

Next, you can view the current default and modify the selection using the update-alternativescommand:

* sudo update-alternatives --config editor

The command displays a table of the editors it knows about with a prompt to change the default:

Output

There are 8 choices for the alternative editor (providing /usr/bin/editor).

Selection Path Priority Status

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

\* 0 /usr/bin/joe 70 auto mode

1 /bin/nano 40 manual mode

2 /usr/bin/jmacs 50 manual mode

3 /usr/bin/joe 70 manual mode

4 /usr/bin/jpico 50 manual mode

5 /usr/bin/jstar 50 manual mode

6 /usr/bin/rjoe 25 manual mode

7 /usr/bin/vim.basic 30 manual mode

8 /usr/bin/vim.tiny 15 manual mode

Press <enter> to keep the current choice[\*], or type selection number:

The asterisk in the far left column indicates the current selection. To change the default, type the "Selection" number for your preferred editor and press Enter. For example, to use nano as the default editor given the above table, we would choose 1:

Output

Press <enter> to keep the current choice[\*], or type selection number: 1

update-alternatives: using /bin/nano to provide /usr/bin/editor (editor) in manual mode

From now on, your preferred editor will be used by commands like visudo and systemctl edit, or when the editor command is called.

## Where To Go From Here?

At this point, you have a solid foundation for your server. You can install any of the software you need on your server now.

## Prerequisites

To follow this tutorial, you will need the following:

* One Debian 9 server set up by following [the Debian 9 initial server setup guide](https://www.digitalocean.com/community/tutorials/initial-server-setup-with-debian-9), including a sudo non-root user and a firewall.
* An account on [Docker Hub](https://hub.docker.com/) if you wish to create your own images and push them to Docker Hub, as shown in Steps 7 and 8.

## Step 1 — Installing Docker

The Docker installation package available in the official Debian repository may not be the latest version. To ensure we get the latest version, we'll install Docker from the official Docker repository. To do that, we'll add a new package source, add the GPG key from Docker to ensure the downloads are valid, and then install the package.

First, update your existing list of packages:

* #apt update

Next, install a few prerequisite packages which let apt use packages over HTTPS:

apt install apt-transport-https ca-certificates curl gnupg2 software-properties-common

Then add the GPG key for the official Docker repository to your system:

curl -fsSL https://download.docker.com/linux/debian/gpg |apt-key add -

Add the Docker repository to APT sources:

* add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/debian $(lsb\_release -cs) stable"

Next, update the package database with the Docker packages from the newly added repo:

* #apt update

Make sure you are about to install from the Docker repo instead of the default Debian repo:

**apt-cache policy docker-ce**

You'll see output like this, although the version number for Docker may be different:

Output of apt-cache policy docker-ce

docker-ce:

Installed: (none)

Candidate: 18.06.1~ce~3-0~debian

Version table:

18.06.1~ce~3-0~debian 500

500 https://download.docker.com/linux/debian stretch/stable amd64 Packages

Notice that docker-ce is not installed, but the candidate for installation is from the Docker repository for Debian 9 (stretch).

Finally, install Docker:

**#apt install docker-ce**

Docker should now be installed, the daemon started, and the process enabled to start on boot. Check that it's running:

#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 Thu 2018-07-05 15:08:39 UTC; 2min 55s ago

Docs: https://docs.docker.com

Main PID: 21319 (dockerd)

CGroup: /system.slice/docker.service

├─21319 /usr/bin/dockerd -H fd://

└─21326 docker-containerd --config /var/run/docker/containerd/containerd.toml

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 the Docker Command Without Sudo (Optional)

By default, the docker command can only be run the **root** user or by a user in the **docker** group, which is automatically created during Docker's installation process. If you attempt to run the docker command without prefixing it with sudo or 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:

#adduser asmith

sudo usermod -aG docker asmith

* sudo usermod -aG docker ${USER}

To apply the new group membership, log out of the server and back in, or type the following:

* su - ${USER}

You will be prompted to enter your user's password to continue.

Confirm that your user is now added to the **docker** group by typing:

* id -nG

Output

sammy sudo docker

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** group. If you choose not to, please prepend the commands with sudo.

Let's explore the docker command next.

## Step 3 — Using the Docker Command

Using docker consists of passing it a chain of options and commands followed by arguments. The syntax takes this form:

* docker [option] [command] [arguments]

To view all available subcommands, type:

* docker

As of Docker 18, the complete list of available subcommands includes:

Output

attach Attach local standard input, output, and error streams 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 to files or directories 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 Docker objects

kill Kill one or more running containers

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

pause Pause all processes within one or more containers

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 one or more containers

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 (streamed to STDOUT by default)

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 one or more running containers

tag Create a tag TARGET\_IMAGE that refers to SOURCE\_IMAGE

top Display the running processes of a container

unpause Unpause all processes within one or more containers

update Update configuration of one or more containers

version Show the Docker version information

wait Block until one or more containers stop, then print their exit codes

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

* docker docker-subcommand --help

To view system-wide information about Docker, use:

* docker info

Let's explore some of these commands. We'll start by working with images.

## Step 4 — Working with Docker Images

Docker containers are built from Docker images. By default, Docker pulls these images from [Docker Hub](https://hub.docker.com/), a Docker registry managed by Docker, the company behind the Docker project. Anyone can host their Docker images on Docker Hub, so most applications and Linux distributions you'll need will have images hosted there.

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

**docker run hello-world**

The output will indicate that Docker in working correctly:

Output

Unable to find image 'hello-world:latest' locally

latest: Pulling from library/hello-world

9db2ca6ccae0: Pull complete

Digest: sha256:4b8ff392a12ed9ea17784bd3c9a8b1fa3299cac44aca35a85c90c5e3c7afacdc

Status: Downloaded newer image for hello-world:latest

Hello from Docker!

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

...

Docker was initially unable to find the hello-world image locally, so it downloaded the image from Docker Hub, which is the default repository. Once the image downloaded, Docker created a container from the image and the application within the container executed, displaying the message.

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

* docker search ubuntu

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

ubuntu Ubuntu is a Debian-based Linux operating sys… 8320 [OK]

dorowu/ubuntu-desktop-lxde-vnc Ubuntu with openssh-server and NoVNC 214 [OK]

rastasheep/ubuntu-sshd Dockerized SSH service, built on top of offi… 170 [OK]

consol/ubuntu-xfce-vnc Ubuntu container with "headless" VNC session… 128 [OK]

ansible/ubuntu14.04-ansible Ubuntu 14.04 LTS with ansible 95 [OK]

ubuntu-upstart Upstart is an event-based replacement for th… 88 [OK]

neurodebian NeuroDebian provides neuroscience research s… 53 [OK]

1and1internet/ubuntu-16-nginx-php-phpmyadmin-mysql-5 ubuntu-16-nginx-php-phpmyadmin-mysql-5 43 [OK]

ubuntu-debootstrap debootstrap --variant=minbase --components=m… 39 [OK]

nuagebec/ubuntu Simple always updated Ubuntu docker images w… 23 [OK]

tutum/ubuntu Simple Ubuntu docker images with SSH access 18

i386/ubuntu Ubuntu is a Debian-based Linux operating sys… 13

1and1internet/ubuntu-16-apache-php-7.0 ubuntu-16-apache-php-7.0 12 [OK]

ppc64le/ubuntu Ubuntu is a Debian-based Linux operating sys… 12

eclipse/ubuntu\_jdk8 Ubuntu, JDK8, Maven 3, git, curl, nmap, mc, … 6 [OK]

darksheer/ubuntu Base Ubuntu Image -- Updated hourly 4 [OK]

codenvy/ubuntu\_jdk8 Ubuntu, JDK8, Maven 3, git, curl, nmap, mc, … 4 [OK]

1and1internet/ubuntu-16-nginx-php-5.6-wordpress-4 ubuntu-16-nginx-php-5.6-wordpress-4 3 [OK]

pivotaldata/ubuntu A quick freshening-up of the base Ubuntu doc… 2

1and1internet/ubuntu-16-sshd ubuntu-16-sshd 1 [OK]

ossobv/ubuntu Custom ubuntu image from scratch (based on o… 0

smartentry/ubuntu ubuntu with smartentry 0 [OK]

1and1internet/ubuntu-16-healthcheck ubuntu-16-healthcheck 0 [OK]

pivotaldata/ubuntu-gpdb-dev Ubuntu images for GPDB development 0

paasmule/bosh-tools-ubuntu Ubuntu based bosh-cli 0 [OK]

...

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

Execute the following command to download the official ubuntu image to your computer:

* docker pull ubuntu

You'll see the following output:

Output

Using default tag: latest

latest: Pulling from library/ubuntu

6b98dfc16071: Pull complete

4001a1209541: Pull complete

6319fc68c576: Pull complete

b24603670dc3: Pull complete

97f170c87c6f: Pull complete

Digest: sha256:5f4bdc3467537cbbe563e80db2c3ec95d548a9145d64453b06939c4592d67b6d

Status: Downloaded newer image for ubuntu:latest

After an image has been downloaded, you can then run a container using the downloaded image with the run subcommand. As you saw with the hello-world example, if an image has not been downloaded when docker is executed with the run subcommand, the Docker client will first download the image, then run a container using it.

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

* docker images

The output should look similar to the following:

Output

REPOSITORY TAG IMAGE ID CREATED SIZE

ubuntu latest 16508e5c265d 13 days ago 84.1MB

hello-world latest 2cb0d9787c4d 7 weeks ago 1.85kB

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.

Let's look at how to run containers in more detail.

## 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 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 Ubuntu. The combination of the **-i** and **–t** switches gives you interactive shell access into the container:

**docker run -it Ubuntu**

The above will start a new instance of an image in a new container

To start a container

Docker ps –a

#docker start ID

To connect to a running container

**docker attach ID\_of\_the\_container**

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

Output

root@d9b100f2f636:/#

Note the container id in the command prompt. In this example, it is d9b100f2f636. You'll need that container ID later to identify the container when you want to remove it.

Now you can run any command inside the container. For example, let's update the package database inside the container. You don't need to prefix any command with sudo, because you're operating inside the container as the **root** user:

* apt update
* apt upgrade

Then install any application in it. Let's install Node.js:

* apt install nodejs

This installs Node.js in the container from the official Ubuntu repository. When the installation finishes, verify that Node.js is installed:

* node -v

You'll see the version number displayed in your terminal:

Output

v8.10.0

Any changes you make inside the container only apply to that container.

To exit the container, type **exit** at the prompt.

Let's look at managing the containers on our system next.

To connect to a running

## Step 6 — Managing 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

In this tutorial, you started two containers; one from the hello-world image and another from the Ubuntu image. Both containers are no longer running, but they still exist on your system.

To view all containers — active and inactive, run docker ps with the -a switch:

* docker ps -a

You'll see output similar to this:

d9b100f2f636 ubuntu "/bin/bash" About an hour ago Exited (0) 8 minutes ago sharp\_volhard

01c950718166 hello-world "/hello" About an hour ago Exited (0) About an hour ago festive\_williams

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

* docker ps -l
* CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
* d9b100f2f636 ubuntu "/bin/bash" About an hour ago Exited (0) 10 minutes ago sharp\_volhard

To start a stopped container, use docker start, followed by the container ID or the container's name. Let's start the Ubuntu-based container with the ID of d9b100f2f636:

* docker start d9b100f2f636

To see active containers use the following

* docker ps

The container will start, and you can use docker ps to see its status:

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

d9b100f2f636 ubuntu "/bin/bash" About an hour ago Up 8 seconds sharp\_volhard

To stop a running container, use docker stop, followed by the container ID or name. This time, we'll use the name that Docker assigned the container, which is sharp\_volhard:

* docker stop sharp\_volhard

To see active containers use the following

* docker ps

Once you've decided you no longer need a container anymore, remove it with the docker rm command, again using either the container ID or the name. Use the docker ps -a command to find the container ID or name for the container associated with the hello-world image and remove it.

* docker rm festive\_williams

You can start a new container and give it a name using the --name switch. You can also use the –rm switch to create a container that removes itself when it's stopped. See the docker run help command for more information on these options and others.

Containers can be turned into images which you can use to build new containers. Let's look at how that works.

## Step 7 — Committing Changes in a Container to a Docker Image

When you start up a Docker image, you can create, modify, and delete files just like you can with a virtual machine. The changes that you make will only apply to that container. You can start and stop it, but once you destroy it with the docker rm command, the changes will be lost for good.

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

After installing Node.js inside the Ubuntu container, you now have a container running off an image, but the container is different from the image you used to create it. But you might want to reuse this Node.js container as the basis for new images later.

Then commit the changes to a new Docker image instance using the following command.

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

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.

For example, for the user **sammy**, with the container ID of d9b100f2f636, the command would be:

* docker commit -m "added Node.js" -a "sammy" d9b100f2f636 sammy/ubuntu-nodejs

When you commit an image, the new image is saved locally on your computer. Later in this tutorial, you'll learn how to push an image to a Docker registry like Docker Hub so others can access it.

Listing the Docker images again will show the new image, as well as the old one that it was derived from:

* docker images

You'll see output like this:

Output

REPOSITORY TAG IMAGE ID CREATED SIZE

sammy/ubuntu-nodejs latest 7c1f35226ca6 7 seconds ago 179MB

ubuntu latest 113a43faa138 4 weeks ago 81.2MB

hello-world latest e38bc07ac18e 2 months ago 1.85kB

In this example, ubuntu-nodejs is the new image, which was derived from the existing ubuntu image from Docker Hub. The size difference reflects the changes that were made. And in this example, the change was that NodeJS was installed. So next time you need to run a container using Ubuntu with NodeJS pre-installed, you can just use the new image.

You can also build Images from a Dockerfile, which lets you automate the installation of software in a new image. However, that's outside the scope of this tutorial.

Now let's share the new image with others so they can create containers from it.

## 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 learn how to create your own private Docker registry, check out [How To Set Up a Private Docker Registry on Ubuntu 14.04](https://www.digitalocean.com/community/tutorials/how-to-set-up-a-private-docker-registry-on-ubuntu-14-04).

To push your image, first log into Docker Hub.

docker login -u docker-registry-username

docker login -u asmith

You'll be prompted to authenticate using your Docker Hub password. If you specified the correct password, authentication should succeed.

**Note:** If your Docker registry username is different from the local username you used to create the image, you will have to tag your image with your registry username. For the example given in the last step, you would type:

* docker tag sammy/ubuntu-nodejs docker-registry-username/ubuntu-nodejs

For example

* docker tag asmith/ubuntu-nodejs asmith/ubuntu-nodejs:v1.6.14.2017
* docker tag imageID asmith/ubuntu-nodejs
* docker images
* docker push asmith/ubuntu-nodejs

Then you may push your own image using:

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

docker push asmith/docker-image-name



To push the ubuntu-nodejs image to the **sammy** repository, the command would be:

* docker push sammy/ubuntu-nodejs:v1.6.14.2017

The process may take some time to complete as it uploads the images, but when completed, the output will look like this:

Output

The push refers to a repository [docker.io/sammy/ubuntu-nodejs]

e3fbbfb44187: Pushed

5f70bf18a086: Pushed

a3b5c80a4eba: Pushed

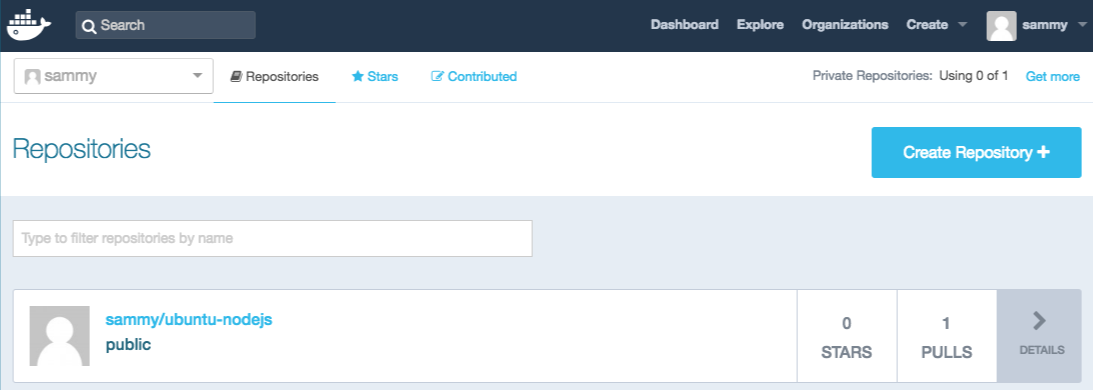
7f18b442972b: Pushed

3ce512daaf78: Pushed

7aae4540b42d: Pushed

...

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/sammy/ubuntu-nodejs]

e3fbbfb44187: Preparing

5f70bf18a086: Preparing

a3b5c80a4eba: Preparing

7f18b442972b: Preparing

3ce512daaf78: Preparing

7aae4540b42d: Waiting

unauthorized: authentication required

Log in with docker login and repeat the push attempt. Then verify that it exists on your Docker Hub repository page.

You can now use docker pull sammy/ubuntu-nodejs to pull the image to a new machine and use it to run a new container.

On your second docker host lets pull the image and create a container

docker login -u asmith

docker pull sammy/ubuntu-nodejs

## Conclusion

# Getting started with Docker

# Installation on Fedora

On Fedora 31 and above install the docker package:

set static ip

dnf update

dnf upgrade

sudo passwd

useradd dockerhubname

passwd dockerhubname

dnf install open-vm-tools

dnf install openssh-server

sytemctl enable sshd

systemctl start sshd

systemctl status sshd

sudo dnf -y update

sudo dnf -y install dnf-plugins-core

sudo dnf config-manager --add-repo https://download.docker.com/linux/fedora/docker-ce.repo

sudo dnf config-manager --set-enabled docker-ce-test

sudo dnf install docker-ce docker-ce-cli containerd.io

sudo systemctl enable --now docker

sudo dnf config-manager --set-enabled docker-ce-nightly

sudo systemctl start docker

sudo systemctl status docker

usermod -aG docker "dockerhubname"

id -nG

docker run hello-world

if you receive an error on the above use the following

dnf install -y grubby

grubby --update-kernel=ALL --args="systemd.unified\_cgroup\_hierarchy=0"

reboot

$ sudo docker run hello-world

Start the Docker daemon at boot

To make Docker start when you boot your system, use the command:

$ sudo systemctl enable docker

For additional systemd configuration options for Docker, like adding an HTTP Proxy, refer to the Docker documentation [Systemd article](https://docs.docker.com/engine/admin/systemd/).

# Why can’t I use docker command as a non root user, by default?

The Docker daemon binds to a Unix socket instead of a TCP port. By default that Unix socket is owned by the user root and other users can access it with sudo. For this reason, Docker daemon always runs as the root user.

You can either [set up sudo](http://www.projectatomic.io/blog/2015/08/why-we-dont-let-non-root-users-run-docker-in-centos-fedora-or-rhel) to give docker access to non-root users.

Or you can create a Unix group called docker and add users to it. When the Docker daemon starts, it makes the ownership of the Unix socket read/writable by the docker group.

**Warning:** The docker group is equivalent to the root user; For details on how this impacts security in your system, see [Docker Daemon Attack Surface](https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface) for details.

You have to log out and log back in (or restart Docker daemon and use newgrp command as mentioned here) for these changes to take effect. Then you can verify if your changes were successful by running Docker without sudo.

$docker run hello-world

# Docker basics

## Getting started with images

Use [docker images](https://docs.docker.com/engine/reference/commandline/images/) to see what images you have locally on your host.

$ sudo docker images

Use [docker search](https://docs.docker.com/engine/reference/commandline/search/) to see what images are available on [Docker Hub](https://hub.docker.com/).

$ sudo docker search fedora

## Running a container

Use docker run to run an application inside a container. The following command pulls Fedora image, creates a container from it and runs Bash shell inside it.

$ sudo docker run -it fedora bash

Other useful [options](https://docs.docker.com/engine/reference/run/#operator-exclusive-options) to use with run are for example -d to run a container in background (to daemonize it) or --name to give container a name which you can later use with docker stop/start.

See also [Docker run reference](https://docs.docker.com/engine/reference/run/).

Other useful commands regarding running containers are for example:

* [docker ps](https://docs.docker.com/engine/reference/commandline/ps/) to list running containers

docker ps -a

* [docker stop](https://docs.docker.com/engine/reference/commandline/stop/) to stop running container and [docker start](https://docs.docker.com/engine/reference/commandline/start/) to start stopped container

docker stop e1b7cdcff91f

docker start e1b7cdcff91f

* [docker logs](https://docs.docker.com/engine/reference/commandline/logs/) to look inside container
* [docker exec](https://docs.docker.com/engine/reference/commandline/exec/) to enter running container, like: docker exec -it [container-id] bash

Syntax

docker exec -it [container-id] bash

Example

docker exec -it e1b7cdcff91f bash

Creating an image

If you change anything (like install new packages in the above example with Bash) in the running container and exit the container the changes are not automatically saved into the Fedora image. If you want to save them in an image, use [docker commit](https://docs.docker.com/engine/reference/commandline/commit/).

Syntax

sudo docker commit CONTAINER\_ID nginx-template

Example

sudo docker commit e1b7cdcff91f httpd-template

To see your image :

docker image ls

Another option how to create an image is by building it from Dockerfile, see below.

# Get Started, Part 2: Containers

* [1: Orientation](https://docs.docker.com/get-started/part1)
* [2: Containers](https://docs.docker.com/get-started/part2)
* [3: Services](https://docs.docker.com/get-started/part3)
* [4: Swarms](https://docs.docker.com/get-started/part4)
* [5: Stacks](https://docs.docker.com/get-started/part5)
* [6: Deploy your app](https://docs.docker.com/get-started/part6)

## Prerequisites

* [Install Docker version 1.13 or higher](https://docs.docker.com/engine/installation/).
* Read the orientation in [Part 1](https://docs.docker.com/get-started/).
* Give your environment a quick test run to make sure you’re all set up:
* docker run hello-world

## Introduction

It’s time to begin building an app the Docker way. We start at the bottom of the hierarchy of such app, a container, which this page covers. Above this level is a service, which defines how containers behave in production, covered in [Part 3](https://docs.docker.com/get-started/part3/). Finally, at the top level is the stack, defining the interactions of all the services, covered in [Part 5](https://docs.docker.com/get-started/part5/).

* Stack
* Services
* **Container** (you are here)

## Your new development environment

In the past, if you were to start writing a Python app, your first order of business was to install a Python runtime onto your machine. But, that creates a situation where the environment on your machine needs to be perfect for your app to run as expected, and also needs to match your production environment.

With Docker, you can just grab a portable Python runtime as an image, no installation necessary. Then, your build can include the base Python image right alongside your app code, ensuring that your app, its dependencies, and the runtime, all travel together.

These portable images are defined by something called a Dockerfile.

## Define a container with Dockerfile

Dockerfile defines what goes on in the environment inside your container. Access to resources like networking interfaces and disk drives is virtualized inside this environment, which is isolated from the rest of your system, so you need to map ports to the outside world, and be specific about what files you want to “copy in” to that environment. However, after doing that, you can expect that the build of your app defined in this Dockerfile behaves exactly the same wherever it runs.

### **Dockerfile**

Create an empty directory. Change directories (cd) into the new directory, create a file called Dockerfile, copy-and-paste the following content into that file, and save it. Take note of the comments that explain each statement in your new Dockerfile.

Use Ssh to connect to the docker vm

#cd /

#mkdir ex2

#cd ex2

#vi Dockerfile

Nano Dockerfile

Copy and paste the following

# Use an official Python runtime as a parent image

FROM python:2.7-slim

# Set the working directory to /app

WORKDIR /app

# Copy the current directory contents into the container at /app

COPY . /app

# Install any needed packages specified in requirements.txt

RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container

EXPOSE 80

# Define environment variable

ENV NAME World

# Run app.py when the container launches

CMD ["python", "app.py"]

This Dockerfile refers to a couple of files we haven’t created yet, namely app.py and requirements.txt. Let’s create those next.

## The app itself

Create two more files, **requirements.txt** and **app.py**, and put them in the same folder with the Dockerfile. This completes our app, which as you can see is quite simple. When the above Dockerfile is built into an image, app.py and requirements.txt is present because of that Dockerfile’s COPY command, and the output from app.py is accessible over HTTP thanks to the EXPOSE command.

### **requirements.txt**

Flask

Redis

### **app.py**

from flask import Flask

from redis import Redis, RedisError

import os

import socket

# Connect to Redis

redis = Redis(host="redis", db=0, socket\_connect\_timeout=2, socket\_timeout=2)

app = Flask(\_\_name\_\_)

@app.route("/")

def hello():

try:

visits = redis.incr("counter")

except RedisError:

visits = "<i>cannot connect to Redis, counter disabled</i>"

html = "<h3>Hello {name}!</h3>" \

"<b>Hostname:</b> {hostname}<br/>" \

"<b>Visits:</b> {visits}"

return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethostname(), visits=visits)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host='0.0.0.0', port=80)

Now we see that pip install -r requirements.txt installs the Flask and Redis libraries for Python, and the app prints the environment variable NAME, as well as the output of a call to socket.gethostname(). Finally, because Redis isn’t running (as we’ve only installed the Python library, and not Redis itself), we should expect that the attempt to use it here fails and produces the error message.

**Note**: Accessing the name of the host when inside a container retrieves the container ID, which is like the process ID for a running executable.

That’s it! You don’t need Python or anything in requirements.txt on your system, nor does building or running this image install them on your system. It doesn’t seem like you’ve really set up an environment with Python and Flask, but you have.

## Build the app

We are ready to build the app. Make sure you are still at the top level of your new directory. Here’s what ls should show:

$ ls

Dockerfile app.py requirements.txt

Now run the build command. This creates a Docker image, which we’re going to name using the --tagoption. Use -t if you want to use the shorter option.

**docker build --tag=friendlyhello .**

Where is your built image? It’s in your machine’s local Docker image registry:

$ docker image ls

REPOSITORY TAG IMAGE ID

friendlyhello latest 326387cea398

Note how the tag defaulted to latest. The full syntax for the tag option would be something like --tag=friendlyhello:v0.0.1.

**Troubleshooting for Linux users**

Proxy server settings

Proxy servers can block connections to your web app once it’s up and running. If you are behind a proxy server, add the following lines to your Dockerfile, using the ENV command to specify the host and port for your proxy servers:

# Set proxy server, replace host:port with values for your servers

ENV http\_proxy host:port

ENV https\_proxy host:port

DNS settings

DNS misconfigurations can generate problems with pip. You need to set your own DNS server address to make pip work properly. You might want to change the DNS settings of the Docker daemon. You can edit (or create) the configuration file at **/etc/docker/daemon.json** with the dns key, as following:

{

"dns": ["your\_dns\_address", "8.8.8.8"]

}

For example in our setup use the dns server we created in a separate container previously

{

"dns": ["10.64.71.1", "8.8.8.8"]

}

In the example above, the first element of the list is the address of your DNS server. The second item is the Google’s DNS which can be used when the first one is not available.

Before proceeding, save daemon.json and restart the docker service.

sudo service docker restart

Once fixed, retry to run the build command.

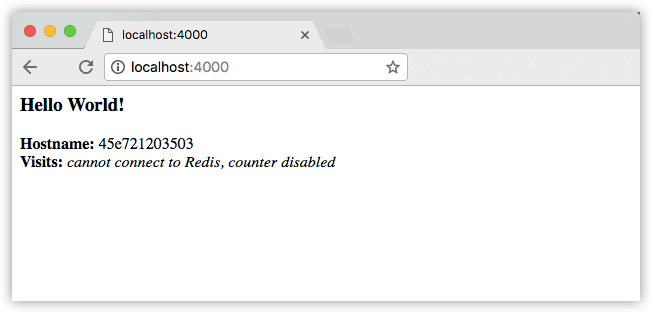
## Run the app

Run the app, mapping your machine’s port 4000 to the container’s published port 80 using -p:

**docker run -p 4000:80 friendlyhello**

You should see a message that **Python is serving your app at http://0.0.0.0:80**. But that message is coming from inside the container, which doesn’t know you mapped port 80 of that container to 4000, making the correct URL **http://localhost:4000**.

Go to that URL in a web browser to see the display content served up on a web page.



**Note**: If you are using Docker Toolbox on Windows 7, use the Docker Machine IP instead of localhost. For example, http://192.168.99.100:4000/. To find the IP address, use the command docker-machine ip.

You can also use the curl command in a shell to view the same content.

$ **curl http://localhost:4000**

<h3>Hello World!</h3><b>Hostname:</b> 8fc990912a14<br/><b>Visits:</b> <i>cannot connect to Redis, counter disabled</i>

This port remapping of 4000:80 demonstrates the difference between EXPOSE within the Dockerfile and what the publish value is set to when running docker run -p. In later steps, map port 4000 on the host to port 80 in the container and use http://localhost.

Hit CTRL+C in your terminal to quit.

**On Windows, explicitly stop the container**

On Windows systems, CTRL+C does not stop the container. So, first type CTRL+C to get the prompt back (or open another shell), **then type**

**docker container ls**

to list the running containers, **followed by**

**docker container stop <Container NAME or ID>** to stop the container.

Otherwise, you get an error response from the daemon when you try to re-run the container in the next step.

Now let’s run the app in the background, in detached mode:

**docker run -d -p 4000:80 friendlyhello**

You get the long container ID for your app and then are kicked back to your terminal. Your container is running in the background. You can also see the abbreviated container ID with docker container ls(and both work interchangeably when running commands):

$ **docker container ls**

CONTAINER ID IMAGE COMMAND CREATED

1fa4ab2cf395 friendlyhello "python app.py" 28 seconds ago

Notice that CONTAINER ID matches what’s on http://localhost:4000.

Now use docker container stop to end the process, using the CONTAINER ID, like so:

**docker container stop 1fa4ab2cf395**

## Share your image

To demonstrate the portability of what we just created, let’s upload our built image and run it somewhere else. After all, you need to know how to push to registries when you want to deploy containers to production.

A registry is a collection of repositories, and a repository is a collection of images—sort of like a GitHub repository, except the code is already built. An account on a registry can create many repositories. The docker CLI uses Docker’s public registry by default.

**Note**: We use Docker’s public registry here just because it’s free and pre-configured, but there are many public ones to choose from, and you can even set up your own private registry using [Docker Trusted Registry](https://docs.docker.com/datacenter/dtr/2.2/guides/).

### **Log in with your Docker ID**

If you don’t have a Docker account, sign up for one at [hub.docker.com](https://hub.docker.com/). Make note of your username.

Log in to the Docker public registry on your local machine.

$ **docker login**

### **Tag the image**

The notation for associating a local image with a repository on a registry is **username/repository:tag**. The tag is optional, but recommended, since it is the mechanism that registries use to give Docker images a version. Give the repository and tag meaningful names for the context, such asget-started:part2. This puts the image in the get-started repository and tag it as part2.

Now, put it all together to tag the image. Run docker tag image with your username, repository, and tag names so that the image uploads to your desired destination. The syntax of the command is:

**docker tag image username/repository:tag**

For example:

**docker tag friendlyhello gordon/repo:part2**

Run [docker image ls](https://docs.docker.com/engine/reference/commandline/image_ls/) to see your newly tagged image.

$ **docker image ls**

REPOSITORY TAG IMAGE ID CREATED SIZE

friendlyhello latest d9e555c53008 3 minutes ago 195MB

gordon/get-started part2 d9e555c53008 3 minutes ago 195MB

python 2.7-slim 1c7128a655f6 5 days ago 183MB

...

### **Publish the image**

Upload your tagged image to the repository:

**docker push username/repository:tag**

**For example**

**docker push asmith/repo:friendlyhello**

Once complete, the results of this upload are publicly available. If you log in to [Docker Hub](https://hub.docker.com/), you see the new image there, with its pull command.

### **Pull and run the image from the remote repository**

From now on, you can use docker run and run your app on any machine with this command:

docker run -p 4000:80 username/repository:tag

If the image isn’t available locally on the machine, Docker pulls it from the repository.

$ **docker run -p 4000:80 gordon/get-started:part2**

**For our example above we would use the following**

**docker run –p 4000:80 asmith/repo1:friendlyhello**

Unable to find image 'gordon/get-started:part2' locally

part2: Pulling from gordon/get-started

10a267c67f42: Already exists

f68a39a6a5e4: Already exists

9beaffc0cf19: Already exists

3c1fe835fb6b: Already exists

4c9f1fa8fcb8: Already exists

ee7d8f576a14: Already exists

fbccdcced46e: Already exists

Digest: sha256:0601c866aab2adcc6498200efd0f754037e909e5fd42069adeff72d1e2439068

Status: Downloaded newer image for gordon/get-started:part2

\* Running on http://0.0.0.0:80/ (Press CTRL+C to quit)

No matter where docker run executes, it pulls your image, along with Python and all the dependencies from requirements.txt, and runs your code. It all travels together in a neat little package, and you don’t need to install anything on the host machine for Docker to run it.

## Conclusion of part two

That’s all for this page. In the next section, we learn how to scale our application by running this container in a **service**.

[Continue to Part 3 >>](https://docs.docker.com/get-started/part3/)

## Recap and cheat sheet (optional)

Here’s [a terminal recording of what was covered on this page](https://asciinema.org/a/blkah0l4ds33tbe06y4vkme6g):

Here is a list of the basic Docker commands from this page, and some related ones if you’d like to explore a bit before moving on.

docker build -t friendlyhello . # Create image using this directory's Dockerfile

docker run -p 4000:80 friendlyhello # Run "friendlyname" mapping port 4000 to 80

docker run -d -p 4000:80 friendlyhello # Same thing, but in detached mode

docker container ls # List all running containers

docker container ls -a # List all containers, even those not running

docker container stop <hash> # Gracefully stop the specified container

docker container kill <hash> # Force shutdown of the specified container

docker container rm <hash> # Remove specified container from this machine

docker container rm $(docker container ls -a -q) # Remove all containers

docker image ls -a # List all images on this machine

docker image rm <image id> # Remove specified image from this machine

docker image rm $(docker image ls -a -q) # Remove all images from this machine

docker login # Log in this CLI session using your Docker credentials

docker tag <image> username/repository:tag # Tag <image> for upload to registry

docker push username/repository:tag # Upload tagged image to registry

docker run username/repository:tag # Run image from a registry

[containers](https://docs.docker.com/glossary/?term=containers), [python](https://docs.docker.com/glossary/?term=python), [code](https://docs.docker.com/glossary/?term=code), [coding](https://docs.docker.com/glossary/?term=coding), [build](https://docs.docker.com/glossary/?term=build), [push](https://docs.docker.com/glossary/?term=push), [run](https://docs.docker.com/glossary/?term=run)

# Get Started, Part 3: Services

* [1: Orientation](https://docs.docker.com/get-started/part1)
* [2: Containers](https://docs.docker.com/get-started/part2)
* [3: Services](https://docs.docker.com/get-started/part3)
* [4: Swarms](https://docs.docker.com/get-started/part4)
* [5: Stacks](https://docs.docker.com/get-started/part5)
* [6: Deploy your app](https://docs.docker.com/get-started/part6)

Prerequisites

* [Install Docker version 1.13 or higher](https://docs.docker.com/engine/installation/).
* Get [Docker Compose](https://docs.docker.com/compose/overview/). On [Docker Desktop for Mac](https://docs.docker.com/docker-for-mac/) and [Docker Desktop for Windows](https://docs.docker.com/docker-for-windows/) it’s pre-installed, so you’re good-to-go. On Linux systems you need to [install it directly](https://github.com/docker/compose/releases). On pre Windows 10 systems *without Hyper-V*, use [Docker Toolbox](https://docs.docker.com/toolbox/overview/).
* Read the orientation in [Part 1](https://docs.docker.com/get-started/).
* Learn how to create containers in [Part 2](https://docs.docker.com/get-started/part2/).
* Make sure you have published the friendlyhello image you created by [pushing it to a registry](https://docs.docker.com/get-started/part2/#share-your-image). We use that shared image here.
* Be sure your image works as a deployed container. Run this command, slotting in your info for username, repo, and tag: docker run -p 4000:80 username/repo:tag, then visit http://localhost:4000/.

Introduction

In part 3, we scale our application and enable load-balancing. To do this, we must go one level up in the hierarchy of a distributed application: the **service**.

* Stack
* **Services** (you are here)
* Container (covered in [part 2](https://docs.docker.com/get-started/part2/))

About services

In a distributed application, different pieces of the app are called “services”. For example, if you imagine a video sharing site, it probably includes a service for storing application data in a database, a service for video transcoding in the background after a user uploads something, a service for the front-end, and so on.

Services are really just “containers in production.” A service only runs one image, but it codifies the way that image runs—what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on. Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.

Luckily it’s very easy to define, run, and scale services with the Docker platform -- just write a docker-compose.yml file.

Your first docker-compose.yml file

A docker-compose.yml file is a YAML file that defines how Docker containers should behave in production.

docker-compose.yml

#cd /

#mkdir ex3

#cd /ex3

#nano docker-compose.yml

Save this file as **docker-compose.yml** . Be sure you have [pushed the image](https://docs.docker.com/get-started/part2/#share-your-image) you created in [Part 2](https://docs.docker.com/get-started/part2/) to a registry, and update this .yml by replacing username/repo:tag with your image details.

version: "3"

services:

web:

# replace username/repo:tag with your name and image details

image: **asmith/repo:friendlyhello**

deploy:

replicas: 5

resources:

limits:

cpus: "0.1"

memory: 50M

restart\_policy:

condition: on-failure

ports:

- "4000:80"

networks:

- webnet

networks:

webnet:

This docker-compose.yml file tells Docker to do the following:

* Pull [the image we uploaded in step 2](https://docs.docker.com/get-started/part2/) from the registry.
* Run 5 instances of that image as a service called web, limiting each one to use, at most, 10% of the CPU (across all cores), and 50MB of RAM.
* Immediately restart containers if one fails.
* Map port 4000 on the host to web’s port 80.
* Instruct web’s containers to share port 80 via a load-balanced network called webnet. (Internally, the containers themselves publish to web’s port 80 at an ephemeral port.)
* Define the webnet network with the default settings (which is a load-balanced overlay network).

# Run your new load-balanced app

Before we can use the docker stack deploy command we first run:

**docker swarm init**

**Note**: We get into the meaning of that command in [part 4](https://docs.docker.com/get-started/part4/). If you don’t run docker swarm init you get an error that “this node is not a swarm manager.”

Now let’s run it. You need to give your app a name. Here, it is set to getstartedlab:

**docker stack deploy -c docker-compose.yml getstartedlab**

**For example**

**docker stack deploy -c docker-compose.yml friendlyhello**

Our single service stack is running 5 container instances of our deployed image on one host. Let’s investigate.

Get the service ID for the one service in our application:

**docker service ls**

Look for output for the web service, prepended with your app name. If you named it the same as shown in this example, the name is getstartedlab\_web. The service ID is listed as well, along with the number of replicas, image name, and exposed ports.

A single container running in a service is called a **task**. Tasks are given unique IDs that numerically increment, up to the number of replicas you defined in docker-compose.yml. List the tasks for your service:

**docker service ps getstartedlab\_web**

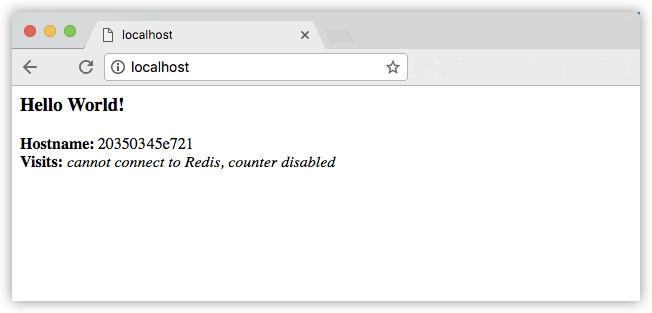
**For example**

**docker service ps friendlyhello**

Tasks also show up if you just list all the containers on your system, though that is not filtered by service:

**docker container ls -q**

You can run **curl -4 http://localhost:4000** several times in a row, or go to that URL in your browser and hit refresh a few times.



**Either way, the container ID changes**, demonstrating the load-balancing; with each request, one of the 5 tasks is chosen, in a round-robin fashion, to respond. The container IDs match your output from the previous command (docker container ls -q).

**Running Windows 10?**

Windows 10 PowerShell should already have curl available, but if not you can grab a Linux terminal emulator like [Git BASH](https://git-for-windows.github.io/), or download [wget for Windows](http://gnuwin32.sourceforge.net/packages/wget.htm) which is very similar.

**Slow response times?**

Depending on your environment’s networking configuration, it may take up to 30 seconds for the containers to respond to HTTP requests. This is not indicative of Docker or swarm performance, but rather an unmet Redis dependency that we address later in the tutorial. For now, the visitor counter isn’t working for the same reason; we haven’t yet added a service to persist data.

Scale the app

#vi docker-compose.yml

Change the number of replicas to 8

You can scale the app by changing the replicas value in docker-compose.yml, saving the change, and re-running the docker stack deploy command:

**docker stack deploy -c docker-compose.yml getstartedlab**

**For example**

**docker stack deploy -c docker-compose.yml friendlyhello**

**docker service ls**

Docker performs an in-place update, no need to tear the stack down first or kill any containers.

Now, re-run docker container ls -q to see the deployed instances reconfigured. If you scaled up the replicas, more tasks, and hence, more containers, are started.

Take down the app and the swarm

Take the app down with docker stack rm:

* **docker stack rm getstartedlab**

**For example**

**docker stack rm friendlyhello**

* Take down the swarm.
* **docker swarm leave --force**

It’s as easy as that to stand up and scale your app with Docker. You’ve taken a huge step towards learning how to run containers in production. Up next, you learn how to run this app as a bonafide swarm on a cluster of Docker machines.

**Note**: Compose files like this are used to define applications with Docker, and can be uploaded to cloud providers using [Docker Cloud](https://docs.docker.com/docker-cloud/), or on any hardware or cloud provider you choose with[Docker Enterprise Edition](https://www.docker.com/enterprise-edition).

[On to “Part 4” >>](https://docs.docker.com/get-started/part4/)

Recap and cheat sheet (optional)

Here’s [a terminal recording of what was covered on this page](https://asciinema.org/a/b5gai4rnflh7r0kie01fx6lip):

To recap, while typing docker run is simple enough, the true implementation of a container in production is running it as a service. Services codify a container’s behavior in a Compose file, and this file can be used to scale, limit, and redeploy our app. Changes to the service can be applied in place, as it runs, using the same command that launched the service: docker stack deploy.

Some commands to explore at this stage:

docker stack ls # List stacks or apps

docker stack deploy -c <composefile> <appname> # Run the specified Compose file

docker service ls # List running services associated with an app

docker service ps <service> # List tasks associated with an app

docker inspect <task or container> # Inspect task or container

docker container ls -q # List container IDs

docker stack rm <appname> # Tear down an application

docker swarm leave --force # Take down a single node swarm from the manager

[services](https://docs.docker.com/glossary/?term=services), [replicas](https://docs.docker.com/glossary/?term=replicas), [scale](https://docs.docker.com/glossary/?term=scale), [ports](https://docs.docker.com/glossary/?term=ports), [compose](https://docs.docker.com/glossary/?term=compose), [compose file](https://docs.docker.com/glossary/?term=compose%20file), [stack](https://docs.docker.com/glossary/?term=stack), [networking](https://docs.docker.com/glossary/?term=networking)

# Get Started, Part 4: Swarms

* [1: Orientation](https://docs.docker.com/get-started/part1)
* [2: Containers](https://docs.docker.com/get-started/part2)
* [3: Services](https://docs.docker.com/get-started/part3)
* [4: Swarms](https://docs.docker.com/get-started/part4)
* [5: Stacks](https://docs.docker.com/get-started/part5)
* [6: Deploy your app](https://docs.docker.com/get-started/part6)

Prerequisites

* [Install Docker version 1.13 or higher](https://docs.docker.com/engine/installation/).
* Get [Docker Compose](https://docs.docker.com/compose/overview/) as described in [Part 3 prerequisites](https://docs.docker.com/get-started/part3/#prerequisites).
* Get [Docker Machine](https://docs.docker.com/machine/overview/), which is pre-installed with [Docker Desktop for Mac](https://docs.docker.com/docker-for-mac/) and [Docker Desktop for Windows](https://docs.docker.com/docker-for-windows/), but on Linux systems you need to [install it directly](https://docs.docker.com/machine/install-machine/#installing-machine-directly). On pre Windows 10 systems *without Hyper-V*, as well as Windows 10 Home, use [Docker Toolbox](https://docs.docker.com/toolbox/overview/).
* Read the orientation in [Part 1](https://docs.docker.com/get-started/).
* Learn how to create containers in [Part 2](https://docs.docker.com/get-started/part2/).
* Make sure you have published the friendlyhello image you created by [pushing it to a registry](https://docs.docker.com/get-started/part2/#share-your-image). We use that shared image here.
* Be sure your image works as a deployed container. Run this command, slotting in your info for username, repo, and tag: docker run -p 80:80 username/repo:tag, then visit http://localhost/.
* Have a copy of your docker-compose.yml from [Part 3](https://docs.docker.com/get-started/part3/) handy.

Introduction

In [part 3](https://docs.docker.com/get-started/part3/), you took an app you wrote in [part 2](https://docs.docker.com/get-started/part2/), and defined how it should run in production by turning it into a service, scaling it up 5x in the process.

Here in part 4, you deploy this application onto a cluster, running it on multiple machines. Multi-container, multi-machine applications are made possible by joining multiple machines into a “Dockerized” cluster called a **swarm**.

Understanding Swarm clusters

A swarm is a group of machines that are running Docker and joined into a cluster. After that has happened, you continue to run the Docker commands you’re used to, but now they are executed on a cluster by a **swarm manager**. The machines in a swarm can be physical or virtual. After joining a swarm, they are referred to as **nodes**.

Swarm managers can use several strategies to run containers, such as “emptiest node” -- which fills the least utilized machines with containers. Or “global”, which ensures that each machine gets exactly one instance of the specified container. You instruct the swarm manager to use these strategies in the Compose file, just like the one you have already been using.

Swarm managers are the only machines in a swarm that can execute your commands, or authorize other machines to join the swarm as **workers**. Workers are just there to provide capacity and do not have the authority to tell any other machine what it can and cannot do.

Up until now, you have been using Docker in a single-host mode on your local machine. But Docker also can be switched into **swarm mode**, and that’s what enables the use of swarms. Enabling swarm mode instantly makes the current machine a swarm manager. From then on, Docker runs the commands you execute on the swarm you’re managing, rather than just on the current machine.

Set up your swarm

A swarm is made up of multiple nodes, which can be either physical or virtual machines. The basic concept is simple enough: run docker swarm init to enable swarm mode and make your current machine a swarm manager, then run docker swarm join on other machines to have them join the swarm as workers. Choose a tab below to see how this plays out in various contexts. We use VMs to quickly create a two-machine cluster and turn it into a swarm.

# Getting started with swarm mode

This tutorial introduces you to the features of Docker Engine Swarm mode. You may want to familiarize yourself with the [key concepts](https://docs.docker.com/engine/swarm/key-concepts/) before you begin.

The tutorial guides you through the following activities:

* initializing a cluster of Docker Engines in swarm mode
* adding nodes to the swarm
* deploying application services to the swarm
* managing the swarm once you have everything running

This tutorial uses Docker Engine CLI commands entered on the command line of a terminal window.

If you are brand new to Docker, see [About Docker Engine](https://docs.docker.com/engine/).

## Set up

To run this tutorial, you need the following:

* [three Linux hosts which can communicate over a network, with Docker installed](https://docs.docker.com/engine/swarm/swarm-tutorial/#three-networked-host-machines)
* [Docker Engine 1.12 or later installed](https://docs.docker.com/engine/swarm/swarm-tutorial/#docker-engine-1-12-or-newer)
* [the IP address of the manager machine](https://docs.docker.com/engine/swarm/swarm-tutorial/#the-ip-address-of-the-manager-machine)
* [open ports between the hosts](https://docs.docker.com/engine/swarm/swarm-tutorial/#open-protocols-and-ports-between-the-hosts)

### **Three networked host machines**

This tutorial requires three Linux hosts which have Docker installed and can communicate over a network. These can be physical machines, virtual machines, Amazon EC2 instances, or hosted in some other way. You can even use Docker Machine from a Linux, Mac, or Windows host. Check out [Getting started - Swarms](https://docs.docker.com/get-started/part4/#prerequisites) for one possible set-up for the hosts.

One of these machines is a manager (called manager1) and two of them are workers (worker1 and worker2).

**Note**: You can follow many of the tutorial steps to test single-node swarm as well, in which case you need only one host. Multi-node commands do not work, but you can initialize a swarm, create services, and scale them.

### **Docker Engine 1.12 or newer**

This tutorial requires Docker Engine 1.12 or newer on each of the host machines. Install Docker Engine and verify that the Docker Engine daemon is running on each of the machines. You can get the latest version of Docker Engine as follows:

* [install Docker Engine on Linux machines](https://docs.docker.com/engine/swarm/swarm-tutorial/#install-docker-engine-on-linux-machines)
* [use Docker Desktop for Mac or Docker Desktop for Windows](https://docs.docker.com/engine/swarm/swarm-tutorial/#use-docker-for-mac-or-docker-for-windows)

#### **INSTALL DOCKER ENGINE ON LINUX MACHINES**

If you are using Linux based physical computers or cloud-provided computers as hosts, simply follow the [Linux install instructions](https://docs.docker.com/engine/installation/) for your platform. Spin up the three machines, and you are ready. You can test both single-node and multi-node swarm scenarios on Linux machines.

#### **USE DOCKER DESKTOP FOR MAC OR DOCKER DESKTOP FOR WINDOWS**

Alternatively, install the latest [Docker Desktop for Mac](https://docs.docker.com/docker-for-mac/) or [Docker Desktop for Windows](https://docs.docker.com/docker-for-windows/) application on one computer. You can test both single-node and multi-node swarm from this computer, but you need to use Docker Machine to test the multi-node scenarios.

* You can use Docker Desktop for Mac or Windows to test single-node features of swarm mode, including initializing a swarm with a single node, creating services, and scaling services. Docker “Moby” on Hyperkit (Mac) or Hyper-V (Windows) serve as the single swarm node.
* Currently, you cannot use Docker Desktop for Mac or Docker Desktop for Windows alone to test amulti-node swarm. However, you can use the included version of [Docker Machine](https://docs.docker.com/machine/overview/) to create the swarm nodes (see [Get started with Docker Machine and a local VM](https://docs.docker.com/machine/get-started/)), then follow the tutorial for all multi-node features. For this scenario, you run commands from a Docker Desktop for Mac or Docker Desktop for Windows host, but that Docker host itself is not participating in the swarm. After you create the nodes, you can run all swarm commands as shown from the Mac terminal or Windows PowerShell with Docker Desktop for Mac or Docker Desktop for Windows running.

### **The IP address of the manager machine**

The IP address must be assigned to a network interface available to the host operating system. All nodes in the swarm need to connect to the manager at the IP address.

Because other nodes contact the manager node on its IP address, you should use a fixed IP address.

You can run ifconfig on Linux or macOS to see a list of the available network interfaces.

If you are using Docker Machine, you can get the manager IP with either docker-machine ls or docker-machine ip <MACHINE-NAME> — for example, docker-machine ip manager1.

The tutorial uses manager1 : 192.168.99.100.

### **Open protocols and ports between the hosts**

The following ports must be available. On some systems, these ports are open by default.

* **TCP port 2377** for cluster management communications
* **TCP** and **UDP port 7946** for communication among nodes
* **UDP port 4789** for overlay network traffic

If you plan on creating an overlay network with encryption (--opt encrypted), you also need to ensure **ip protocol 50** (**ESP**) traffic is allowed.

## What’s next?

After you have set up your environment, you are ready to [create a swarm](https://docs.docker.com/engine/swarm/swarm-tutorial/create-swarm/).

[tutorial](https://docs.docker.com/glossary/?term=tutorial), [cluster management](https://docs.docker.com/glossary/?term=cluster%20management), [swarm mode](https://docs.docker.com/glossary/?term=swarm%20mode)

In order to use the docker machine later on in this tutorial we need to install it

# Create a swarm

After you complete the [tutorial setup](https://docs.docker.com/engine/swarm/swarm-tutorial/) steps, you’re ready to create a swarm. Make sure the Docker Engine daemon is started on the host machines.

1. Open a terminal and ssh into the machine where you want to run your manager node. This tutorial uses a machine named manager1. If you use Docker Machine, you can connect to it via SSH using the following command:
2. $ docker-machine ssh manager1

docker-machine ssh 10.64.71.1 (the ip of the manager)

1. Run the following command to create a new swarm:
2. $ docker swarm init --advertise-addr <MANAGER-IP>

**Note**: If you are using Docker Desktop for Mac or Docker Desktop for Windows to test single-node swarm, simply run docker swarm init with no arguments. There is no need to specify --advertise-addr in this case. To learn more, see the topic on how to [Use Docker Desktop or Mac or Docker Desktop for Windows](https://docs.docker.com/engine/swarm/swarm-tutorial/#use-docker-for-mac-or-docker-for-windows) with Swarm.

In the tutorial, the following command creates a swarm on the manager1 machine:

$ docker swarm init --advertise-addr 192.168.99.100

For example

docker swarm init --advertise-addr 10.64.81.1

* **TCP port 2377** for cluster management communications
* **TCP** and **UDP port 7946** for communication among nodes
* **UDP port 4789** for overlay network traffic

Open the ports on all members of the swarm

firewall-cmd --zone=public --permanent --add-port=2377/tcp

firewall-cmd --zone=public --permanent --add-port=7946/tcp

firewall-cmd --zone=public --permanent --add-port=7946/udp

firewall-cmd --zone=public --permanent --add-port=4789/udp

**on the manager**

docker swarm init --advertise-addr 10.64.71.1

Swarm initialized: current node (dxn1zf6l61qsb1josjja83ngz) is now a manager.

To add a worker to this swarm, run the following command:

**Copy command and the token into memory**

**Add your partner to the swarm**

Use putty to connect to the partner in the swarm for example 10.64.72.1

Paste below into the partner

$su

#

docker swarm join \

--token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxv8rssmk743ojnwacrr2e7c \

10.64.81.1:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

The --advertise-addr flag configures the manager node to publish its address as 192.168.99.100. The other nodes in the swarm must be able to access the manager at the IP address.

The output includes the commands to join new nodes to the swarm. Nodes will join as managers or workers depending on the value for the --token flag.

1. Run docker info to view the current state of the swarm:

$ docker info

Containers: 2

Running: 0

Paused: 0

=Stopped: 2

...snip...

Swarm: active

NodeID: dxn1zf6l61qsb1josjja83ngz

Is Manager: true

Managers: 1

Nodes: 1

...snip...

1. Run the docker node ls command to view information about nodes:

docker node ls

ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS

dxn1zf6l61qsb1josjja83ngz \* manager1 Ready Active Leader

The \* next to the node ID indicates that you’re currently connected on this node.

Docker Engine swarm mode automatically names the node for the machine host name. The tutorial covers other columns in later steps.

## What’s next?

In the next section of the tutorial, we [add two more nodes](https://docs.docker.com/engine/swarm/swarm-tutorial/add-nodes/) to the cluster.

[tutorial](https://docs.docker.com/glossary/?term=tutorial), [cluster management](https://docs.docker.com/glossary/?term=cluster%20management), [swarm mode](https://docs.docker.com/glossary/?term=swarm%20mode)

**Leaving a swarm**

If you want to start over, you can run

**docker swarm leave**

from each node.

Deploy your app on the swarm cluster

The hard part is over. Now you just repeat the process you used in [part 3](https://docs.docker.com/get-started/part3/) to deploy on your new swarm. Just remember that only swarm managers like myvm1 execute Docker commands; workers are just for capacity.

# Deploy the app on the swarm manager

Lets make sure what we did previously has been stopped on all three docker hosts

Take down the app and the swarm

#docker ps

#docker stop containerid

Now that you have myvm1, you can use its powers as a swarm manager to deploy your app by using the same docker stack deploy command you used in part 3 to myvm1, and your local copy of docker-compose.yml.. This command may take a few seconds to complete and the deployment takes some time to be available. Use the docker service ps <service\_name> command on a swarm manager to verify that all services have been redeployed.

You are connected to myvm1 by means of the docker-machine shell configuration, and you still have access to the files on your local host. Make sure you are in the same directory as before, which includes the [docker-compose.yml file you created in part 3](https://docs.docker.com/get-started/part3/#docker-composeyml).

Just like before, run the following command to deploy the app on myvm1.

**#cd /ex3**

**#docker stack deploy -c docker-compose.yml friendlyhello**

**To verify it is running on**

**#docker ps**

And that’s it, the app is deployed on a swarm cluster!

**Note: If your image is stored on a private registry instead of Docker Hub, you need to be logged in using docker login <your-registry> and then you need to add the --with-registry-auth flag to the above command. For example:**

docker login registry.example.com

docker stack deploy --with-registry-auth -c docker-compose.yml getstartedlab

This passes the login token from your local client to the swarm nodes where the service is deployed, using the encrypted WAL logs. With this information, the nodes are able to log into the registry and pull the image.

Now you can use the same [docker commands you used in part 3](https://docs.docker.com/get-started/part3/#run-your-new-load-balanced-app). Only this time notice that the services (and associated containers) have been distributed between both myvm1 and myvm2.

$ docker stack ps friendlyhello

ID NAME IMAGE NODE DESIRED STATE

jq2g3qp8nzwx getstartedlab\_web.1 gordon/get-started:part2 myvm1 Running

88wgshobzoxl getstartedlab\_web.2 gordon/get-started:part2 myvm2 Running

vbb1qbkb0o2z getstartedlab\_web.3 gordon/get-started:part2 myvm2 Running

ghii74p9budx getstartedlab\_web.4 gordon/get-started:part2 myvm1 Running

0prmarhavs87 getstartedlab\_web.5 gordon/get-started:part2 myvm2 Running

**Connecting to VMs with docker-machine env and docker-machine ssh**

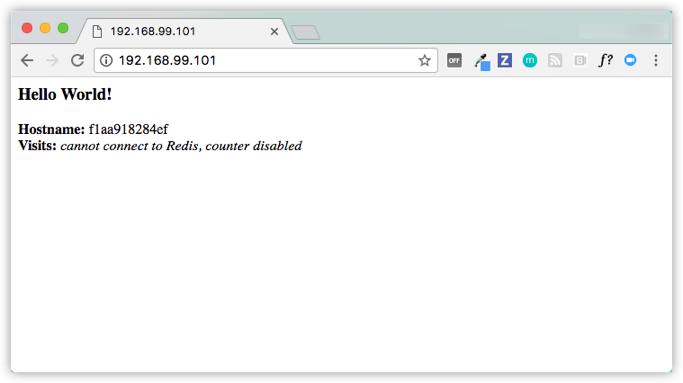
* To set your shell to talk to a different machine like myvm2, simply re-rundocker-machine env in the same or a different shell, then run the given command to point to myvm2. This is always specific to the current shell. If you change to an unconfigured shell or open a new one, you need to re-run the commands. Use docker-machine ls to list machines, see what state they are in, get IP addresses, and find out which one, if any, you are connected to. To learn more, see the [Docker Machine getting started topics](https://docs.docker.com/machine/get-started/#create-a-machine).
* Alternatively, you can wrap Docker commands in the form ofdocker-machine ssh <machine> "<command>", which logs directly into the VM but doesn’t give you immediate access to files on your local host.
* On Mac and Linux, you can use docker-machine scp <file> <machine>:~ to copy files across machines, but Windows users need a Linux terminal emulator like [Git Bash](https://git-for-windows.github.io/) for this to work.

This tutorial demos both docker-machine ssh and docker-machine env, since these are available on all platforms via the docker-machine CLI.

Accessing your cluster

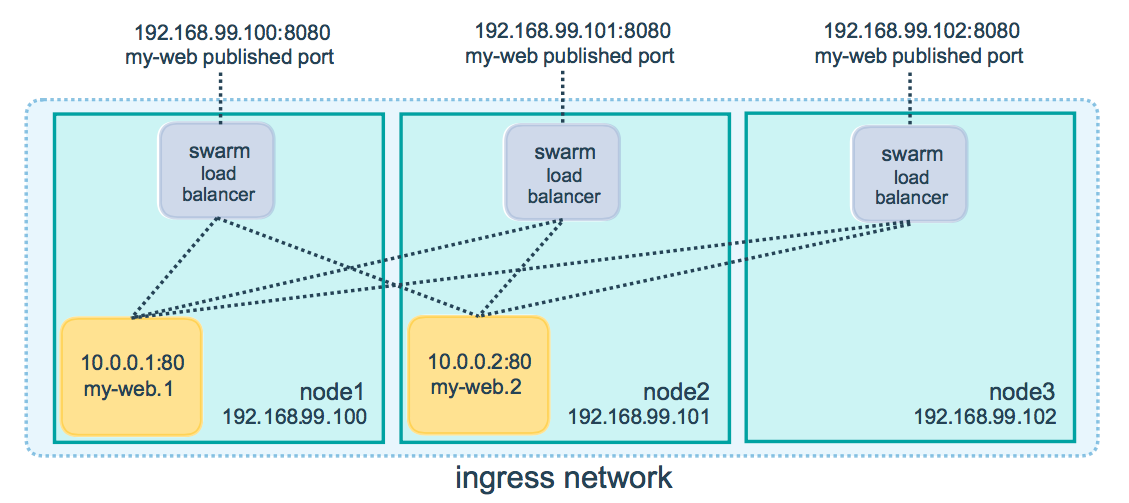
You can access your app from the IP address of **either** myvm1 or myvm2.

The network you created is shared between them and load-balancing. Run docker-machine ls to get your VMs’ IP addresses and visit either of them on a browser, hitting refresh (or just curl them).



There are five possible container IDs all cycling by randomly, demonstrating the load-balancing.

The reason both IP addresses work is that nodes in a swarm participate in an ingress **routing mesh**. This ensures that a service deployed at a certain port within your swarm always has that port reserved to itself, no matter what node is actually running the container. Here’s a diagram of how a routing mesh for a service called my-web published at port 8080 on a three-node swarm would look:



**Having connectivity trouble?**

Keep in mind that to use the ingress network in the swarm, you need to have the following ports open between the swarm nodes before you enable swarm mode:

* Port 7946 TCP/UDP for container network discovery.
* Port 4789 UDP for the container ingress network.

Iterating and scaling your app

From here you can do everything you learned about in parts 2 and 3.

Scale the app by changing the docker-compose.yml file.

Change the app behavior by editing code, then rebuild, and push the new image. (To do this, follow the same steps you took earlier to [build the app](https://docs.docker.com/get-started/part2/#build-the-app) and [publish the image](https://docs.docker.com/get-started/part2/#publish-the-image)).

In either case, simply run docker stack deploy again to deploy these changes.

You can join any machine, physical or virtual, to this swarm, using the same docker swarm joincommand you used on myvm2, and capacity is added to your cluster. Just run docker stack deployafterwards, and your app can take advantage of the new resources.

Cleanup and reboot

Stacks and swarms

You can tear down the stack with docker stack rm. For example:

docker stack rm friendlyhello

**Keep the swarm or remove it?**

At some point later, you can remove this swarm if you want to withdocker-machine ssh myvm2 "docker swarm leave" on the worker and docker-machine ssh myvm1 "docker swarm leave --force" on the manager, but *you need this swarm for part 5, so keep it around for now*.

Unsetting docker-machine shell variable settings

You can unset the docker-machine environment variables in your current shell with the given command.

On **Mac or Linux** the command is:

eval $(docker-machine env -u)

On **Windows** the command is:

& "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env -u | Invoke-Expression

This disconnects the shell from docker-machine created virtual machines, and allows you to continue working in the same shell, now using native docker commands (for example, on Docker Desktop for Mac or Docker Desktop for Windows). To learn more, see the [Machine topic on unsetting environment variables](https://docs.docker.com/machine/get-started/#unset-environment-variables-in-the-current-shell).

Restarting Docker machines

If you shut down your local host, Docker machines stops running. You can check the status of machines by running docker-machine ls.

$ docker-machine ls

NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS

myvm1 - virtualbox Stopped Unknown

myvm2 - virtualbox Stopped Unknown

To restart a machine that’s stopped, run:

docker-machine start <machine-name>

For example:

$ docker-machine start myvm1

Starting "myvm1"...

(myvm1) Check network to re-create if needed...

(myvm1) Waiting for an IP...

Machine "myvm1" was started.

Waiting for SSH to be available...

Detecting the provisioner...

Started machines may have new IP addresses. You may need to re-run the `docker-machine env` command.

$ docker-machine start myvm2

Starting "myvm2"...

(myvm2) Check network to re-create if needed...

(myvm2) Waiting for an IP...

Machine "myvm2" was started.

Waiting for SSH to be available...

Detecting the provisioner...

Started machines may have new IP addresses. You may need to re-run the `docker-machine env` command.

[On to Part 5 >>](https://docs.docker.com/get-started/part5/)

Recap and cheat sheet (optional)

Here’s [a terminal recording of what was covered on this page](https://asciinema.org/a/113837):

In part 4 you learned what a swarm is, how nodes in swarms can be managers or workers, created a swarm, and deployed an application on it. You saw that the core Docker commands didn’t change from part 3, they just had to be targeted to run on a swarm master. You also saw the power of Docker’s networking in action, which kept load-balancing requests across containers, even though they were running on different machines. Finally, you learned how to iterate and scale your app on a cluster.

Here are some commands you might like to run to interact with your swarm and your VMs a bit:

docker-machine create --driver virtualbox myvm1 # Create a VM (Mac, Win7, Linux)

docker-machine create -d hyperv --hyperv-virtual-switch "myswitch" myvm1 # Win10

docker-machine env myvm1 # View basic information about your node

docker-machine ssh myvm1 "docker node ls" # List the nodes in your swarm

docker-machine ssh myvm1 "docker node inspect <node ID>" # Inspect a node

docker-machine ssh myvm1 "docker swarm join-token -q worker" # View join token

docker-machine ssh myvm1 # Open an SSH session with the VM; type "exit" to end

docker node ls # View nodes in swarm (while logged on to manager)

docker-machine ssh myvm2 "docker swarm leave" # Make the worker leave the swarm

docker-machine ssh myvm1 "docker swarm leave -f" # Make master leave, kill swarm

docker-machine ls # list VMs, asterisk shows which VM this shell is talking to

docker-machine start myvm1 # Start a VM that is currently not running

docker-machine env myvm1 # show environment variables and command for myvm1

eval $(docker-machine env myvm1) # Mac command to connect shell to myvm1

& "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env myvm1 | Invoke-Expression # Windows command to connect shell to myvm1

docker stack deploy -c <file> <app> # Deploy an app; command shell must be set to talk to manager (myvm1), uses local Compose file

docker-machine scp docker-compose.yml myvm1:~ # Copy file to node's home dir (only required if you use ssh to connect to manager and deploy the app)

docker-machine ssh myvm1 "docker stack deploy -c <file> <app>" # Deploy an app using ssh (you must have first copied the Compose file to myvm1)

eval $(docker-machine env -u) # Disconnect shell from VMs, use native docker

docker-machine stop $(docker-machine ls -q) # Stop all running VMs

docker-machine rm $(docker-machine ls -q) # Delete all VMs and their disk images

[swarm](https://docs.docker.com/glossary/?term=swarm), [scale](https://docs.docker.com/glossary/?term=scale), [cluster](https://docs.docker.com/glossary/?term=cluster), [machine](https://docs.docker.com/glossary/?term=machine), [vm](https://docs.docker.com/glossary/?term=vm), [manager](https://docs.docker.com/glossary/?term=manager), [worker](https://docs.docker.com/glossary/?term=worker), [deploy](https://docs.docker.com/glossary/?term=deploy), [ssh](https://docs.docker.com/glossary/?term=ssh), [orchestration](https://docs.docker.com/glossary/?term=orchestration)

# Get Started, Part 5: Stacks

* [1: Orientation](https://docs.docker.com/get-started/part1)
* [2: Containers](https://docs.docker.com/get-started/part2)
* [3: Services](https://docs.docker.com/get-started/part3)
* [4: Swarms](https://docs.docker.com/get-started/part4)
* [5: Stacks](https://docs.docker.com/get-started/part5)
* [6: Deploy your app](https://docs.docker.com/get-started/part6)

## Prerequisites

* [Install Docker version 1.13 or higher](https://docs.docker.com/engine/installation/).
* Get [Docker Compose](https://docs.docker.com/compose/overview/) as described in [Part 3 prerequisites](https://docs.docker.com/get-started/part3/#prerequisites).
* Get [Docker Machine](https://docs.docker.com/machine/overview/) as described in [Part 4 prerequisites](https://docs.docker.com/get-started/part4/#prerequisites).
* Read the orientation in [Part 1](https://docs.docker.com/get-started/).
* Learn how to create containers in [Part 2](https://docs.docker.com/get-started/part2/).
* Make sure you have published the friendlyhello image you created by [pushing it to a registry](https://docs.docker.com/get-started/part2/#share-your-image). We use that shared image here.
* Be sure your image works as a deployed container. Run this command, slotting in your info for username, repo, and tag: docker run -p 80:80 username/repo:tag, then visit http://localhost/.
* Have a copy of your docker-compose.yml from [Part 3](https://docs.docker.com/get-started/part3/) handy.
* Make sure that the machines you set up in [part 4](https://docs.docker.com/get-started/part4/) are running and ready. Run docker-machine ls to verify this. If the machines are stopped, run docker-machine start myvm1 to boot the manager, followed by docker-machine start myvm2 to boot the worker.
* Have the swarm you created in [part 4](https://docs.docker.com/get-started/part4/) running and ready. Rundocker-machine ssh myvm1 "docker node ls" to verify this. If the swarm is up, both nodes report a ready status. If not, reinitialize the swarm and join the worker as described in [Set up your swarm](https://docs.docker.com/get-started/part4/#set-up-your-swarm).

## Introduction

In [part 4](https://docs.docker.com/get-started/part4/), you learned how to set up a swarm, which is a cluster of machines running Docker, and deployed an application to it, with containers running in concert on multiple machines.

Here in part 5, you reach the top of the hierarchy of distributed applications: the **stack**. A stack is a group of interrelated services that share dependencies, and can be orchestrated and scaled together. A single stack is capable of defining and coordinating the functionality of an entire application (though very complex applications may want to use multiple stacks).

Some good news is, you have technically been working with stacks since part 3, when you created a Compose file and used docker stack deploy. But that was a single service stack running on a single host, which is not usually what takes place in production. Here, you can take what you’ve learned, make multiple services relate to each other, and run them on multiple machines.

## Add a new service and redeploy

It’s easy to add services to our docker-compose.yml file. First, let’s add a free visualizer service that lets us look at how our swarm is scheduling containers.

1. Open up docker-compose.yml in an editor and replace its contents with the following. Be sure to replace username/repo:tag with your image details.

version: "3"

services:

web:

# replace username/repo:tag with your name and image details

image: username/repo:tag

deploy:

replicas: 5

restart\_policy:

condition: on-failure

resources:

limits:

cpus: "0.1"

memory: 50M

ports:

- "80:80"

networks:

- webnet

visualizer:

image: dockersamples/visualizer:stable

ports:

- "8080:8080"

volumes:

- "/var/run/docker.sock:/var/run/docker.sock"

deploy:

placement:

constraints: [node.role == manager]

networks:

- webnet

networks:

webnet:

The only thing new here is the peer service to web, named visualizer. Notice two new things here: a volumes key, giving the visualizer access to the host’s socket file for Docker, and a placement key, ensuring that this service only ever runs on a swarm manager -- never a worker. That’s because this container, built from [an open source project created by Docker](https://github.com/ManoMarks/docker-swarm-visualizer), displays Docker services running on a swarm in a diagram.

We talk more about placement constraints and volumes in a moment.

1. Make sure your shell is configured to talk to myvm1 (full examples are [here](https://docs.docker.com/get-started/part4/#configure-a-docker-machine-shell-to-the-swarm-manager)).
   * Run docker-machine ls to list machines and make sure you are connected to myvm1, as indicated by an asterisk next to it.
   * If needed, re-run docker-machine env myvm1, then run the given command to configure the shell.

On **Mac or Linux** the command is:

eval $(docker-machine env myvm1)

On **Windows** the command is:

& "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env myvm1 | Invoke-Expression

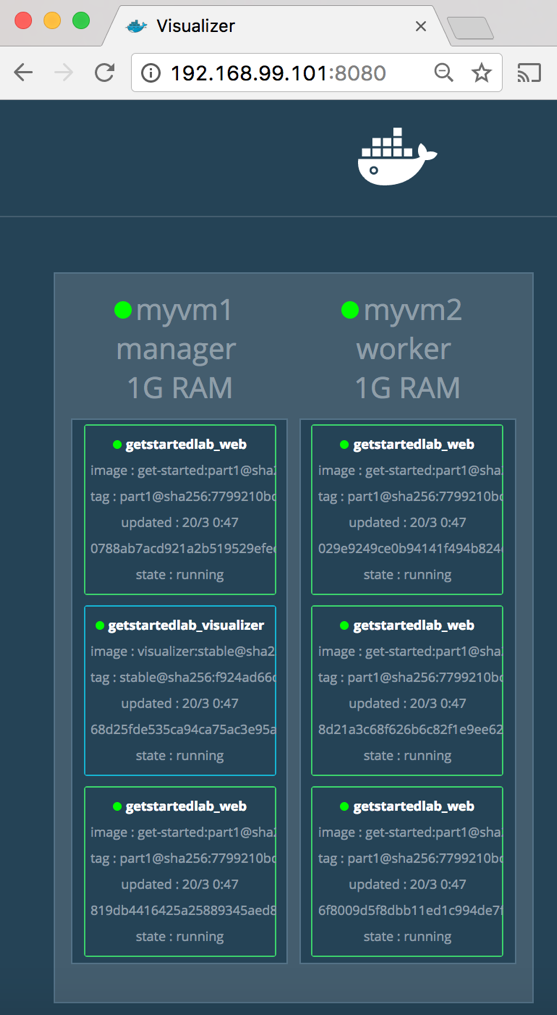
1. Re-run the docker stack deploy command on the manager, and whatever services need updating are updated:
2. $ docker stack deploy -c docker-compose.yml getstartedlab

Updating service getstartedlab\_web (id: angi1bf5e4to03qu9f93trnxm)

Creating service getstartedlab\_visualizer (id: l9mnwkeq2jiononb5ihz9u7a4)

1. Take a look at the visualizer.

You saw in the Compose file that visualizer runs on port 8080. Get the IP address of one of your nodes by running docker-machine ls. Go to either IP address at port 8080 and you can see the visualizer running:



The single copy of visualizer is running on the manager as you expect, and the 5 instances of web are spread out across the swarm. You can corroborate this visualization by running docker stack ps <stack>:

docker stack ps getstartedlab

The visualizer is a standalone service that can run in any app that includes it in the stack. It doesn’t depend on anything else. Now let’s create a service that does have a dependency: the Redis service that provides a visitor counter.

## Persist the data

Let’s go through the same workflow once more to add a Redis database for storing app data.

1. Save this new docker-compose.yml file, which finally adds a Redis service. Be sure to replace username/repo:tag with your image details.

version: "3"

services:

web:

# replace username/repo:tag with your name and image details

image: username/repo:tag

deploy:

replicas: 5

restart\_policy:

condition: on-failure

resources:

limits:

cpus: "0.1"

memory: 50M

ports:

- "80:80"

networks:

- webnet

visualizer:

image: dockersamples/visualizer:stable

ports:

- "8080:8080"

volumes:

- "/var/run/docker.sock:/var/run/docker.sock"

deploy:

placement:

constraints: [node.role == manager]

networks:

- webnet

redis:

image: redis

ports:

- "6379:6379"

volumes:

- "/home/docker/data:/data"

deploy:

placement:

constraints: [node.role == manager]

command: redis-server --appendonly yes

networks:

- webnet

networks:

webnet:

Redis has an official image in the Docker library and has been granted the short image name of just redis, so no username/repo notation here. The Redis port, 6379, has been pre-configured by Redis to be exposed from the container to the host, and here in our Compose file we expose it from the host to the world, so you can actually enter the IP for any of your nodes into Redis Desktop Manager and manage this Redis instance, if you so choose.

Most importantly, there are a couple of things in the redis specification that make data persist between deployments of this stack:

* + redis always runs on the manager, so it’s always using the same filesystem.
  + redis accesses an arbitrary directory in the host’s file system as /data inside the container, which is where Redis stores data.

Together, this is creating a “source of truth” in your host’s physical filesystem for the Redis data. Without this, Redis would store its data in /data inside the container’s filesystem, which would get wiped out if that container were ever redeployed.

This source of truth has two components:

* + The placement constraint you put on the Redis service, ensuring that it always uses the same host.
  + The volume you created that lets the container access ./data (on the host) as /data(inside the Redis container). While containers come and go, the files stored on ./data on the specified host persists, enabling continuity.

You are ready to deploy your new Redis-using stack.

1. Create a ./data directory on the manager:
2. docker-machine ssh myvm1 "mkdir ./data"

Or in our case

#mkdir ./data

1. Make sure your shell is configured to talk to myvm1 (full examples are [here](https://docs.docker.com/get-started/part4/#configure-a-docker-machine-shell-to-the-swarm-manager)).
   * Run docker-machine ls to list machines and make sure you are connected to myvm1, as indicated by an asterisk next to it.
   * If needed, re-run docker-machine env myvm1, then run the given command to configure the shell.

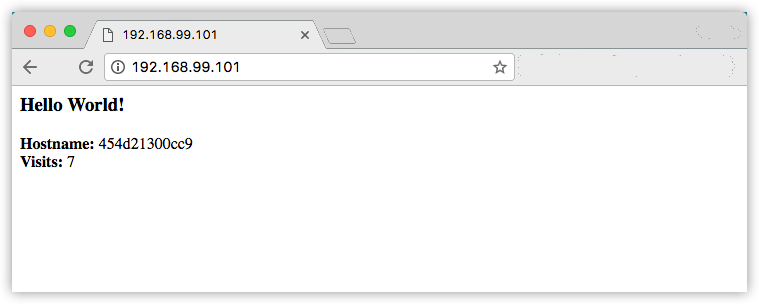
On **Mac or Linux** the command is:

eval $(docker-machine env myvm1)

On **Windows** the command is:

& "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env myvm1 | Invoke-Expression

1. Run docker stack deploy one more time.
2. $ docker stack deploy -c docker-compose.yml friendlyhello
3. Run docker service ls to verify that the three services are running as expected.
4. $ docker service ls
5. ID NAME MODE REPLICAS IMAGE PORTS
6. x7uij6xb4foj getstartedlab\_redis replicated 1/1 redis:latest \*:6379->6379/tcp
7. n5rvhm52ykq7 getstartedlab\_visualizer replicated 1/1 dockersamples/visualizer:stable \*:8080->8080/tcp
8. mifd433bti1d getstartedlab\_web replicated 5/5 gordon/getstarted:latest \*:80->80/tcp
9. Check the web page at one of your nodes, such as http://192.168.99.101, and take a look at the results of the visitor counter, which is now live and storing information on Redis.



Also, check the visualizer at port 8080 on either node’s IP address, and notice see the redis service running along with the web and visualizer services.



[On to Part 6 >>](https://docs.docker.com/get-started/part6/)

## Recap (optional)

Here’s [a terminal recording of what was covered on this page](https://asciinema.org/a/113840):

You learned that stacks are inter-related services all running in concert, and that -- surprise! -- you’ve been using stacks since part three of this tutorial. You learned that to add more services to your stack, you insert them in your Compose file. Finally, you learned that by using a combination of placement constraints and volumes you can create a permanent home for persisting data, so that your app’s data survives when the container is torn down and redeployed.

[stack](https://docs.docker.com/glossary/?term=stack), [data](https://docs.docker.com/glossary/?term=data), [persist](https://docs.docker.com/glossary/?term=persist), [dependencies](https://docs.docker.com/glossary/?term=dependencies), [redis](https://docs.docker.com/glossary/?term=redis), [storage](https://docs.docker.com/glossary/?term=storage), [volume](https://docs.docker.com/glossary/?term=volume), [port](https://docs.docker.com/glossary/?term=port)

Get Started, Part 6: Deploy your app

* [1: Orientation](https://docs.docker.com/get-started/part1)
* [2: Containers](https://docs.docker.com/get-started/part2)
* [3: Services](https://docs.docker.com/get-started/part3)
* [4: Swarms](https://docs.docker.com/get-started/part4)
* [5: Stacks](https://docs.docker.com/get-started/part5)
* [6: Deploy your app](https://docs.docker.com/get-started/part6)

Prerequisites

* [Install Docker](https://docs.docker.com/install/).
* Get [Docker Compose](https://docs.docker.com/compose/overview/) as described in [Part 3 prerequisites](https://docs.docker.com/get-started/part3/#prerequisites).
* Get [Docker Machine](https://docs.docker.com/machine/overview/) as described in [Part 4 prerequisites](https://docs.docker.com/get-started/part4/#prerequisites).
* Read the orientation in [Part 1](https://docs.docker.com/get-started/).
* Learn how to create containers in [Part 2](https://docs.docker.com/get-started/part2/).
* Make sure you have published the friendlyhello image you created by [pushing it to a registry](https://docs.docker.com/get-started/part2/#share-your-image). We use that shared image here.
* Be sure your image works as a deployed container. Run this command, slotting in your info for username, repo, and tag: docker run -p 80:80 username/repo:tag, then visit http://localhost/.
* Have [the final version of docker-compose.yml from Part 5](https://docs.docker.com/get-started/part5/#persist-the-data) handy.

Introduction

You’ve been editing the same Compose file for this entire tutorial. Well, we have good news. That Compose file works just as well in production as it does on your machine. In this section, we will go through some options for running your Dockerized application.

Choose an option

* [Docker Enterprise](https://docs.docker.com/get-started/part6/#enterprise)
* [Docker Engine - Community](https://docs.docker.com/get-started/part6/#community)

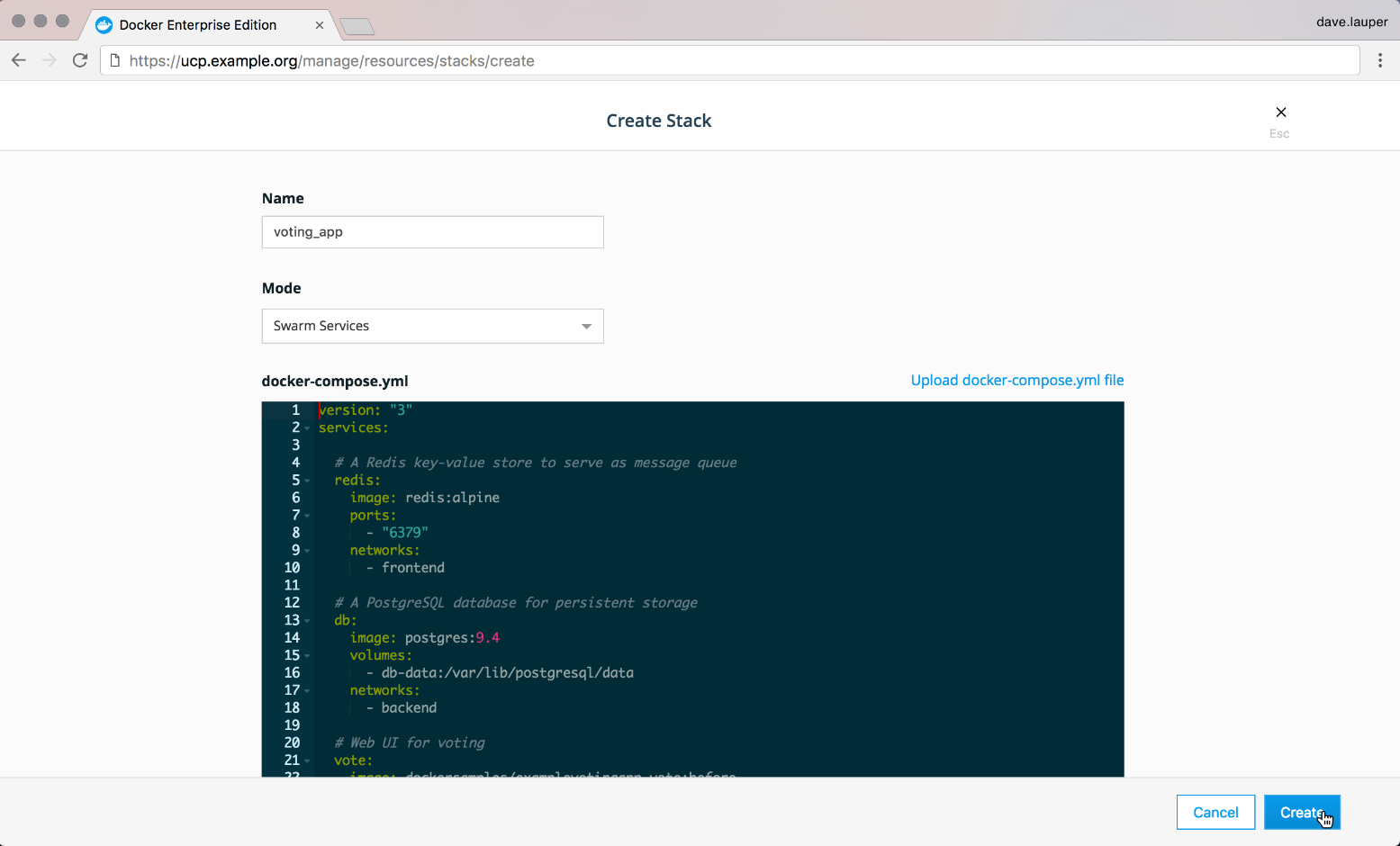
Customers of Docker Enterprise Edition run a stable, commercially-supported version of Docker Engine, and as an add-on they get our first-class management software, Docker Datacenter. You can manage every aspect of your application through the interface using Universal Control Plane, run a private image registry with Docker Trusted Registry, integrate with your LDAP provider, sign production images with Docker Content Trust, and many other features.

Bringing your own server to Docker Enterprise and setting up Docker Datacenter essentially involves two steps:

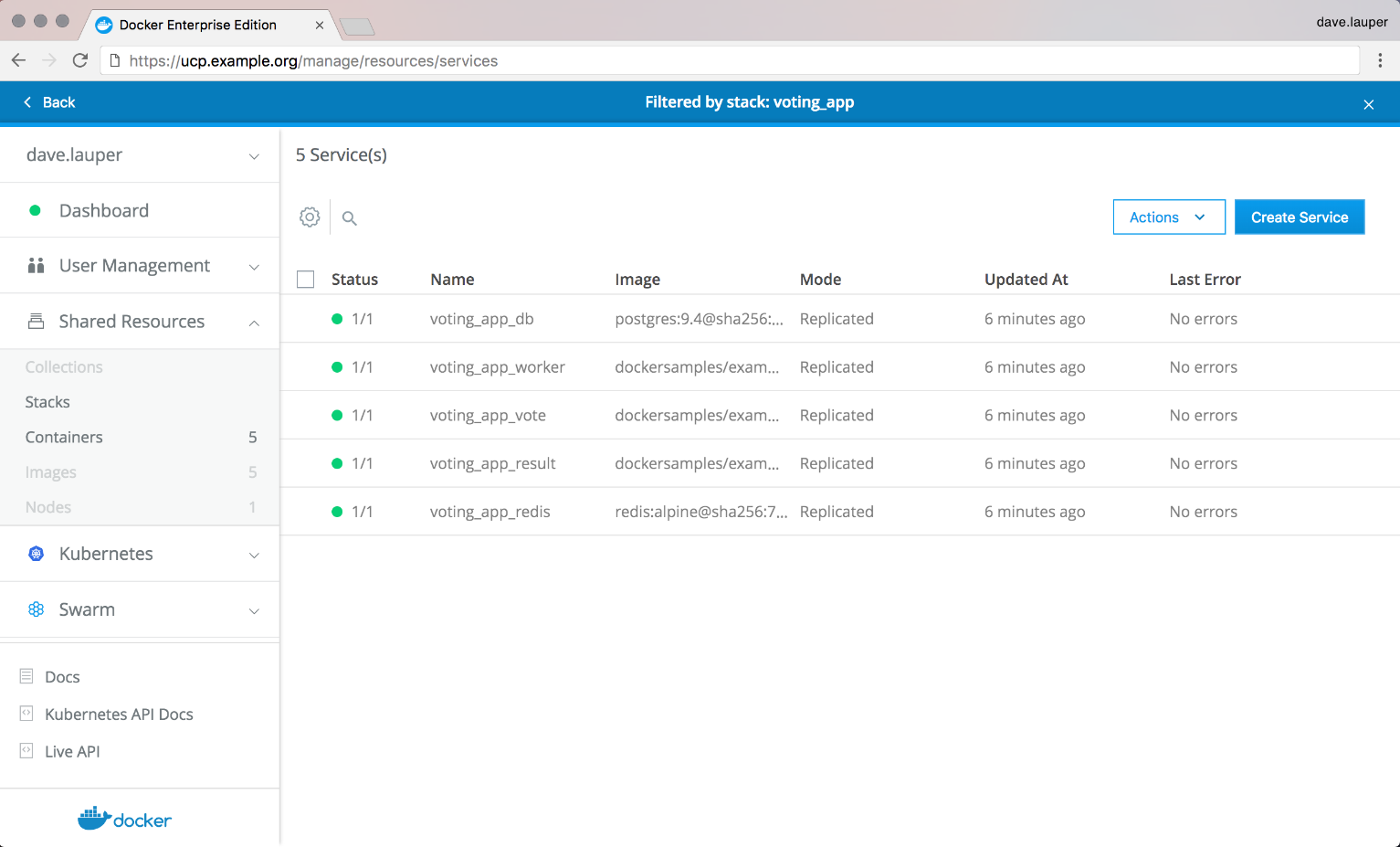
1. [Get Docker Enterprise for your server’s OS from Docker Hub](https://hub.docker.com/search?offering=enterprise&type=edition).
2. Follow the [instructions to install Docker Enterprise on your own host](https://docs.docker.com/datacenter/install/linux/).

**Note**: Running Windows containers? View our [Windows Server setup guide](https://docs.docker.com/install/windows/docker-ee/).

Once you’re all set up and Docker Enterprise is running, you can [deploy your Compose file from directly within the UI](https://docs.docker.com/ee/ucp/swarm/deploy-multi-service-app/).



After that, you can see it running, and can change any aspect of the application you choose, or even edit the Compose file itself.



Congratulations!

You’ve taken a full-stack, dev-to-deploy tour of the entire Docker platform.

There is much more to the Docker platform than what was covered here, but you have a good idea of the basics of containers, images, services, swarms, stacks, scaling, load-balancing, volumes, and placement constraints.

Want to go deeper? Here are some resources we recommend:

* [Samples](https://docs.docker.com/samples/): Our samples include multiple examples of popular software running in containers, and some good labs that teach best practices.
* [User Guide](https://docs.docker.com/engine/userguide/): The user guide has several examples that explain networking and storage in greater depth than was covered here.
* [Admin Guide](https://docs.docker.com/engine/admin/): Covers how to manage a Dockerized production environment.
* [Training](https://training.docker.com/): Official Docker courses that offer in-person instruction and virtual classroom environments.
* [Blog](https://blog.docker.com/): Covers what’s going on with Docker lately.

[deploy](https://docs.docker.com/glossary/?term=deploy), [production](https://docs.docker.com/glossary/?term=production), [datacenter](https://docs.docker.com/glossary/?term=datacenter), [cloud](https://docs.docker.com/glossary/?term=cloud), [aws](https://docs.docker.com/glossary/?term=aws), [azure](https://docs.docker.com/glossary/?term=azure), [provider](https://docs.docker.com/glossary/?term=provider), [admin](https://docs.docker.com/glossary/?term=admin), [enterprise](https://docs.docker.com/glossary/?term=enterprise)

# How to use Dockerfiles

If you're looking for the most efficient way of working with Docker images, use Dockerfiles.

With Docker images, you can easily deploy containers without having to worry about maintaining the same overhead required by virtual machines. These containers can be pulled from the [Docker Hub](https://hub.docker.com/) with ease, for easy use. Let's say, for example, you want to pull down the latest Ubuntu image for development purposes. However, before you start development, there are a number of changes you need to make to the image (such as updating, upgrading, and adding the build-essential package to the mix). Now, what if you plan on needing that same customized image for a number of development projects? You could always pull the latest Ubuntu image and then manually run the commands to update, upgrade, and install build-

Or you could make use of Dockerfiles.

A Dockerfile is a file used to build a Docker image to your specifics. With a Dockerfile constructed, you could then easily build the same image over and over, without having to walk through the process

I want to show you how to craft a Dockerfile so you can make the process of building a specific image a no-brainer. The image we will create is as described above: The latest image of Ubuntu, updated and upgraded, and with the build-essential package installed. This will be a fairly simple example of how Dockerfiles are used.

## The basics

The first thing you need to know is that the Dockerfile is a text file, named Dockerfile, that contains commands. The available commands are:

* ADD - copies the files from a source on the host into the container's own filesystem at the set destination
* CMD - can be used for executing a specific command within the container
* ENTRYPOINT - sets a default application to be used every time a container is created with the image
* ENV - sets environment variables
* EXPOSE - associates a specific port to enable networking between the container and the outside world
* FROM - defines the base image used to start the build process
* MAINTAINER - defines a full name and email address of the image creator
* RUN - central executing directive for Dockerfiles
* USER - sets the UID (or username) which is to run the container
* VOLUME - is used to enable access from the container to a directory on the host machine
* WORKDIR - sets the path where the command, defined with CMD, is to be executed

You do not have to use every command. In fact, I am going to demonstrate a Dockerfile using only FROM, MAINTAINER, and RUN.

The first thing we're going to do is create a new directory to work within; so open a terminal window and issue the command

1. mkdir ~/mydockerbuild. Change into that newly created directory with the command
2. cd ~/mydockerbuild.

Now we create our Dockerfile with the command *nano Dockerfile* and add the following contents:

FROM ubuntu:latest

MAINTAINER NAME EMAIL

RUN apt-get -y update && apt-get -y upgrade && apt-get install -y build-essential

Where NAME is the name to be used as the maintainer and EMAIL is the maintainer's email address.

## Example to be used in file

FROM ubuntu:latest

MAINTAINER student [student@fed1.com](mailto:student@fed1.com)

RUN apt-get -y update && apt-get -y upgrade && apt-get install -y build-essential

Save and close that file.

# Building the image

Now we build an image from our Dockerfile. This is run with the command (by a user in the docker group):

Syntax

docker build -t "name:Dockerfile" .

Example

docker build -t "myimage:Dockerfile" .

Where NAME is the name of the image to be built.

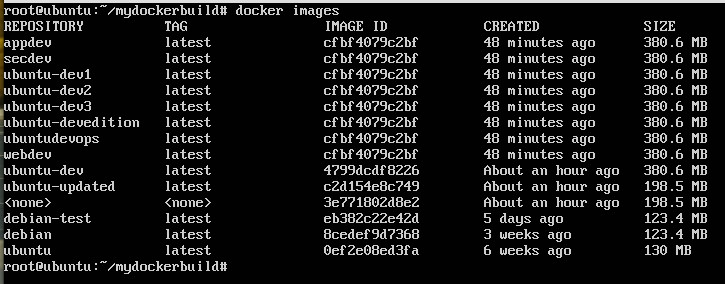
Say I want to build multiple images from this Dockerfile, one for web development, app development, and security development. I could build them with the following commands:

docker build -t "webdev:Dockerfile" .

docker build -t "appdev:Dockerfile" .

docker build -t "secdev:Dockerfile" .

Do note the trailing period is necessary for each command.

Once the build completes, you can issue the command docker images to see all of your newly created images 

All of my docker images conveniently listed.

And that is all there is to using Dockerfiles. Granted, this is a very simplistic illustration, but it will give you the springboard by which you can create more complicated Dockerfiles and docker images.

## Find out more

For more information on building and using Dockerfiles, I recommend reading [this best practices piece](https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/) from the official docker support page. Docker containers are a very powerful and flexible way to expand your devops and even the services and applications your company offers. Give Dockerfiles a try and see if they don't make the creation of images significantly more efficient.

See Instead

* [The Docker Ecosystem: An Introduction to Common Components](https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-an-introduction-to-common-components)
* [Dockerfile reference](https://docs.docker.com/engine/reference/builder/)
* [Best practices for writing Dockerfiles](https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/)

### Introduction

**Docker** containers are created by using base images. An image can be basic, with nothing but the operating-system fundamentals, or it can consist of a sophisticated pre-built application stack ready for launch.

When building your images with Docker, each action taken (i.e. a command executed such as apt-get install) forms a new layer on top of the previous one. These base images then can be used to create new containers.

In this DigitalOcean article, we will see about automating this process as much as possible, as well as demonstrate the best practices and methods to make most of Docker and containers via Dockerfiles: scripts to build containers, step-by-step, layer-by-layer, automatically from a base image.

## Glossary

### 1. Docker in Brief

### 2. Dockerfiles

### 3. Dockerfile Syntax

1. What is Syntax?
2. Dockerfile Syntax Example

### 4. Dockerfile Commands

* ADD
* CMD
* ENTRYPOINT
* ENV
* EXPOSE
* FROM
* MAINTAINER
* RUN
* USER
* VOLUME
* WORKDIR

### 5. How To Use Dockerfiles

# How to Setup DNS Server with Docker Container

A Dockerfile is used to create a Docker container image, this will be used to created the DNS Server. An easy way to setup a basic DNS server with Docker is to use the BIND DNS server bundled with the Webmin interface. In this article we cover how to implement DNS server using docker container.

BIND is an open source software that implements the Domain Name System (DNS) protocols for the Internet. It is a reference implementation of those protocols, but it is also production-grade software, suitable for use in high-volume and high-reliability applications.

Automated builds of the image are readily available on [Dockerhub](https://hub.docker.com/r/sameersbn/bind) and is the recommended method of installation

docker pull sameersbn/bind

Alternatively you can build the image yourself.

docker build -t sameersbn/bind github.com/sameersbn/docker-bind

To start a BIND DNS server you execute:

docker run --name bind -d --restart=always \

--publish 53:53/tcp --publish 53:53/udp --publish 10000:10000/tcp \

--volume /srv/docker/bind:/data \

sameersbn/bind

**To connect to it**

docker exec –it bind bash

**To connect to the container**

docker run --name bind -it --rm \

--publish 53:53/tcp --publish 53:53/udp --publish 10000:10000/tcp \

--volume /srv/docker/bind:/data \

sameersbn/bind: –h

Alternatively, you can use the sample docker-compose.yml file to start the container using [Docker Compose](https://linoxide.com/linux-how-to/build-run-apps-docker-compose/).

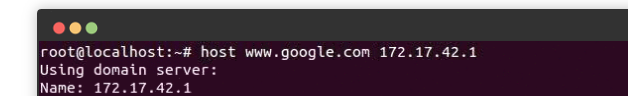
When the container is started the Webmin service is also started and is accessible from the web browser at http://localhost:10000. Login to Webmin with the username root and password password. Specify --env ROOT\_PASSWORD=secretpassword on the docker run command to set a password of your choosing.

The launch of Webmin can be disabled by adding --env WEBMIN\_ENABLED=false to the docker run command. Note that the ROOT\_PASSWORD parameter has no effect when the launch of Webmin is disabled.

**Testing DNS Server**

You can customize the launch command of BIND server by specifying arguments to named on the docker run command. For example the following command prints the help menu of named command:

docker run --name bind -it --rm \  
--publish 53:53/tcp --publish 53:53/udp --publish 10000:10000/tcp \  
--volume /srv/docker/bind:/data \  
sameersbn/bind –h



For the BIND to preserve its state across container shutdown and startup you should mount a volume at /data.

SELinux users should update the security context of the host mount point so that it plays nicely with Docker:

mkdir -p /srv/docker/bind  
chcon -Rt svirt\_sandbox\_file\_t /srv/docker/bind

**Deployment and Maintenance**

To upgrade the BIND DNS Docker image

Download the updated Docker image:

docker pull sameersbn/bind:9.9.5-20170129

Stop the currently running image:

docker stop bind

Remove the stopped container

docker rm -v bind

Start the updated image

docker run -name bind -d \ [OPTIONS] \ sameersbn/bind:9.9.5-20170129

Open the web browser to https://172.17.42.1:10000 and login to webmin as user root and password SecretPassword. This will give you the possibility to configure your DNS server using the Webmin UI.

**Conclusion**

For debugging and maintenance purposes you may want access the containers shell. If you are using Docker version 1.3.0 or higher you can access a running containers shell by starting bash using docker exec:

docker exec -it bind bash

DNS from Outside will not work

We need to configure the firewall

firewall-cmd --list-all-zones

firewall-cmd --zone=externalc --permanent --add-service=dns

firewall-cmd --reload

firewall-cmd --list-all --zone=public

cp /usr/lib/firewalld/services/ssh.xml /etc/firewalld/services/ssh2.xml

cd /etc/firewalld/services

vi ssh2.xml

<?xml version="1.0" encoding="utf-8"?>

<service>

<short>SSH2</short>

<description>Secure Shell (SSH) is a protocol for logging into and executing commands on remote machines. It provides secure encrypted communications. If you plan on accessing your machine remotely via SSH over a firewalled interface, enable this option. You need the openssh-server package installed for this option to be useful.</description>

<port protocol="tcp" port="8022"/>

</service>

esc :wq

firewall-cmd --reload

firewall-cmd --get-services

You will see the new definition for ssh2

firewall-cmd --zone=external --permanent --add-forward-port=port=53:proto=tcp:toport=53:toaddr=172.17.0.2

firewall-cmd --zone=external --permanent --add-forward-port=port=53:proto=udp:toport=53:toaddr=172.17.0.2

Ip of docker container is the 172.17.0.2

[root@security1 services]# systemctl restart firewalld.service

or

firewall-cmd --reload

firewall-cmd --direct --permanent --add-rule ipv4 nat POSTROUTING 0 -o external -j MASQUERADE

firewall-cmd --direct --permanent --add-rule ipv4 filter FORWARD 0 -i docker0 -o external -j ACCEPT

firewall-cmd --direct --permanent --add-rule ipv4 filter FORWARD 0 -i external -o docker0 -m state --state RELATED,ESTABLISHED -j ACCEPT

firewall-cmd --reload

Now on your windows 7 on the public network

For example your real pc – change the dns to point to 10.64.71.1 (the ip of the VM with the container on it)

Test dns from

You may have to restart your virtual machine if it does not work.

### 6. Dockerfile Example: Creating an Image to Install MongoDB

## Docker in Brief

The [Docker project](https://www.docker.com/) offers higher-level tools which work together, 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-based (i.e. Linux Containers) 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**.

**Note:** To learn more about Docker and its parts (e.g. Docker daemon, CLI, images etc.), check out our introductory article to the project: [How To Install and Use Docker on Ubuntu 16.04](https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-16-04).

## Dockerfiles

Each Dockerfile is a script, composed of various commands (instructions) and arguments listed successively to automatically perform actions on a base image in order to create (or form) a new one. They are used for organizing things and greatly help with deployments by simplifying the process start-to-finish.

Dockerfiles begin with defining an image FROM which the build process starts. Followed by various other methods, commands and arguments (or conditions), in return, provide a new image which is to be used for creating docker containers.

They can be used by providing a Dockerfile's content - in various ways - to the **docker daemon** to build an image (as explained in the **How To Use** section).

## Dockerfile Syntax

Before we begin talking about Dockerfiles, let's quickly go over its syntax and what that actually means.

### What is Syntax?

Very simply, syntax in programming means a structure to order commands, arguments, and everything else that is required to program an application to perform a procedure (i.e. a function / collection of instructions).

These structures are based on rules, clearly and explicitly defined, and they are to be followed by the programmer to interface with whichever computer application (e.g. interpreters, daemons etc.) uses or expects them. If a script (i.e. a file containing series of tasks to be performed) is not correctly structured (i.e. wrong syntax), the computer program will not be able to parse it. Parsing roughly can be understood as going over an input with the end goal of understanding what is meant.

Dockerfiles use simple, clean, and clear syntax which makes them strikingly easy to create and use. They are designed to be self explanatory, especially because they allow commenting just like a good and properly written application source-code.

### Dockerfile Syntax Example

Dockerfile syntax consists of two kind of main line blocks: comments and commands + arguments.

# Line blocks used for commenting

command argument argument ..

**A Simple Example:**

# Print "Hello docker!"

RUN echo "Hello docker!"

## Dockerfile Commands (Instructions)

Currently there are about a dozen different set of commands which Dockerfiles can contain to have Docker build an image. In this section, we will go over all of them, individually, before working on a Dockerfile example.

**Note:** As explained in the previous section (Dockerfile Syntax), all these commands are to be listed (i.e. written) successively, inside a single plain text file (i.e. Dockerfile), in the order you would like them performed (i.e. executed) by the docker daemon to build an image. However, some of these commands (e.g. MAINTAINER) can be placed anywhere you see fit (but always after FROM command), as they do not constitute of any execution but rather value of a definition (i.e. just some additional information).

### ADD

The ADD command gets two arguments: a source and a destination. It basically copies the files from the source on the host into the container's own filesystem at the set destination. If, however, the source is a URL (e.g. <http://github.com/user/file/>), then the contents of the URL are downloaded and placed at the destination.

Example:

# Usage: ADD [source directory or URL] [destination directory]

ADD /my\_app\_folder /my\_app\_folder

### CMD

The command CMD, similarly to RUN, can be used for executing a specific command. However, unlike RUN it is not executed during build, but when a container is instantiated using the image being built. Therefore, it should be considered as an initial, default command that gets executed (i.e. run) with the creation of containers based on the image.

**To clarify:** an example for CMD would be running an application upon creation of a container which is already installed using RUN (e.g. RUN apt-get install …) inside the image. This default application execution command that is set with CMD becomes the default and replaces any command which is passed during the creation.

Example:

# Usage 1: CMD application "argument", "argument", ..

CMD "echo" "Hello docker!"

### ENTRYPOINT

ENTRYPOINT argument sets the concrete default application that is used every time a container is created using the image. For example, if you have installed a specific application inside an image and you will use this image to only run that application, you can state it with ENTRYPOINT and whenever a container is created from that image, your application will be the target.

If you couple ENTRYPOINT with CMD, you can remove "application" from CMD and just leave "arguments" which will be passed to the ENTRYPOINT.

Example:

# Usage: ENTRYPOINT application "argument", "argument", ..

# Remember: arguments are optional. They can be provided by CMD

# or during the creation of a container.

ENTRYPOINT echo

# Usage example with CMD:

# Arguments set with CMD can be overridden during \*run\*

CMD "Hello docker!"

ENTRYPOINT echo

### ENV

The ENV command is used to set the environment variables (one or more). These variables consist of “key value” pairs which can be accessed within the container by scripts and applications alike. This functionality of Docker offers an enormous amount of flexibility for running programs.

Example:

# Usage: ENV key value

ENV SERVER\_WORKS 4

### EXPOSE

The EXPOSE command is used to associate a specified port to enable networking between the running process inside the container and the outside world (i.e. the host).

Example:

# Usage: EXPOSE [port]

EXPOSE 8080

To learn about Docker networking, check out the [Docker container networking documentation](https://docs.docker.com/engine/userguide/networking/).

### FROM

FROM directive is probably the most crucial amongst all others for Dockerfiles. It defines the base image to use to start the build process. It can be any image, including the ones you have created previously. If a FROM image is not found on the host, Docker will try to find it (and download) from the **Docker Hub** or other container repository. It needs to be the first command declared inside a Dockerfile.

Example:

# Usage: FROM [image name]

FROM ubuntu

MAINTAINER

One of the commands that can be set anywhere in the file - although it would be better if it was declared on top - is MAINTAINER. This non-executing command declares the author, hence setting the author field of the images. It should come nonetheless after FROM.

Example:

# Usage: MAINTAINER [name]

MAINTAINER authors\_name

### RUN

The RUN command is the central executing directive for Dockerfiles. It takes a command as its argument and runs it to form the image. Unlike CMD, it actually **is** used to build the image (forming another layer on top of the previous one which is committed).

Example:

# Usage: RUN [command]

RUN aptitude install -y riak

### USER

The USER directive is used to set the UID (or username) which is to run the container based on the image being built.

Example:

# Usage: USER [UID]

USER 751

### VOLUME

The VOLUME command is used to enable access from your container to a directory on the host machine (i.e. mounting it).

Example:

# Usage: VOLUME ["/dir\_1", "/dir\_2" ..]

VOLUME ["/my\_files"]

### WORKDIR

The WORKDIR directive is used to set where the command defined with CMD is to be executed.

Example:

# Usage: WORKDIR /path

WORKDIR ~/

## How to Use Dockerfiles

Using Dockerfiles is as simple as having the Docker daemon run one. The output after executing the script will be the ID of the new docker image.

Usage:

# Build an image using the Dockerfile at current location

# Example: docker build -t [name] .

docker build -t my\_mongodb .

# Dockerfile Example: Creating an Image to Install MongoDB

In this final section for Dockerfiles, we will create a Dockerfile document and populate it step-by-step with the end result of having a Dockerfile, which can be used to create a docker image to run MongoDB containers.

**Note:** After starting to edit the Dockerfile, all the content and arguments from the sections below are to be written (appended) inside of it successively, following our example and explanations from the **Docker Syntax** section. You can see what the end result will look like at the latest section of this walkthrough.

### Creating the Empty Dockerfile

Using the nano text editor, let's start editing our Dockerfile.

nano Dockerfile

### Defining Our File and Its Purpose

Albeit optional, it is always a good practice to let yourself and everybody figure out (when necessary) what this file is and what it is intended to do. For this, we will begin our Dockerfile with fancy comments (#) to describe it.

############################################################

# Dockerfile to build MongoDB container images

# Based on Ubuntu

############################################################

### Setting The Base Image to Use

# Set the base image to Ubuntu

FROM ubuntu

### Defining The Maintainer (Author)

# File Author / Maintainer

MAINTAINER Example McAuthor

### Setting Arguments and Commands for Downloading MongoDB

################## BEGIN INSTALLATION ######################

# Install MongoDB Following the Instructions at MongoDB Docs

# Ref: http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/

# Add the package verification key

RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10

# Add MongoDB to the repository sources list

RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' | tee /etc/apt/sources.list.d/mongodb.list

# Update the repository sources list

RUN apt-get update

# Install MongoDB package (.deb)

RUN apt-get install -y mongodb-10gen

# Create the default data directory

RUN mkdir -p /data/db

##################### INSTALLATION END #####################

### Setting The Default Port For MongoDB

# Expose the default port

EXPOSE 27017

# Default port to execute the entrypoint (MongoDB)

CMD ["--port 27017"]

# Set default container command

ENTRYPOINT usr/bin/mongod

### Saving The Dockerfile

After you have appended everything to the file, it is time to save and exit. Press CTRL+X and then Y to confirm and save the Dockerfile.

This is what the final file should look like:

############################################################

# Dockerfile to build MongoDB container images

# Based on Ubuntu

############################################################

# Set the base image to Ubuntu

FROM ubuntu

# File Author / Maintainer

MAINTAINER Example McAuthor

################## BEGIN INSTALLATION ######################

# Install MongoDB Following the Instructions at MongoDB Docs

# Ref: http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/

# Add the package verification key

RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10

# Add MongoDB to the repository sources list

RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' | tee /etc/apt/sources.list.d/mongodb.list

# Update the repository sources list

RUN apt-get update

# Install MongoDB package (.deb)

RUN apt-get install -y mongodb-10gen

# Create the default data directory

RUN mkdir -p /data/db

##################### INSTALLATION END #####################

# Expose the default port

EXPOSE 27017

# Default port to execute the entrypoint (MongoDB)

CMD ["--port 27017"]

# Set default container command

ENTRYPOINT usr/bin/mongod

### Building Our First Image

Using the explanations from before, we are ready to create our first MongoDB image with docker!

docker build -t my\_mongodb .

**Note:** The **-t [name]** flag here is used to tag the image. To learn more about what else you can do during build, run docker build --help.

### Running A MongoDB Instance

Using the image we have build, we can now proceed to the final step: creating a container running a MongoDB instance inside, using a name of our choice (if desired with **-name [name]**).

docker run -name my\_first\_mdb\_instance -i -t my\_mongodb

**Note:** If a name is not set, we will need to deal with complex, alphanumeric IDs which can be obtained by listing all the containers using docker ps -l.

**Note:** To detach yourself from the container, use the escape sequence CTRL+P followed by CTRL+Q.

Enjoy!

Writing a Dockerfile

There are already existing Dockerfiles in [Fedora-Dockerfiles](https://github.com/fedora-cloud/Fedora-Dockerfiles) repository. You can use them as examples for creating your own Dockerfile. Each directory contains a Dockerfile and a README with instructions how to build the image and run a container from it.

A Dockerfile content can be as simple as:

FROM fedora:latest

CMD env

For description of instructions used in Dockerfile see [Dockerfile reference](https://docs.docker.com/engine/reference/builder/) and [Best practices for writing Dockerfiles](https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/)

Building an image from Dockerfile

In a directory with a Dockerfile run

$ sudo docker build -t "my-image" .

If the build is successful you can see the my-image image in docker images output.

See also [Build your own image](https://docs.docker.com/engine/getstarted/step_four/) and [Building an image from a Dockerfile](https://docs.docker.com/engine/getstarted/step_four/#step-2-build-an-image-from-your-dockerfile) for more thorough description.

Authors: [Adam Samalik](mailto:%61%73%61%6D%61%6C%69%6B@%72%65%64%68%61%74.%63%6F%6D), [Budh Ram Gurung](mailto:%62%75%64%68%72%61%6D.%67%75%72%75%6E%67%30%31@%67%6D%61%69%6C.%63%6F%6D), [Jiri Popelka](mailto:%6A%70%6F%70%65%6C%6B%61@%72%65%64%68%61%74.%63%6F%6D), [Josef Stribny](mailto:%6A%73%74%72%69%62%6E%79@%72%65%64%68%61%74.%63%6F%6D)

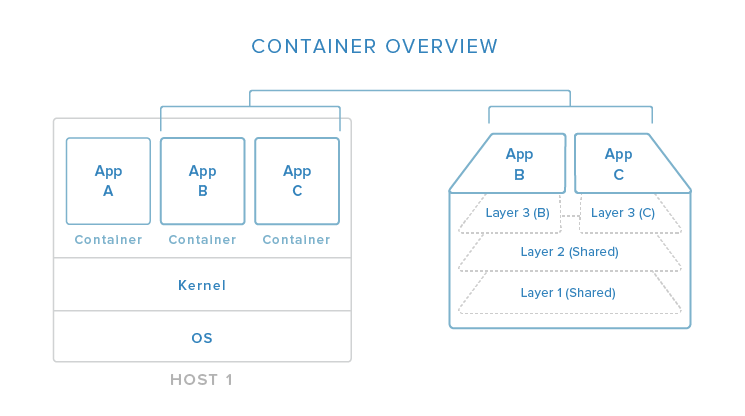
# The Docker Ecosystem: An Introduction to Common Components

### Introduction

Containerization is the process of distributing and deploying applications in a portable and predictable way. It accomplishes this by packaging components and their dependencies into standardized, isolated, lightweight process environments called containers.  Many organizations are now interested in designing applications and services that can be easily deployed to distributed systems, allowing the system to scale easily and survive machine and application failures.  Docker, a containerization platform developed to simplify and standardize deployment in various environments, was largely instrumental in spurring the adoption of this style of service design and management.  A large amount of software has been created to build on this ecosystem of distributed container management.

## Docker and Containerization

Docker is the most common containerization software in use today. While other containerizing systems exist, Docker makes container creation and management simple and integrates with many open source projects.



In this image, you can begin to see (in a simplified view) how containers relate to the host system. Containers isolate individual applications and use operating system resources that have been abstracted by Docker. In the exploded view on the right, we can see that containers can be built by "layering", with multiple containers sharing underlying layers, decreasing resource usage.

Docker’s main advantages are:

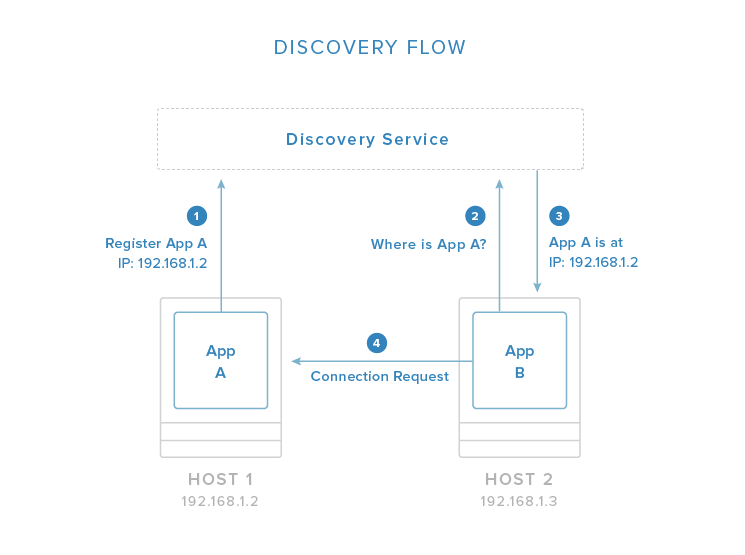
* **Lightweight resource utilization**: instead of virtualizing an entire operating system, containers isolate at the process level and use the host’s kernel.
* **Portability**: all of the dependencies for a containerized application are bundled inside of the container, allowing it to run on any Docker host.
* **Predictability**: The host does not care about what is running inside of the container and the container does not care about which host it is running on.  The interfaces are standardized and the interactions are predictable.

Typically, when designing an application or service to use Docker, it works best to break out functionality into individual containers, a design decision known as service-oriented architecture. This gives you the ability to easily scale or update components independently in the future. Having this flexibility is one of the many reasons that people are interested in Docker for development and deployment.

To find out more about containerizing applications with Docker, click [here](https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-an-overview-of-containerization).

## Service Discovery and Global Configuration Stores

Service discovery is one component of an overall strategy aimed at making container deployments scalable and flexible. Service discovery is used so that containers can find out about the environment they have been introduced to without administrator intervention. They can find connection information for the components they must interact with, and they can register themselves so that other tools know that they are available. These tools also typically function as globally distributed configuration stores where arbitrary config settings can be set for the services operating in your infrastructure.



In the above image, you can see an example flow in which one application registers its connection information with the discovery service system. Once registered, other applications can query the discovery service to find out how to connect to the application.

These tools are often implemented as simple key-value stores that are distributed among the hosts in a clustered environment. Generally, the key-value stores provide an HTTP API for accessing and setting values. Some include additional security measures like encrypted entries or access control mechanisms. The distributed stores are essential for managing the clustered Docker hosts in addition to their primary function of providing self-configuration details for new containers.

Some of the responsibilities of service discovery stores are:

* Allowing applications to obtain the data needed to connect with to the services they depend on.
* Allowing services to register their connection information for the above purpose.
* Providing a globally accessible location to store arbitrary configuration data.
* Storing information about cluster members as needed by any cluster management software.

Some popular service discovery tools and related projects are:

* [**etcd**](https://www.digitalocean.com/community/tutorials/how-to-use-etcdctl-and-etcd-coreos-s-distributed-key-value-store): service discovery / globally distributed key-value store
* [**consul**](https://www.digitalocean.com/community/tutorials/an-introduction-to-using-consul-a-service-discovery-system-on-ubuntu-14-04): service discovery / globally distributed key-value store
* [**zookeeper**](https://www.digitalocean.com/community/tutorials/an-introduction-to-mesosphere#a-basic-overview-of-apache-mesos): service discovery / globally distributed key-value store
* [**crypt**](http://xordataexchange.github.io/crypt/): project to encrypt etcd entries
* [**confd**](https://www.digitalocean.com/community/tutorials/how-to-use-confd-and-etcd-to-dynamically-reconfigure-services-in-coreos): watches key-value store for changes and triggers reconfiguration of services with new values

To learn more about service discovery with Docker, visit our guide [here](https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-service-discovery-and-distributed-configuration-stores).

## Networking Tools

Containerized applications lend themselves to a service-oriented design that encourages breaking out functionality into discrete components. While this makes management and scaling easier, it requires even more assurance regarding the functionality and reliability of networking between the components. Docker itself provides the basic networking structures necessary for container-to-container and container-to-host communication.

Docker's native networking capabilities provide two mechanisms for hooking containers together. The first is to expose a container's ports and optionally map to the host system for external routing. You can select the host port to map to or allow Docker to randomly choose a high, unused port. This is a generic way of providing access to a container that works well for most purposes.

The other method is to allow containers to communicate by using Docker "links". A linked container will get connection information about its counterpart, allowing it to automatically connect if it is configured to pay attention to those variables. This allows contact between containers on the same host without having to know beforehand the port or address where the service will be located.

This basic level of networking is suitable for single-host or closely managed environments. However, the Docker ecosystem has produce a variety of projects that focus on expanding the networking functionality available to operators and developers. Some additional networking capabilities available through additional tools include:

* Overlay networking to simplify and unify the address space across multiple hosts.
* Virtual private networks adapted to provide secure communication between various components.
* Assigning per-host or per-application subnetting
* Establishing macvlan interfaces for communication
* Configuring custom MAC addresses, gateways, etc. for your containers

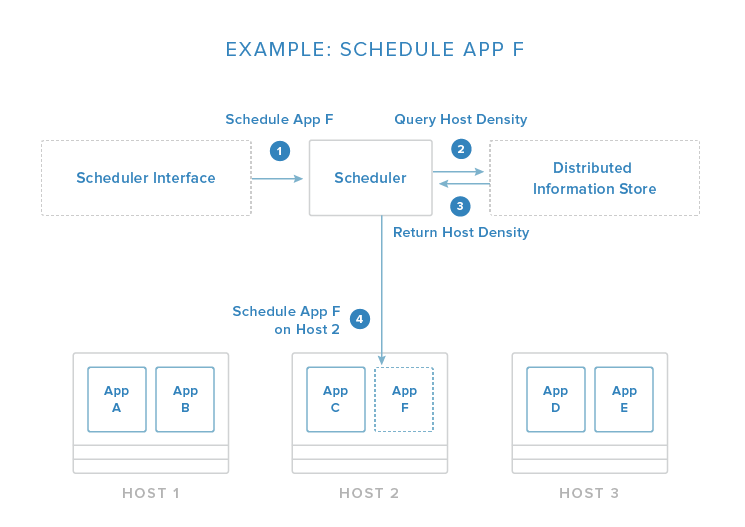
Some projects that are involved with improving Docker networking are:

* **flannel**: Overlay network providing each host with a separate subnet.
* **weave**: Overlay network portraying all containers on a single network.
* **pipework**: Advanced networking toolkit for arbitrarily advanced networking configurations.

For a more in-depth look at the different approaches to networking with Docker, click [here](https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-networking-and-communication).

## Scheduling, Cluster Management, and Orchestration

Another component needed when building a clustered container environment is a scheduler. Schedulers are responsible for starting containers on the available hosts.



The image above demonstrates a simplified scheduling decision. The request is given through an API or management tool. From here, the scheduler evaluates the conditions of the request and the state of the available hosts. In this example, it pulls information about container density from a distributed data store / discovery service (as discussed above) so that it can place the new application on the least busy host.

This host selection process is one of the core responsibilities of the scheduler. Usually, it has functions that automate this process with the administrator having the option to specify certain constraints. Some of these constraints may be:

* Schedule the container on the same host as another given container.
* Make sure that the container is not placed on the same host as another given container.
* Place the container on a host with a matching label or metadata.
* Place the container on the least busy host.
* Run the container on every host in the cluster.

The scheduler is responsible for loading containers onto relevant hosts and starting, stopping, and managing the life cycle of the process.

Because the scheduler must interact with each host in the group, cluster management functions are also typically included. These allow the scheduler to get information about the members and perform administration tasks. Orchestration in this context generally refers to the combination of container scheduling and managing hosts.

Some popular projects that function as schedulers and fleet management tools are:

* [**fleet**](https://www.digitalocean.com/community/tutorials/how-to-use-fleet-and-fleetctl-to-manage-your-coreos-cluster): scheduler and cluster management tool.
* [**marathon**](https://www.digitalocean.com/community/tutorials/an-introduction-to-mesosphere#a-basic-overview-of-marathon): scheduler and service management tool.
* [**Swarm**](https://github.com/docker/swarm/): scheduler and service management tool.
* [**mesos**](https://www.digitalocean.com/community/tutorials/an-introduction-to-mesosphere#a-basic-overview-of-apache-mesos): host abstraction service that consolidates host resources for the scheduler.
* [**kubernetes**](https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes): advanced scheduler capable of managing container groups.
* [**compose**](https://github.com/docker/docker/issues/9694): container orchestration tool for creating container groups.

To find out more about basic scheduling, container grouping, and cluster management software for Docker, click [here](https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-scheduling-and-orchestration).

## Conclusion

By now, you should be familiar with the general function of most of the software associated with the Docker ecosystem. Docker itself, along with all of the supporting projects, provide a software management, design, and deployment strategy that enables massive scalability. By understanding and leveraging the capabilities of various projects, you can execute complex application deployments that are flexible enough to account for variable operating requirements.

### [**The Docker Ecosystem**](https://www.digitalocean.com/community/tutorial_series/the-docker-ecosystem)

The Docker project has given many developers and administrators an easy platform with which to build and deploy scalable applications. In this series, we will be exploring how Docker and the components designed to integrate with it provide the tools needed to easily deliver highly available, distributed applications.

# The Docker Ecosystem: An Overview of Containerization

### Introduction

There are often many roadblocks that stand in the way of easily moving your application through the development cycle and eventually into production. Besides the actual work of developing your application to respond appropriately in each environment, you may also face issues with tracking down dependencies, scaling your application, and updating individual components without affecting the entire application.

Docker containerization and service-oriented design attempts to solve many of these problems. Applications can be broken up into manageable, functional components, packaged individually with all of their dependencies, and deployed on irregular architecture easily. Scaling and updating components is also simplified.

In this guide, we will discuss the benefits of containerization and how Docker helps to solve many of the issues we mentioned above. Docker is the core component in distributed container deployments that provide easy scalability and management.

## A Brief History of Linux Containerization

Containerization and isolation are not new concepts in the computing world. Some Unix-like operating systems have leveraged mature containerization technologies for over a decade.

In Linux, LXC, the building block that formed the foundation for later containerization technologies was added to the kernel in 2008. LXC combined the use of kernel cgroups (allows for isolating and tracking resource utilization) and namespaces (allows groups to be separated so they cannot "see" each other) to implement lightweight process isolation.

Later, Docker was introduced as a way of simplifying the tooling required to create and manage containers. It initially used LXC as its default execution driver (it has since developed a library called libcontainer for this purpose). Docker, while not introducing many new ideas, made them accessible to the average developer and system administrator by simplifying the process and standardizing on an interface. It spurred a renewed interest in containerization in the Linux world among developers.

While some of the topics we will discuss in this article are more general, we will be focusing mainly on Docker containerization due to its overwhelming popularity and its standard adoption.

## What Containerization Brings to the Picture

Containers come with many very attractive benefits for both developers and system administrators / operations teams.

Some of the most benefits are listed below.

### Abstraction of the host system away from the containerized application

Containers are meant to be completely standardized. This means that the container connects to the host and to anything outside of the container using defined interfaces. A containerized application should not rely on or be concerned with details about the underlying host's resources or architecture. This simplifies development assumptions about the operating environment. Likewise, to the host, every container is a black box. It does not care about the details of the application inside.

### Easy Scalability

One of the benefits of the abstraction between the host system and the containers is that, given the correct application design, scaling can be simple and straight-forward. Service-oriented design (discussed later) combined with containerized applications provide the groundwork for easy scalability.

A developer may run a few containers on their workstation, while this system may be scaled horizontally in a staging or testing area. When the containers go into production, they can scale out again.

### Simple Dependency Management and Application Versioning

Containers allow a developer to bundle an application or an application component along with all of its dependencies as a unit. The host system does not have to be concerned with the dependencies needed to run a specific application. As long as it can run Docker, it should be able to run all Docker containers.

This makes dependency management easy and also simplifies application version management as well. Host systems and operations teams are no longer responsible for managing the dependency needs of an application because, apart from a reliance on related containers, they should all be contained within the container itself.

### Extremely lightweight, isolated execution environments

While containers do not provide the same level of isolation and resource management as virtualization technologies, what they win from the trade off is an extremely lightweight execution environment. Containers are isolated at the process level, sharing the host's kernel. This means that the container itself does not include a complete operating system, leading to almost instant startup times. Developers can easily run hundreds of containers from their workstation without an issue.

### Shared Layering

Containers are lightweight in a different sense in that they are committed in "layers". If multiple containers are based on the same layer, they can share the underlying layer without duplication, leading to very minimal disk space utilization for later images.

### Composability and Predictability

Docker files allow users to define the exact actions needed to create a new container image. This allows you to write your execution environment as if it were code, storing it in version control if desirable. The same Docker file built in the same environment will always produce an identical container image.

## Using Dockerfiles for Repeatable, Consistent Builds

While it is possible to create container images using an interactive process, it is often better to place the configuration steps within a Dockerfile once the necessary steps are known. Dockerfiles are simple build files that describe how to create a container image from a known starting point.

Dockerfiles are incredible useful and fairly easy to master. Some of the benefits they provide are:

* **Easy versioning**: The Dockerfiles themselves can be committed to version control to track changes and revert any mistakes
* **Predicatability**: Building images from a Dockerfile helps remove human error from the image creation process.
* **Accountability**: If you plan on sharing your images, it is often a good idea to provide the Dockerfile that created the image as a way for other users to audit the process. It basically provides a command history of the steps taken to create the image.
* **Flexibility**: Creating images from a Dockerfile allows you to override the defaults that interactive builds are given. This means that you do not have to provide as many runtime options to get the image to function as intended.

Dockerfiles are a great tool for automating container image building to establish a repeatable process.

## The Architecture of Containerized Applications

When designing applications to be deployed within containers, one of the first areas of concern is the actual architecture of the application. Generally, containerized applications work best when implementing a service-oriented design.

Service-oriented applications break the functionality of a system into discrete components that communicate with each other over well-defined interfaces. Container technology itself encourages this type of design because it allows each component to scale out or upgrade independently.

Applications implementing this type of design should have the following qualities:

* They should not care about or rely on any specifics of the host system
* Each component should provide consistent APIs that consumers can use to access the service
* Each service should take cues from environmental variables during initial configuration
* Application data should be stored outside of the container on mounted volumes or in data containers

These strategies allow each component to be independently swapped out or upgraded as long as the API is maintained. They also lend themselves towards focused horizontal scalability due to the fact that each component can be scaled according to the bottleneck being experienced.

Rather than hard coding specific values, each component generally can define reasonable defaults. The component can use these as fallback values, but should prefer values that it can gather from its environment. This is often accomplished through the aid of service discovery tools, which the component can query during its startup procedure.

Taking the configuration out of the actual container and placing it into the environment allows for easy changes to application behavior without rebuilding the container image. It also allows a single setting to influence multiple instances of a component. In general, service-oriented design couples well with environmental configuration strategies because both allow for more flexible deployments and more straight-forward scaling.

## Using a Docker Registry for Container Management

Once your application is split into functional components and configured to respond appropriately to other containers and configuration flags within the environment, the next step is usually to make your container images available through a registry. Uploading container images to a registry allows Docker hosts to pull down the image and spin up container instances by simply knowing the image name.

There are various Docker registries available for this purpose. Some are public registries where anyone can see and use the images that have been committed, while other registries are private. Images can be tagged so that they are easy to target for downloads or updating.

## Conclusion

Docker provides the fundamental building block necessary for distributed container deployments. By packaging application components in their own containers, horizontal scaling becomes a simple process of spinning up or shutting down multiple instances of each component. Docker provides the tools necessary to not only build containers, but also manage and share them with new users or hosts.

While containerized applications provide the necessary process isolation and packaging to assist in deployment, there are many other components necessary to adequately manage and scale containers over a distributed cluster of hosts. In our [next guide](https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-service-discovery-and-distributed-configuration-stores), we will discuss how service discovery and globally distributed configuration stores contribute to clustered container deployments.

In this tutorial you installed Docker, worked with images and containers, and pushed a modified image to Docker Hub. Now that you know the basics, explore the [other Docker tutorials](https://www.digitalocean.com/community/tags/docker?type=tutorials) in the DigitalOcean Community.

# How To Install Docker Compose on Debian 9

### Introduction

[Docker](https://docs.docker.com/) is a great tool for automating the deployment of Linux applications inside software containers, but to take full advantage of its potential each component of an application should run in its own individual container. For complex applications with a lot of components, orchestrating all the containers to start up, communicate, and shut down together 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 of your Docker containers and configurations. This became so popular that the Docker team decided to make [Docker Compose](https://docs.docker.com/compose/) based on the Fig source, which is now deprecated. Docker Compose makes it easier for users to orchestrate the processes of Docker containers, including starting up, shutting down, and setting up intra-container linking and volumes.

In this tutorial, we'll show you how to install the latest version of Docker Compose to help you manage multi-container applications on a Debian 9 server.

## Prerequisites

To follow this article, you will need:

* A Debian 9 server and a non-root user with sudo privileges. This [initial server setup with Debian 9 tutorial](https://www.digitalocean.com/community/tutorials/initial-server-setup-with-ubuntu-18-04) explains how to set this up.
* Docker installed with the instructions from **Step 1** and **Step 2** of [How To Install and Use Docker on Debian 9](https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-debian-9)

**Note:** Even though the Prerequisites give instructions for installing Docker on Debian 9, the dockercommands in this article should work on other operating systems as long as Docker is installed.

## Step 1 — Installing Docker Compose

Although we can install Docker Compose from the official Debian repositories, it is several minor versions behind the latest release, so we'll install it from Docker's GitHub repository. The command below is slightly different than the one you'll find on the [Releases](https://github.com/docker/compose/releases) page. By using the -o flag to specify the output file first rather than redirecting the output, this syntax avoids running into a permission denied error caused when using sudo.

We'll check the [current release](https://github.com/docker/compose/releases) and, if necessary, update it in the command below:

* sudo curl -L https://github.com/docker/compose/releases/download/1.22.0/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose

Next we'll set the permissions:

* sudo chmod +x /usr/local/bin/docker-compose

Then we'll verify that the installation was successful by checking the version:

* docker-compose --version

This will print out the version we installed:

Output

docker-compose version 1.22.0, build f46880fe

Now that we have Docker Compose installed, we're ready to run a "Hello World" example.

## Step 2 — Running a Container with Docker Compose

The public Docker registry, Docker Hub, includes a Hello World image for demonstration and testing. It illustrates the minimal configuration required to run a container using Docker Compose: a YAML file that calls a single image. We'll create this minimal configuration to run our hello-world container.

First, we'll create a directory for the YAML file and move into it:

* mkdir hello-world
* cd hello-world

Then, we'll create the YAML file:

* 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 in the YAML file is used as part of the container name. The second line specifies which image to use to create the container. When we run the docker-compose up command, it will look for a local image by the name we specified, hello-world. With this in place, we’ll save and exit the file.

We can look manually at images on our system with the docker images command:

* docker images

When there are no local images at all, only the column headings display:

Output

REPOSITORY TAG IMAGE ID CREATED SIZE

Now, while still in the ~/hello-world directory, we'll execute the following command:

* docker-compose up

The first time we run the command, if there's no local image named hello-world, Docker Compose will pull it from the Docker Hub public repository:

Output

Pulling my-test (hello-world:)...

latest: Pulling from library/hello-world

9db2ca6ccae0: Pull complete

Digest: sha256:4b8ff392a12ed9ea17784bd3c9a8b1fa3299cac44aca35a85c90c5e3c7afacdc

Status: Downloaded newer image for hello-world:latest

. . .

After pulling the image, docker-compose creates a container, attaches, and runs the [hello](https://github.com/docker-library/hello-world/blob/85fd7ab65e079b08019032479a3f306964a28f4d/hello-world/Dockerfile) program, which in turn confirms that the installation appears to be working:

Output

. . .

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 |

. . .

Then it prints an explanation of what it did:

Output

To generate this message, Docker took the following steps:

my-test\_1 | 1. The Docker client contacted the Docker daemon.

my-test\_1 | 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.

my-test\_1 | (amd64)

my-test\_1 | 3. The Docker daemon created a new container from that image which runs the

my-test\_1 | executable that produces the output you are currently reading.

my-test\_1 | 4. The Docker daemon streamed that output to the Docker client, which sent it

my-test\_1 | to your terminal.

Docker containers only run as long as the command is active, so once hello finished running, the container stopped. Consequently, when we look at active processes, the column headers will appear, but the hello-world container won't be listed because it's not running:

* docker ps

Output

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

We can see the container information, which we'll need in the next step, by using the -a flag. This shows all containers, not just active ones:

* docker ps -a

Output

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

06069fd5ca23 hello-world "/hello" 35 minutes ago Exited (0) 35 minutes ago hello-world\_my-test\_1

This displays the information we'll need to remove the container when we're done with it.

## Step 3 — Removing the Image (Optional)

To avoid using unnecessary disk space, we'll remove the local image. To do so, we'll need to delete all the containers that reference the image using the docker rm command, followed by either the CONTAINER IDor the NAME. Below, we're using the CONTAINER ID from the docker ps -a command we just ran. Be sure to substitute the ID of your container:

* docker rm 06069fd5ca23

Once all containers that reference the image have been removed, we can remove the image:

* docker rmi hello-world

## Conclusion

We've now installed Docker Compose, tested our installation by running a Hello World example, and removed the test image and container.

While the Hello World example confirmed our installation, the simple configuration 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. To see the power of Docker Compose in action, you might like to check out this practical example, [How To Configure a Continuous Integration Testing Environment with Docker and Docker Compose on Ubuntu 16.04](https://www.digitalocean.com/community/tutorials/how-to-configure-a-continuous-integration-testing-environment-with-docker-and-docker-compose-on-ubuntu-16-04).

# Installing Webmin Docker Container on Dply

[Ahmed Mahmoud](https://howtolearn.me/author/ahmedmahmoud/)  [Development](https://howtolearn.me/category/development/), [Web Development](https://howtolearn.me/category/development/web-development/)



We’ve covered installing [Docker](https://www.docker.com/) on [Dply](http://dply.co/)[.co](https://dply.co/) in a previous post, this post will be a follow on to this post and we’ll use Docker containers to install Webmin Docker container on dply.co.

This post will be an introduction to deploying web applications on dply.co using Docker containers, please note that the free tier of dply.co is only for 2 hours so you should only use it for trial purposes, of course, you can always pay to make it a production version.

## Step One

Follow the [first post](https://howtolearn.me/installing-docker-on-dply/) to create an account and install Docker.

## Step Two

In the terminal window, we’ll need to install Docker compose using the following command;

sudo curl -o /usr/local/bin/docker-compose -L “https://github.com/docker/compose/releases/download/1.11.2/docker-compose-$(uname -s)-$(uname -m)”

1 - Webmin Docker on Dply

## Step Three

You’ll then need to apply executable permissions for the binaries using the following command;

chmod +x /usr/local/bin/docker-compose

## 1-1 - Webmin Docker on Dply

## Step Four

In this step, we should create the docker-compose file to deploy Webmin use the following command to do that;

nano docker-compose.yml

1-2 - Webmin Docker on Dply

## Step Five

Now, copy and paste the following code in the newly created file;

webmin:  
image: irasnyd/webmin-ldap  
ports:  
– “10000:10000”  
environment:  
– “WEBMIN\_ROOT\_PASSWORD=supersecret”  
– “WEBMIN\_LDAP\_ADMINISTRATORS=user1:pass1,user2:pass2”  
– “WEBMIN\_SSL=0”  
– “LDAP\_URI=ldap://ldap.example.com”  
– “LDAP\_BINDDN=cn=admin,dc=example,dc=com”  
– “LDAP\_BINDPW=adminpassword”  
– “LDAP\_BASE=dc=example,dc=com”  
– “LDAP\_BASE\_UID=10000”  
– “LDAP\_BASE\_GID=10000”  
– “LDAP\_DEFAULT\_SHELL=/bin/bash”  
– “LDAP\_DEFAULT\_GROUP=ldapusers”  
– “LDAP\_USER\_BASE=ou=Users,dc=example,dc=com”  
– “LDAP\_GROUP\_BASE=ou=Groups,dc=example,dc=com”  
mem\_limit: “1g”  
restart: “always”

Source: https://github.com/irasnyd/webmin-ldap/blob/master/docker-compose.yml

## Step Six

Now type the following command to start the deployment process;

docker-compose up

## Step Seven

After the installation process is complete, you can log to your container from the browser using your IP address that you can get from your Dply dashboard e.g. (http://machine\_IP:10000).

## 6 - wordpress Docker on Dply

## Step Eight

Now, you’ll be directed to Webmin login window, use “user1” as username and “pass1” as the password.