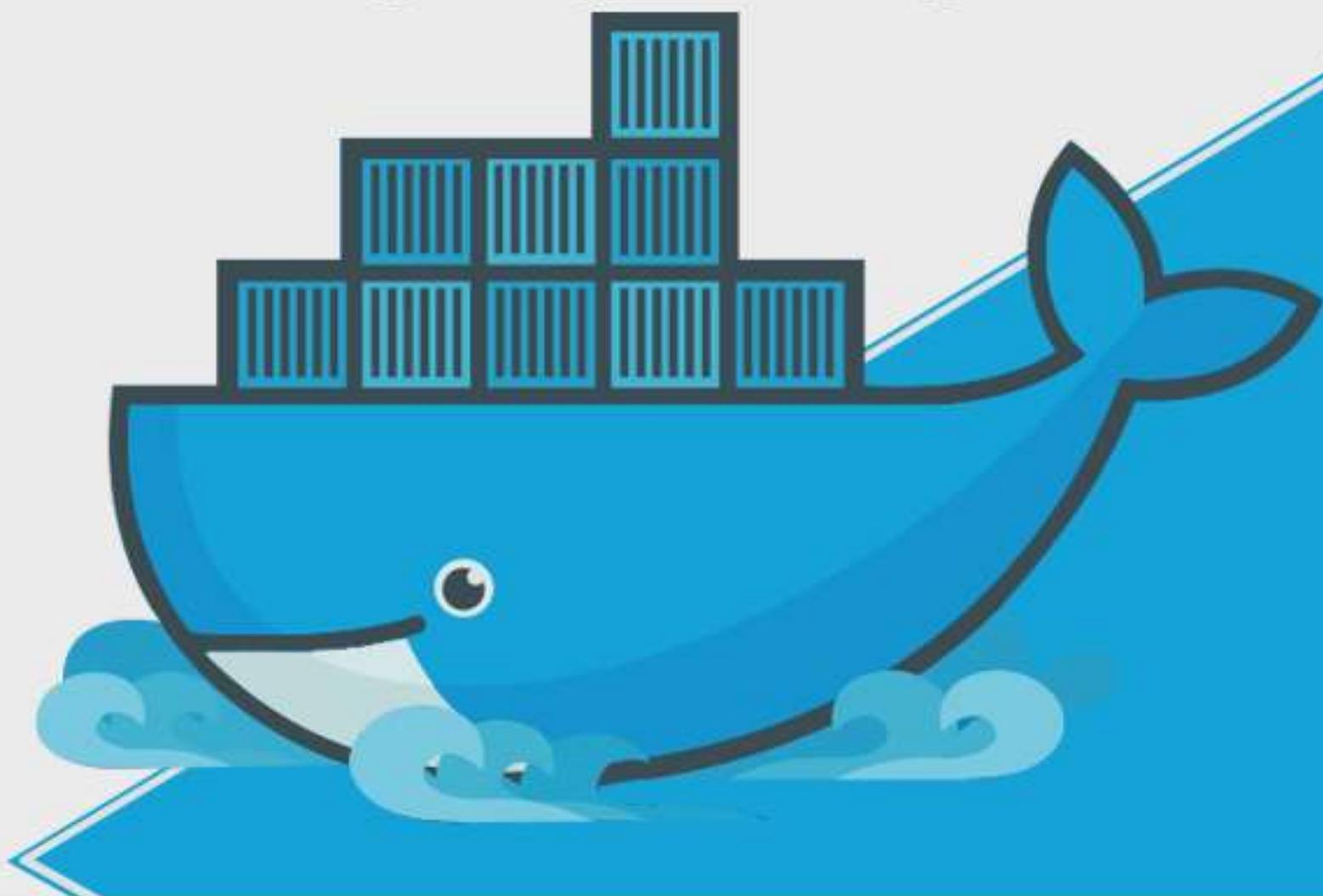




Docker



tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

About the Tutorial

This tutorial explains the various aspects of the Docker Container service. Starting with the basics of Docker which focuses on the installation and configuration of Docker, it gradually moves on to advanced topics such as Networking and Registries. The last few chapters of this tutorial cover the development aspects of Docker and how you can get up and running on the development environments using Docker Containers.

Audience

This tutorial is meant for those who are interested in learning Docker as a container service. This product has spread like wildfire across the industry and is really making an impact on the development of new generation applications. So anyone who is interested in learning all the aspects of Docker should go through this tutorial.

Prerequisites

The prerequisite is that the readers should be familiar with the basic concepts of Windows and the various programs that are already available on the Windows operating system. In addition, it would help if the readers have some exposure to Linux.

Copyright & Disclaimer

© Copyright 2017 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com.

Table of Contents

About the Tutorial.....	i
Audience	i
Prerequisites	i
Copyright & Disclaimer.....	i
Table of Contents.....	ii
1. DOCKER – OVERVIEW	1
2. DOCKER – INSTALLING DOCKER ON LINUX	3
Docker Version.....	10
Docker Info	11
Docker for Windows.....	12
Docker ToolBox.....	13
3. DOCKER – INSTALLATION	15
Docker for Windows.....	15
Docker ToolBox.....	16
Working with Docker Toolbox	19
4. DOCKER – DOCKER HUB	22
5. DOCKER – IMAGES.....	26
Displaying Docker Images.....	26
Downloading Docker Images	27
Removing Docker Images	28
docker images -q	29
docker inspect.....	30
6. DOCKER – CONTAINERS.....	31
Running a Container.....	31

Listing of Containers.....	31
 docker ps -a.....	32
 docker history	33
7. DOCKER – WORKING WITH CONTAINERS	34
 docker top.....	34
 docker stop	35
 docker rm.....	35
 docker stats.....	36
 docker attach	37
 docker pause.....	38
 docker unpause.....	39
 docker kill.....	39
 Docker – Container Lifecycle	40
8. DOCKER – DOCKER ARCHITECTURE	42
9. DOCKER – CONTAINER AND HOSTS	44
 Docker Images.....	44
 Running a Container.....	44
 Listing All Containers.....	45
 Stopping a Container.....	45
10. DOCKER – CONFIGURING DOCKER	46
 service docker stop	46
 service docker start	46
11. DOCKER – CONTAINERS AND SHELLS.....	48
 nsenter.....	49
12. DOCKER – DOCKER FILE	51

13. DOCKER – BUILDING DOCKER FILES.....	53
docker build	53
14. DOCKER – PUBLIC REPOSITORIES.....	56
docker tag	58
docker push.....	59
15. DOCKER – MANAGING PORTS	61
docker inspect.....	63
16. DOCKER – PRIVATE REGISTRIES	66
17. DOCKER – BUILDING A WEB SERVER DOCKER FILE	70
18. DOCKER – INSTRUCTION COMMANDS	73
CMD Instruction	73
ENTRYPOINT	74
ENV	75
WORKDIR	77
19. DOCKER – CONTAINER LINKING.....	79
20. DOCKER – DOCKER STORAGE	81
Data Volumes.....	83
Changing the Storage Driver for a Container	85
Creating a Volume.....	86
Listing all the Volumes	87
21. DOCKER – DOCKER NETWORKING	88
Listing All Docker Networks.....	88
Inspecting a Docker network.....	89
Creating Your Own New Network	91

22. DOCKER – SETTING NODE.JS.....	93
23. DOCKER – SETTING MONGODB	96
24. DOCKER – SETTING NGINX.....	101
25. DOCKER – DOCKER TOOLBOX	105
Running in Powershell.....	106
Pulling Images and Running Containers.....	106
Kitematic.....	107
26. DOCKER – SETTING ASP.NET.....	112
Prerequisites	112
Installing the ASP.Net Container	114
27. DOCKER – DOCKER CLOUD	117
Getting started.....	117
Connecting to the Cloud Provider.....	118
Setting Up Nodes	125
Deploying a Service	127
28. DOCKER – LOGGING	129
Daemon Logging.....	129
Container Logging	131
29. DOCKER – DOCKER COMPOSE	132
Docker Compose – Installation.....	132
Creating Your First Docker-Compose File	133
30. DOCKER – CONTINUOUS INTEGRATION.....	136
31. DOCKER – KUBERNETES ARCHITECTURE.....	140
32. DOCKER – WORKING OF KUBERNETES	142

1. Docker – Overview

Docker is a container management service. The keywords of Docker are **develop**, **ship** and **run** anywhere. The whole idea of Docker is for developers to easily develop applications, ship them into containers which can then be deployed anywhere.

The initial release of Docker was in March 2013 and since then, it has become the buzzword for modern world development, especially in the face of Agile-based projects.



Features of Docker

- Docker has the ability to reduce the size of development by providing a smaller footprint of the operating system via containers.
- With containers, it becomes easier for teams across different units, such as development, QA and Operations to work seamlessly across applications.
- You can deploy Docker containers anywhere, on any physical and virtual machines and even on the cloud.
- Since Docker containers are pretty lightweight, they are very easily scalable.

Components of Docker

Docker has the following components

- **Docker for Mac** – It allows one to run Docker containers on the Mac OS.
- **Docker for Linux** - It allows one to run Docker containers on the Linux OS.

- **Docker for Windows** - It allows one to run Docker containers on the Windows OS.
- **Docker Engine** – It is used for building Docker images and creating Docker containers.
- **Docker Hub** – This is the registry which is used to host various Docker images.
- **Docker Compose** – This is used to define applications using multiple Docker containers.

We will discuss all these components in detail in the subsequent chapters.

The official site for Docker is <https://www.docker.com/>. The site has all information and documentation about the Docker software. It also has the download links for various operating systems.

2. Docker – Installing Docker on Linux

To start the installation of Docker, we are going to use an Ubuntu instance. You can use Oracle Virtual Box to setup a virtual Linux instance, in case you don't have it already.

The following screenshot shows a simple Ubuntu server which has been installed on Oracle Virtual Box. There is an OS user named **demo** which has been defined on the system having entire root access to the sever.

```
demo@ubuntu:~$
```

To install Docker, we need to follow the steps given below.

Step 1: Before installing Docker, you first have to ensure that you have the right Linux kernel version running. Docker is only designed to run on Linux kernel version 3.8 and higher. We can do this by running the following command:

uname

This method returns the system information about the Linux system.

Syntax

```
uname -a
```

Options

a – This is used to ensure that the system information is returned.

Return Value

This method returns the following information on the Linux system:

- kernel name
- node name
- kernel release
- kernel version
- machine
- processor
- hardware platform
- operating system

Example

```
uname -a
```

Output

When we run above command, we will get the following result:

```
demo@ubuntu:~$ uname -a
Linux ubuntu 4.2.0-27-generic #32~14.04.1-Ubuntu SMP Fri Jan 22 15:32:27 UTC 2016 i686 i686 i686 GNU/Linux
demo@ubuntu:~$ _
```

From the output, we can see that the Linux kernel version is 4.2.0-27 which is higher than version 3.8, so we are good to go.

Step 2: You need to update the OS with the latest packages, which can be done via the following command:

```
apt-get
```

This method installs packages from the Internet on to the Linux system.

Syntax

```
sudo apt-get update
```

Options

- **sudo** - The **sudo** command is used to ensure that the command runs with root access.
- **update** - The **update** option is used ensure that all packages are updated on the Linux system.

Return Value

None

Example

```
sudo apt-get update
```

Output

When we run the above command, we will get the following result:

```
Hit http://us.archive.ubuntu.com trusty-backports/universe Sources
Hit http://us.archive.ubuntu.com trusty-backports/multiverse Sources
Hit http://us.archive.ubuntu.com trusty-backports/main i386 Packages
Hit http://us.archive.ubuntu.com trusty-backports/restricted i386 Packages
Hit http://us.archive.ubuntu.com trusty-backports/universe i386 Packages
Hit http://us.archive.ubuntu.com trusty-backports/multiverse i386 Packages
Hit http://us.archive.ubuntu.com trusty-backports/main Translation-en
Hit http://us.archive.ubuntu.com trusty-backports/multiverse Translation-en
Hit http://us.archive.ubuntu.com trusty-backports/restricted Translation-en
Hit http://us.archive.ubuntu.com trusty-backports/universe Translation-en
Hit http://us.archive.ubuntu.com trusty Release
Hit http://us.archive.ubuntu.com trusty/main Sources
Hit http://us.archive.ubuntu.com trusty/restricted Sources
Hit http://us.archive.ubuntu.com trusty/universe Sources
Hit http://us.archive.ubuntu.com trusty/multiverse Sources
Hit http://us.archive.ubuntu.com trusty/main i386 Packages
Hit http://us.archive.ubuntu.com trusty/restricted i386 Packages
Hit http://us.archive.ubuntu.com trusty/universe i386 Packages
Hit http://us.archive.ubuntu.com trusty/multiverse i386 Packages
Hit http://us.archive.ubuntu.com trusty/main Translation-en
Hit http://us.archive.ubuntu.com trusty/multiverse Translation-en
Hit http://us.archive.ubuntu.com trusty/restricted Translation-en
Hit http://us.archive.ubuntu.com trusty/universe Translation-en
Ign http://us.archive.ubuntu.com trusty/main Translation-en_US
Ign http://us.archive.ubuntu.com trusty/multiverse Translation-en_US
Ign http://us.archive.ubuntu.com trusty/restricted Translation-en_US
Ign http://us.archive.ubuntu.com trusty/universe Translation-en_US
Fetched 3,906 kB in 21s (184 kB/s)
Reading package lists... Done
demo@ubuntu:~$
```

This command will connect to the internet and download the latest system packages for Ubuntu.

Step 3: The next step is to install the necessary certificates that will be required to work with the Docker site later on to download the necessary Docker packages. It can be done with the following command:

```
sudo apt-get install apt-transport-https ca-certificates
```

```
demo@ubuntudemo:~$ sudo apt-get install apt-transport-https ca-certificates
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages will be upgraded:
  apt-transport-https ca-certificates
2 upgraded, 0 newly installed, 0 to remove and 105 not upgraded.
Need to get 215 kB of archives.
After this operation, 8,192 B disk space will be freed.
Get:1 http://us.archive.ubuntu.com/ubuntu/ trusty-updates/main apt-transport-https amd64 1.0.1ubuntu2.15 [25.0 kB]
Get:2 http://us.archive.ubuntu.com/ubuntu/ trusty-updates/main ca-certificates i1 20160104ubuntu0.14.04.1 [190 kB]
Fetched 215 kB in 1s (152 kB/s)
Preconfiguring packages ...
(Reading database ... 57694 files and directories currently installed.)
Preparing to unpack .../apt-transport-https_1.0.1ubuntu2.15_amd64.deb ...
Unpacking apt-transport-https (1.0.1ubuntu2.15) over (1.0.1ubuntu2.11) ...
Preparing to unpack .../ca-certificates_20160104ubuntu0.14.04.1_all.deb ...
Unpacking ca-certificates (20160104ubuntu0.14.04.1) over (20141019ubuntu0.14.04.1) ...
Processing triggers for man-db (2.6.7.1-1ubuntu1) ...
Setting up apt-transport-https (1.0.1ubuntu2.15) ...
Setting up ca-certificates (20160104ubuntu0.14.04.1) ...
Processing triggers for ca-certificates (20160104ubuntu0.14.04.1) ...
Updating certificates in /etc/ssl/certs... 19 added, 19 removed; done.
Running hooks in /etc/ca-certificates/update.d....done.
demo@ubuntudemo:~$
```

Step 4: The next step is to add the new GPG key. This key is required to ensure that all data is encrypted when downloading the necessary packages for Docker.

The following command will download the key with the ID 58118E89F3A912897C070ADBF76221572C52609D from the **keyserver** hkp://ha.pool.sks-keyservers.net:80 and adds it to the **adv** keychain. Please note that this particular key is required to download the necessary Docker packages.

```
sudo apt-key adv \
  --keyserver hkp://ha.pool.sks-keyservers.net:80 \
  --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```

```
demo@ubuntudemo:~$ sudo apt-key adv \ --keyserver hkp://ha.pool.sks-keyservers.net:80 \ --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
Executing: gpg --ignore-time-conflict --no-options --no-default-keyring --homedir /tmp/tmp.KcaZ3WImGt --no-auto-check-trustdb --trust-model always --keyring /etc/apt/trusted.gpg --primary-keyring /etc/apt/trusted.gpg --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
gpg: requesting key 2C52609D from hkp server ha.pool.sks-keyservers.net
gpg: key 2C52609D: public key "Docker Release Tool (releasedocker) <docker@dockr.com>" imported
gpg: Total number processed: 1
gpg:           imported: 1 (RSA: 1)
demo@ubuntudemo:~$
```

Step 5: Next, depending on the version of Ubuntu you have, you will need to add the relevant site to the **docker.list** for the **apt package manager**, so that it will be able to detect the Docker packages from the Docker site and download them accordingly.

- Precise 12.04 (LTS) – deb https://apt.dockerproject.org/repo ubuntu-precise main
- Trusty 14.04 (LTS) – deb https://apt.dockerproject.org/repo ubuntu-trusty main
- Wily 15.10 – deb https://apt.dockerproject.org/repo ubuntu-wily main
- Xenial 16.04 (LTS) – deb https://apt.dockerproject.org/repo ubuntu-xenial main

Since our OS is Ubuntu 14.04, we will use the Repository name as “deb https://apt.dockerproject.org/repo ubuntu-trusty main”

And then, we will need to add this repository to the **docker.list** as mentioned above.

```
echo "deb https://apt.dockerproject.org/repo ubuntu-trusty main" | sudo tee
/etc/apt/sources.list.d/docker.list
```

```
demo@ubuntudemo:~$ echo "deb https://apt.dockerproject.org/repo ubuntu-trusty
in" | sudo tee /etc/apt/sources.list.d/docker.list
deb https://apt.dockerproject.org/repo ubuntu-trusty main
demo@ubuntudemo:~$ _
```

Step 6: Next, we issue the **apt-get update command** to update the packages on the Ubuntu system.

```
Hit http://us.archive.ubuntu.com trusty-backports/multiverse Packages
Hit http://us.archive.ubuntu.com trusty-backports/main Translation-en
Hit http://us.archive.ubuntu.com trusty-backports/multiverse Translation-en
Hit http://us.archive.ubuntu.com trusty-backports/restricted Translation-en
Hit http://us.archive.ubuntu.com trusty-backports/universe Translation-en
Hit http://us.archive.ubuntu.com trusty Release
Hit http://us.archive.ubuntu.com trusty/main Sources
Hit http://us.archive.ubuntu.com trusty/restricted Sources
Hit http://us.archive.ubuntu.com trusty/universe Sources
Hit http://us.archive.ubuntu.com trusty/multiverse Sources
Hit http://us.archive.ubuntu.com trusty/main amd64 Packages
Hit http://us.archive.ubuntu.com trusty/restricted amd64 Packages
Hit http://us.archive.ubuntu.com trusty/universe amd64 Packages
Hit http://us.archive.ubuntu.com trusty/multiverse amd64 Packages
Hit http://us.archive.ubuntu.com trusty/main i386 Packages
Hit http://us.archive.ubuntu.com trusty/restricted i386 Packages
Hit http://us.archive.ubuntu.com trusty/universe i386 Packages
Hit http://us.archive.ubuntu.com trusty/multiverse i386 Packages
Hit http://us.archive.ubuntu.com trusty/main Translation-en
Hit http://us.archive.ubuntu.com trusty/multiverse Translation-en
Hit http://us.archive.ubuntu.com trusty/restricted Translation-en
Hit http://us.archive.ubuntu.com trusty/universe Translation-en
Ign http://us.archive.ubuntu.com trusty/main Translation-en_US
Ign http://us.archive.ubuntu.com trusty/multiverse Translation-en_US
Ign http://us.archive.ubuntu.com trusty/restricted Translation-en_US
Ign http://us.archive.ubuntu.com trusty/universe Translation-en_US
Fetched 3,333 KB in 36s (90.8 kB/s)
Reading package lists... Done
demo@ubuntudemo:~$ _
```

Step 7: If you want to verify that the package manager is pointing to the right repository, you can do it by issuing the **apt-cache command**.

```
apt-cache policy docker-engine
```

In the output, you will get the link to <https://apt.dockerproject.org/repo/>

```
1.8.2-0~trusty 0
  500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
s
1.8.1-0~trusty 0
  500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
s
1.8.0-0~trusty 0
  500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
s
1.7.1-0~trusty 0
  500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
s
1.7.0-0~trusty 0
  500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
s
1.6.2-0~trusty 0
  500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
s
1.6.1-0~trusty 0
  500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
s
1.6.0-0~trusty 0
  500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
s
1.5.0-0~trusty 0
  500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
s
demo@ubuntudemo:~$
```

Step 8: Issue the **apt-get update command** to ensure all the packages on the local system are up to date.

```
Hit http://us.archive.ubuntu.com trusty-backports/main Translation-en
Hit http://us.archive.ubuntu.com trusty-backports/multiverse Translation-en
Hit http://us.archive.ubuntu.com trusty-backports/restricted Translation-en
Hit http://us.archive.ubuntu.com trusty-backports/universe Translation-en
Hit http://us.archive.ubuntu.com trusty Release
Hit http://us.archive.ubuntu.com trusty/main Sources
Hit http://us.archive.ubuntu.com trusty/restricted Sources
Hit http://us.archive.ubuntu.com trusty/universe Sources
Hit http://us.archive.ubuntu.com trusty/multiverse Sources
Hit http://us.archive.ubuntu.com trusty/main amd64 Packages
Hit http://us.archive.ubuntu.com trusty/restricted amd64 Packages
Hit http://us.archive.ubuntu.com trusty/universe amd64 Packages
Hit http://us.archive.ubuntu.com trusty/multiverse amd64 Packages
Hit http://us.archive.ubuntu.com trusty/main i386 Packages
Hit http://us.archive.ubuntu.com trusty/restricted i386 Packages
Hit http://us.archive.ubuntu.com trusty/universe i386 Packages
Hit http://us.archive.ubuntu.com trusty/multiverse i386 Packages
Hit http://us.archive.ubuntu.com trusty/main Translation-en
Hit http://us.archive.ubuntu.com trusty/multiverse Translation-en
Hit http://us.archive.ubuntu.com trusty/restricted Translation-en
Hit http://us.archive.ubuntu.com trusty/universe Translation-en
Ign http://us.archive.ubuntu.com trusty/main Translation-en_US
Ign http://us.archive.ubuntu.com trusty/multiverse Translation-en_US
Ign http://us.archive.ubuntu.com trusty/restricted Translation-en_US
Ign http://us.archive.ubuntu.com trusty/universe Translation-en_US
Fetched 30.2 kB in 15s (1,980 B/s)
Reading package lists... Done
demo@ubuntudemo:~$
```

Step 9: For Ubuntu Trusty, Wily, and Xenial, we have to install the linux-image-extra-* kernel packages, which allows one to use the **aufs storage driver**. This driver is used by the newer versions of Docker.

It can be done by using the following command:

```
sudo apt-get install linux-image-extra-$(uname -r) linux-image-extra-virtual
```

```
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-4.2.0-27-generic
Found initrd image: /boot/initrd.img-4.2.0-27-generic
Found linux image: /boot/vmlinuz-3.13.0-105-generic
Found initrd image: /boot/initrd.img-3.13.0-105-generic
Found memtest86+ image: /memtest86+.elf
Found memtest86+ image: /memtest86+.bin
done
Setting up linux-image-extra-3.13.0-105-generic (3.13.0-105.152) ...
run-parts: executing /etc/kernel/postinst.d/apt-auto-removal 3.13.0-105-generic
/boot/vmlinuz-3.13.0-105-generic
run-parts: executing /etc/kernel/postinst.d/initramfs-tools 3.13.0-105-generic
boot/vmlinuz-3.13.0-105-generic
update-initramfs: Generating /boot/initrd.img-3.13.0-105-generic
run-parts: executing /etc/kernel/postinst.d/update-notifier 3.13.0-105-generic
boot/vmlinuz-3.13.0-105-generic
run-parts: executing /etc/kernel/postinst.d/zz-update-grub 3.13.0-105-generic
boot/vmlinuz-3.13.0-105-generic
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-4.2.0-27-generic
Found initrd image: /boot/initrd.img-4.2.0-27-generic
Found linux image: /boot/vmlinuz-3.13.0-105-generic
Found initrd image: /boot/initrd.img-3.13.0-105-generic
Found memtest86+ image: /memtest86+.elf
Found memtest86+ image: /memtest86+.bin
done
Setting up linux-image-generic (3.13.0.105.113) ...
Setting up linux-image-extra-virtual (3.13.0.105.113) ...
demo@ubuntudemo:~$
```

Step 10: The final step is to install Docker and we can do this with the following command:

```
sudo apt-get install -y docker-engine
```

Here, **apt-get** uses the install option to download the Docker-engine image from the Docker website and get Docker installed.

The Docker-engine is the official package from the Docker Corporation for Ubuntu-based systems.

```
Selecting previously unselected package liberror-perl.
Preparing to unpack .../liberror-perl_0.17-1.1_all.deb ...
Unpacking liberror-perl (0.17-1.1) ...
Selecting previously unselected package git-man.
Preparing to unpack .../git-man_1%3a1.9.1-1ubuntu0.3_all.deb ...
Unpacking git-man (1:1.9.1-1ubuntu0.3) ...
Selecting previously unselected package git.
Preparing to unpack .../git_1%3a1.9.1-1ubuntu0.3_amd64.deb ...
Unpacking git (1:1.9.1-1ubuntu0.3) ...
Selecting previously unselected package cgroup-lite.
Preparing to unpack .../cgroup-lite_1.9_all.deb ...
Unpacking cgroup-lite (1.9) ...
Processing triggers for man-db (2.6.7.1-1ubuntu1) ...
Processing triggers for ureadahead (0.100.0-16) ...
ureadahead will be reprofiled on next reboot
Setting up libltdl7:amd64 (2.4.2-1.7ubuntu1) ...
Setting up libsystemd-journal0:amd64 (204-Subuntu20.20) ...
Setting up aufs-tools (1:3.2+20130722-1.1) ...
Setting up docker-engine (1.12.3-0~trusty) ...
docker start/running, process 22612
Setting up liberror-perl (0.17-1.1) ...
Setting up git-man (1:1.9.1-1ubuntu0.3) ...
Setting up git (1:1.9.1-1ubuntu0.3) ...
Setting up cgroup-lite (1.9) ...
cgroup-lite start/running
Processing triggers for libc-bin (2.19-0ubuntu6.7) ...
Processing triggers for ureadahead (0.100.0-16) ...
demo@ubuntudemo:~$
```

In the next section, we will see how to check for the version of Docker that was installed.

Docker Version

To see the version of Docker running, you can issue the following command:

Syntax

```
docker version
```

Options

- **version** – It is used to ensure the Docker command returns the Docker version installed.

Return Value

The output will provide the various details of the Docker version installed on the system.

Example

```
sudo docker version
```

Output

When we run the above program, we will get the following result:

```
demo@ubuntudemo:~$ sudo docker version
Client:
Version:      1.12.3
API version:  1.24
Go version:   go1.6.3
Git commit:   6b644ec
Built:        Wed Oct 26 21:44:32 2016
OS/Arch:      linux/amd64

Server:
Version:      1.12.3
API version:  1.24
Go version:   go1.6.3
Git commit:   6b644ec
Built:        Wed Oct 26 21:44:32 2016
OS/Arch:      linux/amd64
demo@ubuntudemo:~$ _
```

Docker Info

To see more information on the Docker running on the system, you can issue the following command:

Syntax

```
docker info
```

Options

- **info** – It is used to ensure that the Docker command returns the detailed information on the Docker service installed.

Return Value

The output will provide the various details of the Docker installed on the system such as

- Number of containers
- Number of images
- The storage driver used by Docker
- The root directory used by Docker
- The execution driver used by Docker

Example

```
sudo docker info
```

Output

When we run the above command, we will get the following result:

```
Backing Filesystem: extfs
Dirs: 0
Dirperm1 Supported: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
  Volume: local
  Network: bridge null host overlay
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Security Options: apparmor
Kernel Version: 4.2.0-27-generic
Operating System: Ubuntu 14.04.4 LTS
OSType: linux
Architecture: x86_64
CPUs: 1
Total Memory: 993.1 MiB
Name: ubuntudemo
ID: ECDA:IFR3:ZCQJ:FMXL:APJR:BT6Y:JJ75:FUE6:DNP5:PD7B:AOAD:YVB4
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Registry: https://index.docker.io/v1/
WARNING: No swap limit support
Insecure Registries:
  127.0.0.0/8
demo@ubuntudemo:~$
```

Docker for Windows

Docker has out-of-the-box support for Windows, but you need to have the following configuration in order to install Docker for Windows.

System Requirements

Windows OS	Windows 10 64 bit
Memory	2 GB RAM (recommended)

You can download Docker for Windows from: <https://docs.docker.com/docker-for-windows/>



Docker ToolBox

Docker ToolBox has been designed for older versions of Windows, such as Windows 8.1 and Windows 7. You need to have the following configuration in order to install Docker for Windows.

System Requirements

Windows OS	Windows 7 , 8, 8.1
Memory	2 GB RAM (recommended)
Virtualization	This should be enabled.

You can download Docker ToolBox from: <https://www.docker.com/products/docker-toolbox>



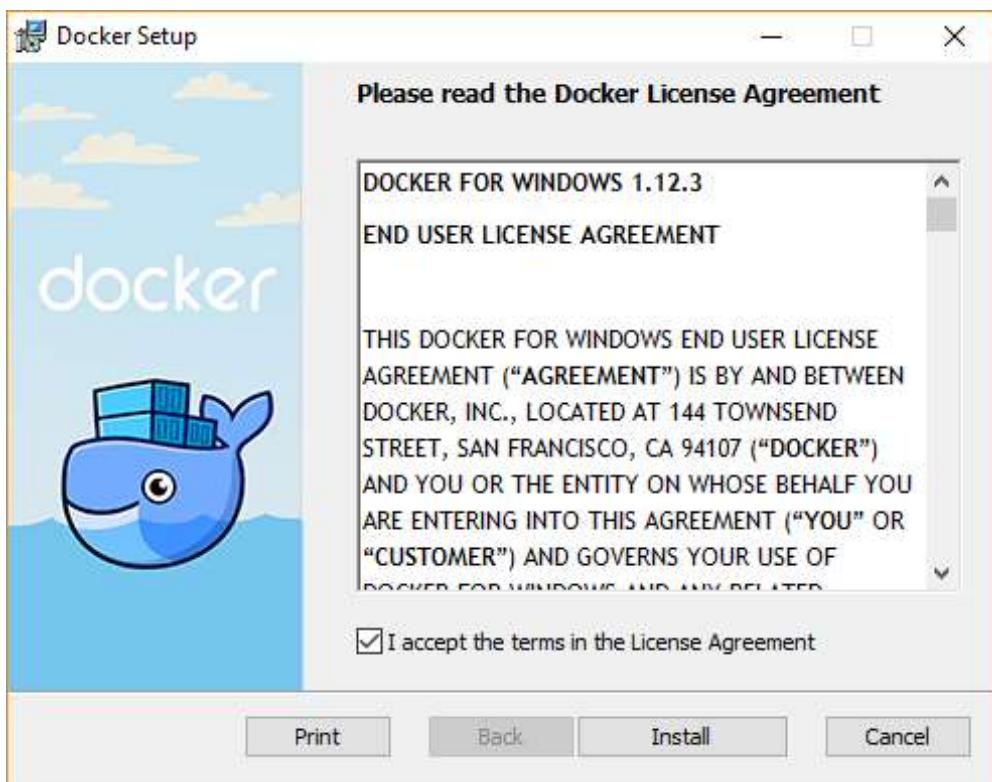
3. Docker – Installation

Let's go through the installation of each product.

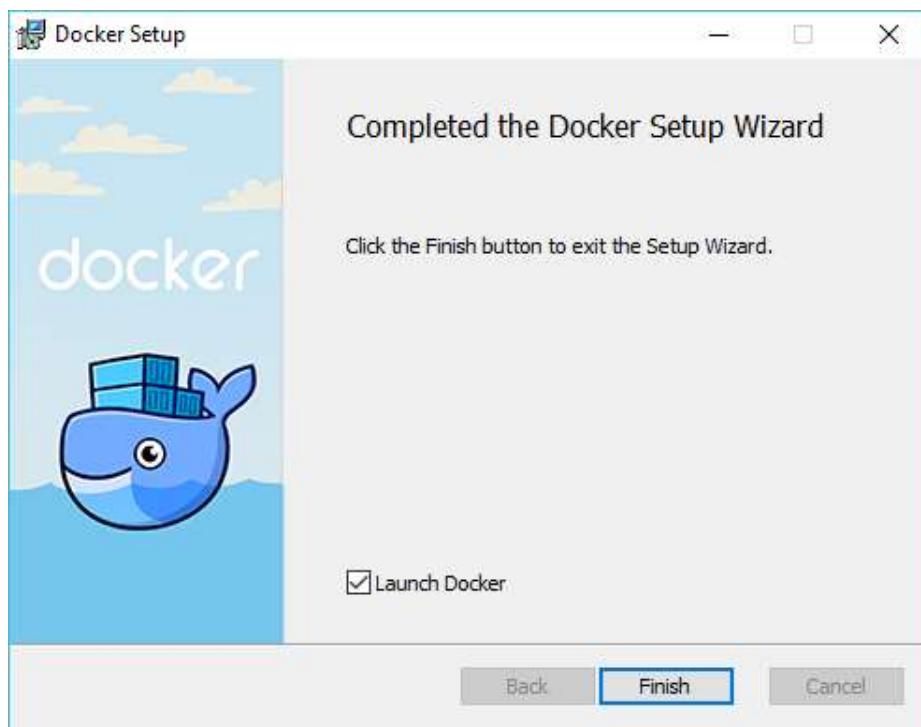
Docker for Windows

Once the installer has been downloaded, double-click it to start the installer and then follow the steps given below.

Step 1: Click on the Agreement terms and then the Install button to proceed ahead with the installation.



Step 2: Once complete, click the Finish button to complete the installation.



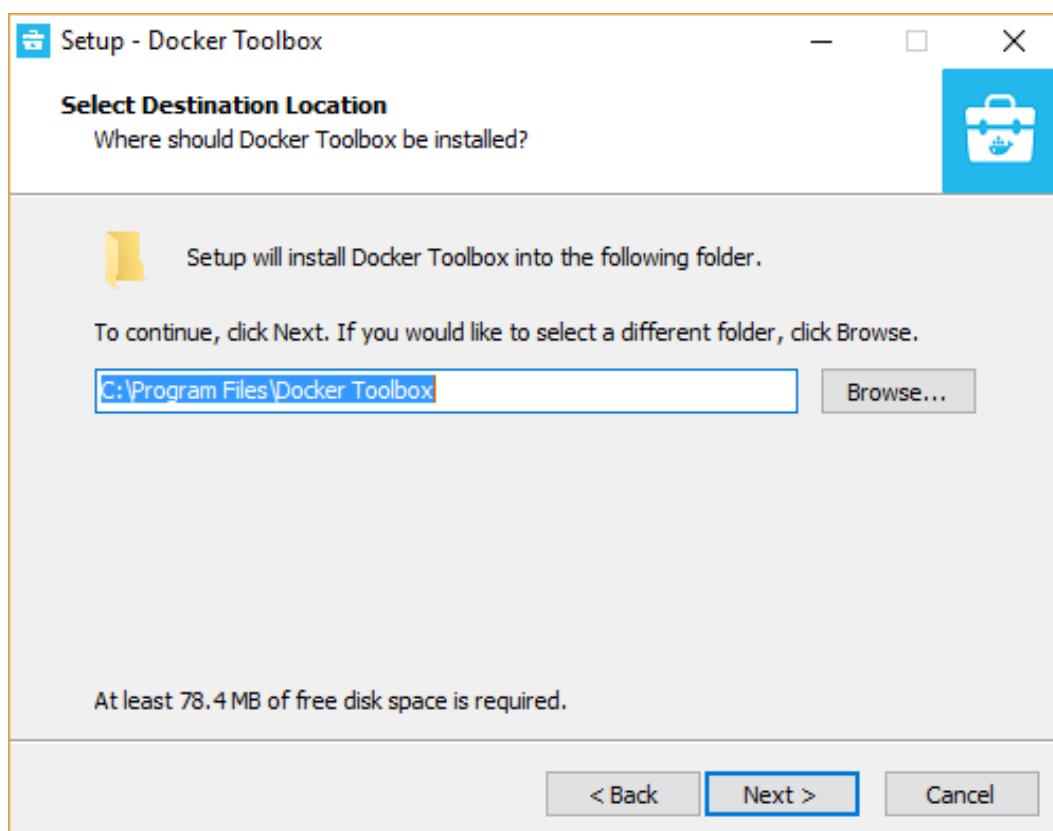
Docker ToolBox

Once the installer has been downloaded, double-click it to start the installer and then follow the steps given below.

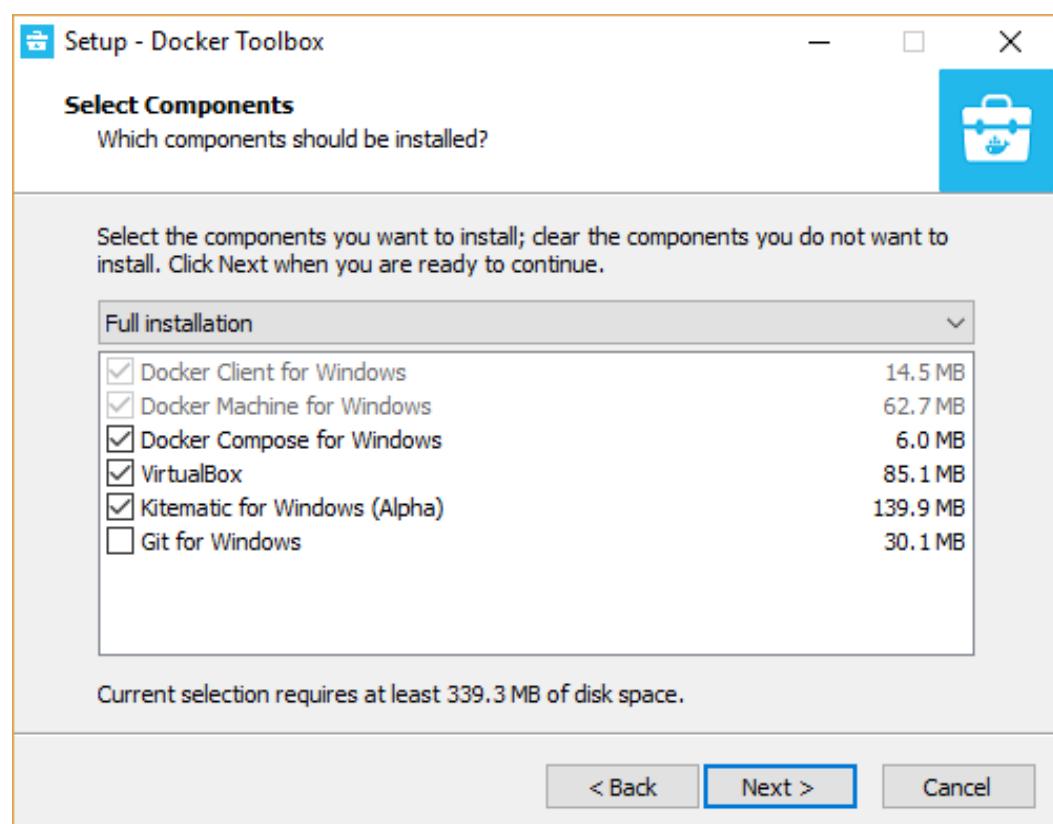
Step 1: Click the Next button on the start screen.



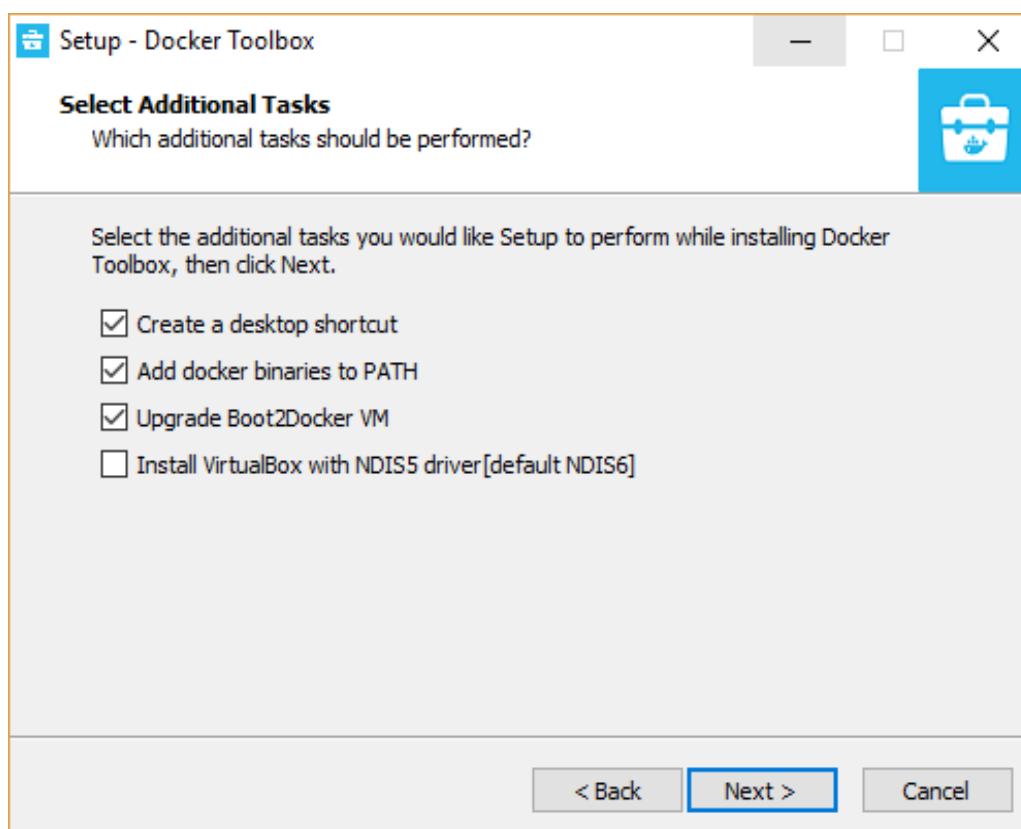
Step 2: Keep the default location on the next screen and click the Next button.



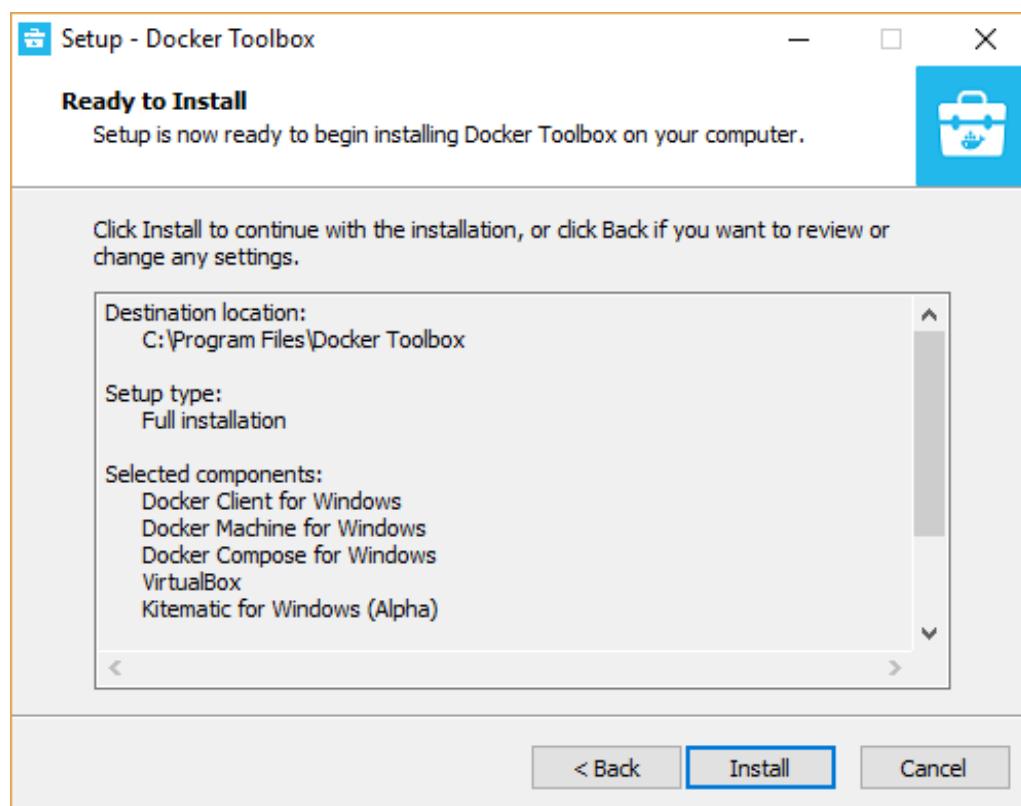
Step 3: Keep the default components and click the Next button to proceed.



Step 4: Keep the Additional Tasks as they are and then click the Next button.



Step 5: On the final screen, click the Install button.



Working with Docker Toolbox

Let's now look at how Docker Toolbox can be used to work with Docker containers on Windows. The first step is to launch the Docker Toolbox application for which the shortcut is created on the desktop when the installation of Docker toolbox is carried out.



Next, you will see the configuration being carried out when Docker toolbox is launched.

```
(default) Starting the VM...
(default) Check network to re-create if needed...
(default) Windows might ask for the permission to create a network adapter. Sometimes, such confirmation window is minimized in the taskbar.
(default) Found a new host-only adapter: "VirtualBox Host-Only Ethernet Adapter #3"
(default) Windows might ask for the permission to configure a network adapter. Sometimes, such confirmation window is minimized in the taskbar.
(default) Windows might ask for the permission to configure a dhcp server. Sometimes, such confirmation window is minimized in the taskbar.
(default) Waiting for an IP...
```

Once done, you will see Docker configured and launched. You will get an interactive shell for Docker.

To test that Docker runs properly, we can use the Docker **run command** to download and run a simple **HelloWorld Docker container**.

The working of the Docker **run command** is given below:

docker run

This command is used to run a command in a Docker container.

Syntax

docker run image

Options

- **Image** – This is the name of the image which is used to run the container.

Return Value

The output will run the command in the desired container.

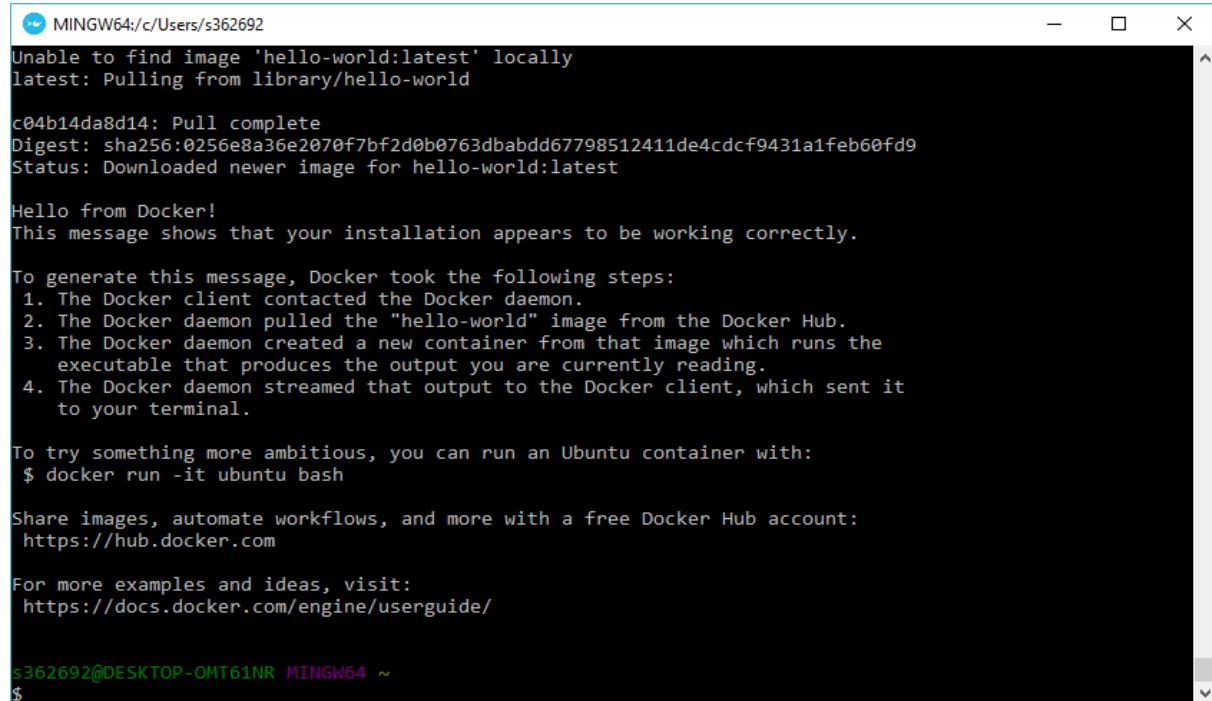
Example

```
sudo docker run hello-world
```

This command will download the **hello-world** image, if it is not already present, and run the **hello-world** as a container.

Output

When we run the above command, we will get the following result:



```
MINGW64:/c/Users/s362692
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world

c04b14da8d14: Pull complete
Digest: sha250:0256e8a36e2070f7bf2d0b0763dbabdd67798512411de4cdcf9431a1feb60fd9
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
 executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
 to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
https://hub.docker.com

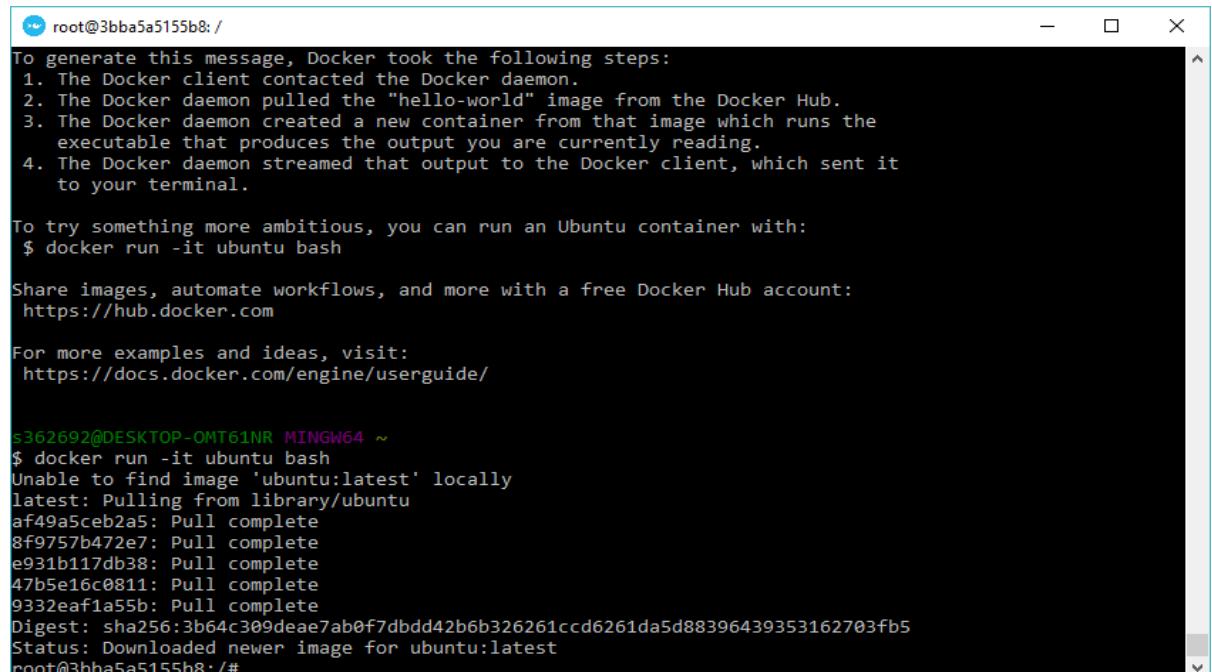
For more examples and ideas, visit:
https://docs.docker.com/engine/userguide/
```

s362692@DESKTOP-OMT61NR MINGW64 ~
\$

If you want to run the Ubuntu OS on Windows, you can download the Ubuntu Image using the following command:

Docker run -it Ubuntu bash

Here you are telling Docker to run the command in the interactive mode via the **-it** option.



```
root@3bba5a5155b8:/
To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
 executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
 to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
https://hub.docker.com

For more examples and ideas, visit:
https://docs.docker.com/engine/userguide/
```

s362692@DESKTOP-OMT61NR MINGW64 ~
\$ docker run -it ubuntu bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
af49a5ceb2a5: Pull complete
8f0757b472e7: Pull complete
e931b117db38: Pull complete
47b5e16c0811: Pull complete
9332eaf1a55b: Pull complete
Digest: sha256:3b64c309deae7ab0f7dbdd42b6b326261cccd6261da5d88396439353162703fb5
Status: Downloaded newer image for ubuntu:latest
root@3bba5a5155b8:/#

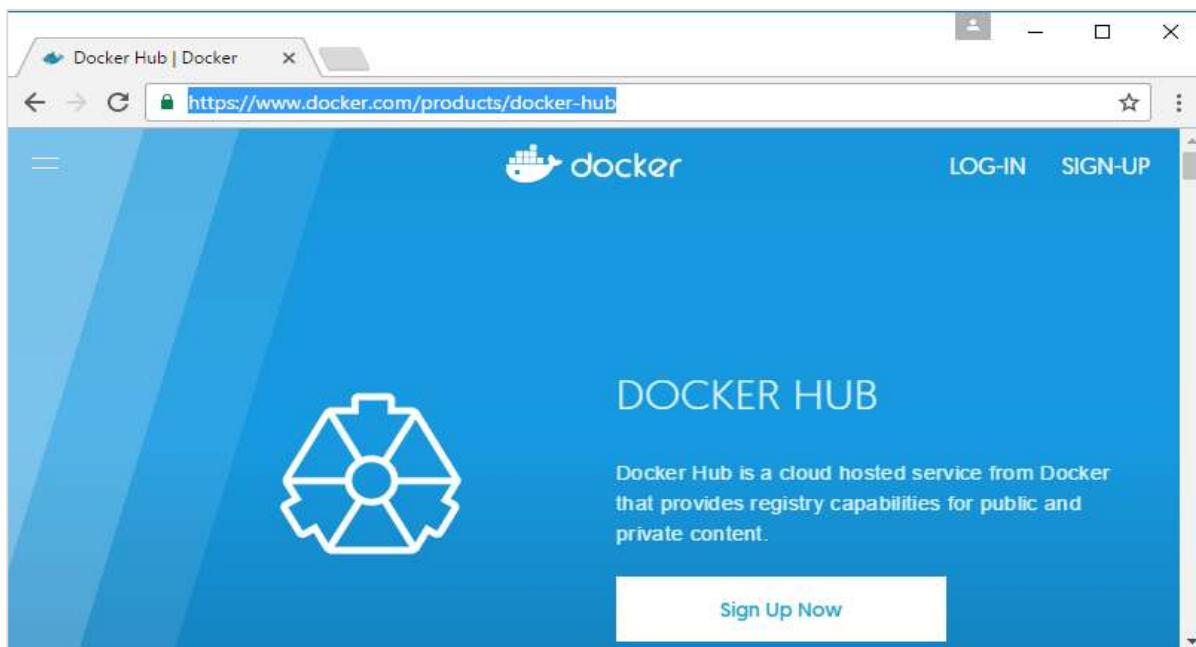
In the output you can see that the Ubuntu image is downloaded and run and then you will be logged in as a root user in the Ubuntu container.

4. Docker – Docker Hub

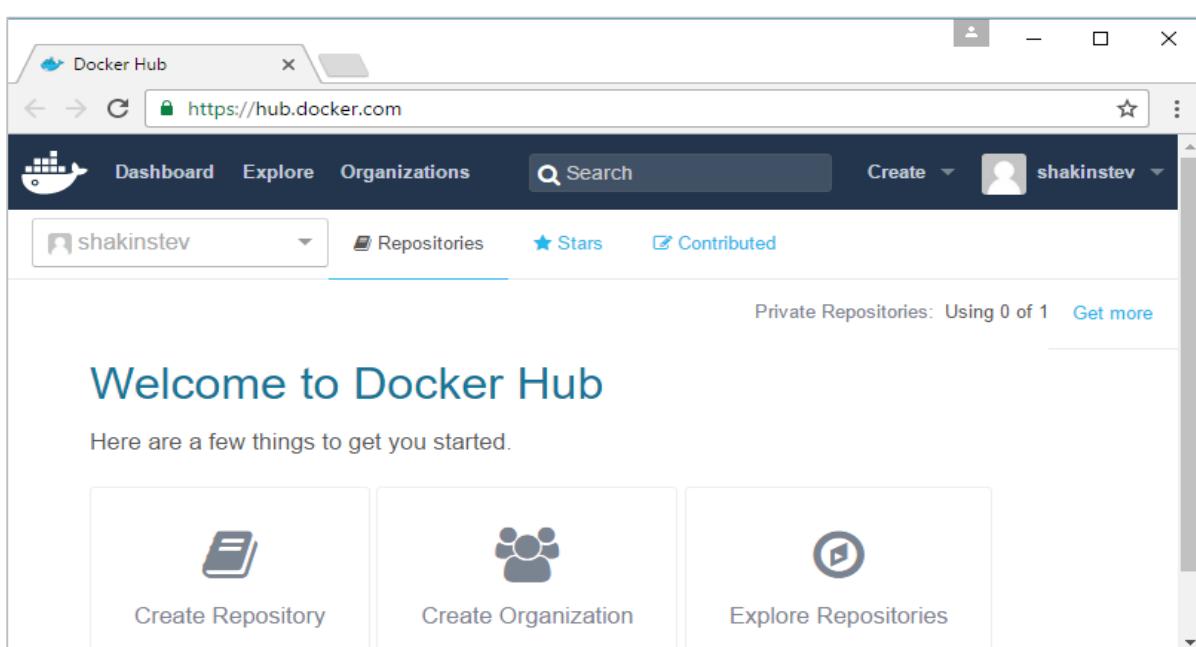
Docker Hub is a registry service on the cloud that allows you to download Docker images that are built by other communities. You can also upload your own Docker built images to Docker hub. In this chapter, we will see how to download and the use the Jenkins Docker image from Docker hub.

The official site for Docker hub is: <https://www.docker.com/products/docker-hub>

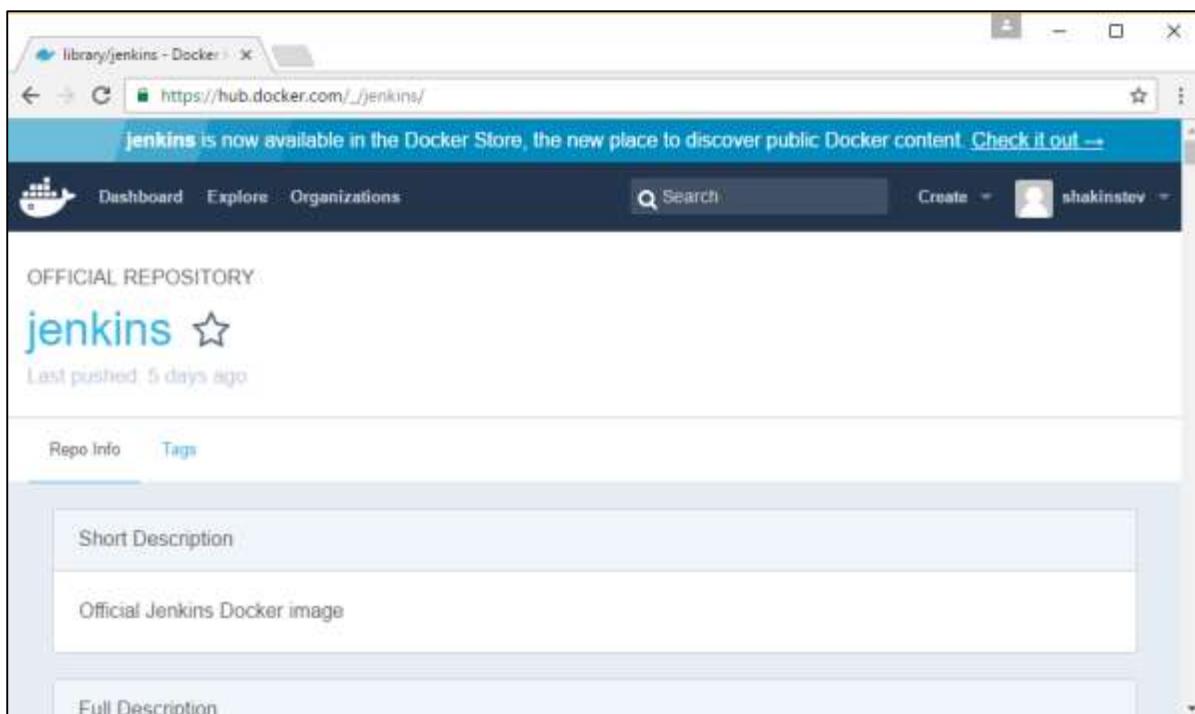
Step 1: First you need to do a simple sign-up on Docker hub.



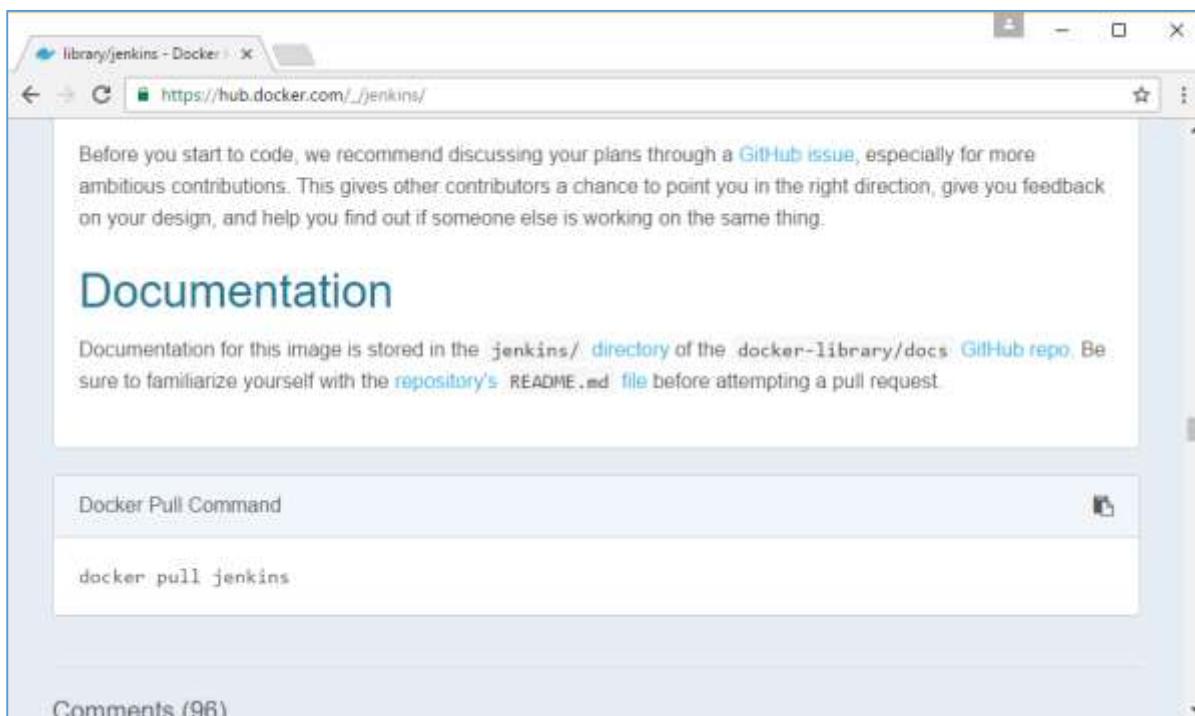
Step 2: Once you have signed up, you will be logged into Docker Hub.



Step 3: Next, let's browse and find the Jenkins image.



Step 4: If you scroll down on the same page, you can see the Docker **pull** command. This will be used to download the Jenkins image onto the local Ubuntu server.



Step 5: Now, go to the Ubuntu server and run the following command:

```
sudo docker pull jenkins
```

```
a079defbaeff: Pull complete
66181a89effa: Pull complete
f4d8f7d94b9c: Pull complete
98e5c3e08215: Pull complete
992fde8f3336: Pull complete
65b58e072756: Pull complete
0b0b6d6525a1: Pull complete
4e7171e4505a: Pull complete
469745638476: Pull complete
49d5aaafff78: Pull complete
c01281524fd6: Pull complete
00a759703a0b: Pull complete
da411a858795: Pull complete
7b8a0b4fd7d0: Pull complete
cbd9e145ea6b: Pull complete
700f8f527cd7: Pull complete
88d27231965c: Pull complete
a067af206313: Pull complete
211049e028a4: Pull complete
7249723069d8: Pull complete
6465c437f020: Pull complete
954c67861e66: Pull complete
6a14c8afbb3a: Pull complete
ec070f7e511e: Pull complete
983246da862f: Pull complete
998d1854867e: Pull complete
Digest: sha256:878e055f96c90af9281fd859f7c69ac289e0178594ff36bbb85e53b789691bec
Status: Downloaded newer image for jenkins:latest
demo@ubuntuserver:~$
```

To run Jenkins, you need to run the following command:

```
sudo docker run -p 8080:8080 -p 50000:50000 jenkins
```

Note the following points about the above **sudo** command:

- We are using the **sudo** command to ensure it runs with **root** access.
- Here, **jenkins** is the name of the image we want to download from Docker hub and install on our Ubuntu machine.
- **-p** is used to map the port number of the internal Docker image to our main Ubuntu server so that we can access the container accordingly.

```
*****
***** Jenkins initial setup is required. An admin user has been created and a password
***** generated.
***** Please use the following password to proceed to installation:
***** 69a504bd19634390b4e67fdd0a908e67
***** This may also be found at: /var/jenkins_home/secrets/initialAdminPassword
*****
***** -->
***** setting agent port for jnlp
***** setting agent port for jnlp... done
Dec 01, 2016 8:16:21 PM hudson.model.UpdateSite updateData
INFO: Obtained the latest update center data file for UpdateSource default
Dec 01, 2016 8:16:22 PM hudson.model.UpdateSite updateData
INFO: Obtained the latest update center data file for UpdateSource default
Dec 01, 2016 8:16:22 PM hudson.model.DownloadService$Downloadable load
INFO: Obtained the updated data file for hudson.tasks.Maven.MavenInstaller
Dec 01, 2016 8:16:22 PM hudson.WebAppMain$3 run
INFO: Jenkins is fully up and running
Dec 01, 2016 8:16:25 PM hudson.model.DownloadService$Downloadable load
INFO: Obtained the updated data file for hudson.tools.JDKInstaller
Dec 01, 2016 8:16:25 PM hudson.model.AsyncPeriodicWork$1 run
INFO: Finished Download metadata. 18,218 ms
```

You will then have Jenkins successfully running as a container on the Ubuntu machine.

5. Docker – Images

In Docker, everything is based on Images. An image is a combination of a file system and parameters. Let's take an example of the following command in Docker.

```
docker run hello-world
```

- The Docker command is specific and tells the Docker program on the Operating System that something needs to be done.
- The **run** command is used to mention that we want to create an instance of an image, which is then called a **container**.
- Finally, "hello-world" represents the image from which the container is made.

Now let's look at how we can use the CentOS image available in Docker Hub to run CentOS on our Ubuntu machine. We can do this by executing the following command on our Ubuntu machine:

```
sudo docker run centos -it /bin/bash
```

Note the following points about the above **sudo** command:

- We are using the **sudo** command to ensure that it runs with **root** access.
- Here, **centos** is the name of the image we want to download from Docker Hub and install on our Ubuntu machine.
- **-it** is used to mention that we want to run in **interactive mode**.
- **/bin/bash** is used to run the bash shell once CentOS is up and running.

Displaying Docker Images

To see the list of Docker images on the system, you can issue the following command.

```
docker images
```

This command is used to display all the images currently installed on the system.

Syntax

```
docker images
```

Options

None

Return Value

The output will provide the list of images on the system.

Example

```
sudo docker images
```

Output

When we run the above command, it will produce the following result:

```
demo@ubuntuserver:~$ sudo docker images
[sudo] password for demo:
REPOSITORY          TAG      IMAGE ID      CREATED
newcentos           latest   7a86f8ffcb25    9 days ago
196.5 MB
jenkins             latest   998d1854867e    2 weeks ago
714.1 MB
centos              latest   97cad5e16cb6    4 weeks ago
196.5 MB
demo@ubuntuserver:~$ _
```

From the above output, you can see that the server has three images: **centos**, **newcentos**, and **jenkins**. Each image has the following attributes:

- **TAG** – This is used to logically tag images.
- **Image ID** – This is used to uniquely identify the image.
- **Created** – The number of days since the image was created.
- **Virtual Size** – The size of the image.

Downloading Docker Images

Images can be downloaded from Docker Hub using the Docker **run** command. Let's see in detail how we can do this.

Syntax

The following syntax is used to run a command in a Docker container.

```
docker run image
```

Options

- **Image** – This is the name of the image which is used to run the container.

Return Value

The output will run the command in the desired container.

Example

```
sudo docker run centos
```

This command will download the **centos** image, if it is not already present, and run the OS as a container.

Output

When we run the above command, we will get the following result:

```
demo@ubuntuserver:~$ sudo docker run centos
Unable to find image 'centos:latest' locally
latest: Pulling from centos

3690474eb5b4: Pull complete
af0819ed1fac: Pull complete
05fe84bf6d3f: Pull complete
97cad5e16cb6: Pull complete
Digest: sha256:934ff980b04db1b7484595bac0c8e6f838e1917ad3a38f904ece64f70bbca040
Status: Downloaded newer image for centos:latest
demo@ubuntuserver:~$ _
```

You will now see the CentOS Docker image downloaded. Now, if we run the Docker **images** command to see the list of images on the system, we should be able to see the **centos** image as well.

```
demo@ubuntuserver:~$ sudo docker run centos
Unable to find image 'centos:latest' locally
latest: Pulling from centos

3690474eb5b4: Pull complete
af0819ed1fac: Pull complete
05fe84bf6d3f: Pull complete
97cad5e16cb6: Pull complete
Digest: sha256:934ff980b04db1b7484595bac0c8e6f838e1917ad3a38f904ece64f70bbca040
Status: Downloaded newer image for centos:latest
demo@ubuntuserver:~$ sudo docker images
REPOSITORY          TAG           IMAGE ID            CREATED             VIRTUAL SIZE
jenkins            latest         998d1854867e      2 weeks ago        714.1 MB
centos             latest         97cad5e16cb6      4 weeks ago       196.5 MB
demo@ubuntuserver:~$
```

Removing Docker Images

The Docker images on the system can be removed via the **docker rmi** command. Let's look at this command in more detail.

```
docker rmi
```

This command is used to remove Docker images.

Syntax

```
docker rmi ImageID
```

Options

- **ImageID** – This is the ID of the image which needs to be removed.

Return Value

The output will provide the Image ID of the deleted Image.

Example

```
sudo docker rmi 7a86f8ffcb25
```

Here, **7a86f8ffcb25** is the Image ID of the **newcentos** image.

Output

When we run the above command, it will produce the following result:

```
demo@ubuntuserver:~$ sudo docker rmi 7a86f8ffcb25
Untagged: newcentos:latest
Deleted: 7a86f8ffcb258e42c11d971a04b1145151b80122e566bc2b544f8fc3f94caf1e
demo@ubuntuserver:~$
```

Let's see some more Docker commands on images.

docker images -q

This command is used to return only the Image ID's of the images.

Syntax

```
docker images -q
```

Options

- **-q** – It tells the Docker command to return the Image ID's only.

Return Value

The output will show only the Image ID's of the images on the Docker host.

Example

```
sudo docker images -q
```

Output

When we run the above command, it will produce the following result:

```
demo@ubuntuserver:~$ sudo docker images -q
998d1854867e
97cad5e16cb6
demo@ubuntuserver:~$ _
```

docker inspect

This command is used see the details of an image or container.

Syntax

```
docker inspect Repository
```

Options

- **Repository** – This is the name of the Image.

Return Value

The output will show detailed information on the Image.

Example

```
sudo docker inspect jenkins
```

Output

When we run the above command, it will produce the following result:

```
"Hostname": "6b3797ab1e90",
"Image": "sha256:532b1ef702484a402708f3b65a61e6ddf307bbf2fdfa01be553bc6c
a7678ce6c",
"Labels": {},
"MacAddress": "",
"Memory": 0,
"MemorySwap": 0,
"NetworkDisabled": false,
"UnBuild": [],
"OpenStdin": false,
"PortSpecs": null,
"StdinOnce": false,
"Tty": false,
"User": "jenkins",
"Volumes": {
    "/var/jenkins_home": {}
},
"WorkingDir": "",

},
"Created": "2016-11-16T20:52:37.568557509Z",
"DockerVersion": "1.12.3",
"Id": "998d1854867eb7873a9f45ff4c3ab25bcf5378c77fc955d344e47cb27e5df723",
"Os": "linux",
"Parent": "983246da862f43a967b36cc2fc1af580df3f79760df841c1954e73253014ecee"
,
"Size": 5960,
"VirtualSize": 714121162
}
]
demo@ubuntuserver:~$ _
```

6. Docker – Containers

Containers are instances of Docker images that can be run using the Docker **run** command. The basic purpose of Docker is to run containers. Let's discuss how to work with containers.

Running a Container

Running of containers is managed with the Docker **run** command. To run a container in an interactive mode, first launch the Docker container.

```
sudo docker run -it centos /bin/bash
```

Then hit Ctrl+p and you will return to your OS shell.

```
demo@ubuntuserver:~$ sudo docker run -it centos /bin/bash  
[root@9f215ed0b0d3 ~]#
```

You will then be running in the instance of the CentOS system on the Ubuntu server.

Listing of Containers

One can list all of the containers on the machine via the **docker ps** command. This command is used to return the currently running containers.

```
docker ps
```

Syntax

```
docker ps
```

Options

None

Return Value

The output will show the currently running containers.

Example

```
sudo docker ps
```

Output

When we run the above command, it will produce the following result:

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
9f215ed0b0d3	centos:latest	"/bin/bash"	About a minute ago
Up About a minute		cocky_colden	
demo@ubuntuserver:~\$			

Let's see some more variations of the **docker ps** command.

docker ps -a

This command is used to list all of the containers on the system.

Syntax

```
docker ps -a
```

Options

- a** – It tells the **docker ps** command to list all of the containers on the system.

Return Value

The output will show all containers.

Example

```
sudo docker ps -a
```

Output

When we run the above command, it will produce the following result:

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS		
9f215ed0b0d3	centos:latest	"/bin/bash"	4 minutes ago
Up 4 minutes			
cocky_colden			
e5a02936065a	centos:latest	"/bin/bash"	39 minutes ago
Exited (0) 39 minutes ago			
ecstatic_hodgkin			
9b286dd1f16a	jenkins:latest	"/bin/tini -- /usr/l	18 hours ago
Exited (0) About an hour ago	0.0.0.0:8080->8080/tcp, 0.0.0.0:50000->50000		
cp_jolly_wright			
3646aa260a2d	jenkins:latest	"/bin/tini -- /usr/l	9 days ago
Exited (0) 9 days ago	0.0.0.0:8080->8080/tcp, 0.0.0.0:50000->50000		
cp_reverent_morse			
demo@ubuntuserver:~\$	_		

docker history

With this command, you can see all the commands that were run with an image via a container.

Syntax

```
docker history ImageID
```

Options

- **ImageID** – This is the Image ID for which you want to see all the commands that were run against it.

Return Value

The output will show all the commands run against that image.

Example

```
sudo docker history centos
```

The above command will show all the commands that were run against the **centos** image.

Output

When we run the above command, it will produce the following result:

```
demo@ubuntuserver:~$ sudo docker images
REPOSITORY          TAG      IMAGE ID      CREATED
VIRTUAL SIZE
jenkins            latest   998d1854867e    2 weeks ago
714.1 MB
centos              latest   97cad5e16cb6    4 weeks ago
196.5 MB
demo@ubuntuserver:~$ sudo docker history centos
IMAGE      CREATED      CREATED BY
SIZE
97cad5e16cb6    4 weeks ago    /bin/sh -c #(nop)  CMD ["/bin/bash"]
0 B
05fe84bf6d3f    4 weeks ago    /bin/sh -c #(nop)  LABEL name=CentOS B
e Ima 0 B
af0819ed1fac    4 weeks ago    /bin/sh -c #(nop) ADD file:54df3580ac9
66389  196.5 MB
3690474eb5b4    3 months ago   /bin/sh -c #(nop)  MAINTAINER https://
thub. 0 B
demo@ubuntuserver:~$ _
```

7. Docker – Working with Containers

In this chapter, we will explore in detail what we can do with containers.

docker top

With this command, you can see the top processes within a container.

Syntax

```
docker top ContainerID
```

Options

- ContainerID – This is the Container ID for which you want to see the top processes.

Return Value

The output will show the top-level processes within a container.

Example

```
sudo docker top 9f215ed0b0d3
```

The above command will show the top-level processes within a container.

Output

When we run the above command, it will produce the following result:

```
demo@ubuntuserver:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES
9f215ed0b0d3        centos:latest      "/bin/bash"         12 minutes ago   Up 12 minutes       cocky_colden
demo@ubuntuserver:~$ sudo docker top 9f215ed0b0d3
JID           PID    PPID   TIME          CMD
root          1606   678    00:00:00   /bin/bash
demo@ubuntuserver:~$
```

docker stop

This command is used to stop a running container.

Syntax

```
docker stop ContainerID
```

Options

- **ContainerID** – This is the Container ID which needs to be stopped.

Return Value

The output will give the ID of the stopped container.

Example

```
sudo docker stop 9f215ed0b0d3
```

The above command will stop the Docker container **9f215ed0b0d3**.

Output

When we run the above command, it will produce the following result:

```
demo@ubuntuserver:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES
9f215ed0b0d3        centos:latest      "/bin/bash"        22 minutes ago   Up 22 minutes       cocky_colden
demo@ubuntuserver:~$ sudo docker stop 9f215ed0b0d3
9f215ed0b0d3
demo@ubuntuserver:~$ sudo docker rm 9f215ed0b0d3
9f215ed0b0d3
demo@ubuntuserver:~$ _
```

docker rm

This command is used to delete a container.

Syntax

```
docker rm ContainerID
```

Options

- **ContainerID** – This is the Container ID which needs to be removed.

Return Value

The output will give the ID of the removed container.

Example

```
sudo docker rm 9f215ed0b0d3
```

The above command will remove the Docker container **9f215ed0b0d3**.

Output

When we run the above command, it will produce the following result:

```
demo@ubuntuserver:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES
9f215ed0b0d3        centos:latest      "/bin/bash"         22 minutes ago   Up 22 minutes   cocky_colden
demo@ubuntuserver:~$ sudo docker stop 9f215ed0b0d3
9f215ed0b0d3
demo@ubuntuserver:~$ sudo docker rm 9f215ed0b0d3
9f215ed0b0d3
demo@ubuntuserver:~$ _
```

docker stats

This command is used to provide the statistics of a running container.

Syntax

```
docker stats ContainerID
```

Options

- **ContainerID** – This is the Container ID for which the stats need to be provided.

Return Value

The output will show the CPU and Memory utilization of the Container.

Example

```
sudo docker rm 9f215ed0b0d3
```

The above command will provide CPU and memory utilization of the Container **9f215ed0b0d3**.

Output

When we run the above command, it will produce the following result:

CONTAINER	CPU %	MEM USAGE/LIMIT	MEM %
NET I/O 07b0b6f434fe 648 B/648 B	0.00%	416 KiB/1.416 GiB	0.03%

docker attach

This command is used to attach to a running container.

Syntax

```
docker attach ContainerID
```

Options

- **ContainerID** – This is the Container ID to which you need to attach.

Return Value

None

Example

```
sudo docker attach 07b0b6f434fe
```

The above command will attach to the Docker container **07b0b6f434fe**.

Output

When we run the above command, it will produce the following result:

```
demo@ubuntuserver:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES
07b0b6f434fe        centos:latest      "/bin/bash"         3 minutes ago    Up 3 minutes       cocky_pare
demo@ubuntuserver:~$ sudo docker attach 07b0b6f434fe
[root@07b0b6f434fe ~]# _
```

Once you have attached to the Docker container, you can run the above command to see the process utilization in that Docker container.

```
top - 15:24:06 up 2:06, 0 users, load average: 0.00, 0.01, 0.02
Tasks: 2 total, 1 running, 1 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.0 us, 0.3 sy, 0.0 ni, 99.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 s
KiB Mem : 1484856 total, 1057152 free, 52368 used, 375336 buff/cache
KiB Swap: 1519612 total, 1519612 free, 0 used. 1403868 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
 1 root 20 0 11784 2992 2644 S 0.0 0.2 0:00.01 bash
 15 root 20 0 51864 3772 3272 R 0.0 0.3 0:00.00 top
```

docker pause

This command is used to pause the processes in a running container.

Syntax

```
docker pause ContainerID
```

Options

- **ContainerID** – This is the Container ID to which you need to pause the processes in the container.

Return Value

The ContainerID of the paused container.

Example

```
sudo docker pause 07b0b6f434fe
```

The above command will pause the processes in a running container **07b0b6f434fe**.

Output

When we run the above command, it will produce the following result:

```
demo@ubuntuserver:~$ sudo docker ps
[sudo] password for demo:
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              NAMES
07b0b6f434fe        centos:latest      "/bin/bash"            18 minutes ago   Up 18 minutes   cocky_pare
demo@ubuntuserver:~$ sudo docker pause 07b0b6f434fe
07b0b6f434fe
demo@ubuntuserver:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              NAMES
07b0b6f434fe        centos:latest      "/bin/bash"            19 minutes ago   Up 19 minutes (Paused)   cocky_pare
demo@ubuntuserver:~$ _
```

docker unpause

This command is used to **unpause** the processes in a running container.

Syntax

```
docker unpause ContainerID
```

Options

- **ContainerID** – This is the Container ID to which you need to unpause the processes in the container.

Return Value

The ContainerID of the running container.

Example

```
sudo docker unpause 07b0b6f434fe
```

The above command will unpause the processes in a running container: 07b0b6f434fe

Output

When we run the above command, it will produce the following result:

```
demo@ubuntuserver:~$ sudo docker unpause 07b0b6f434fe
07b0b6f434fe
demo@ubuntuserver:~$
```

docker kill

This command is used to kill the processes in a running container.

Syntax

```
docker kill ContainerID
```

Options

- **ContainerID** – This is the Container ID to which you need to kill the processes in the container.

Return Value

The ContainerID of the running container.

Example

```
sudo docker kill 07b0b6f434fe
```

The above command will kill the processes in the running container **07b0b6f434fe**.

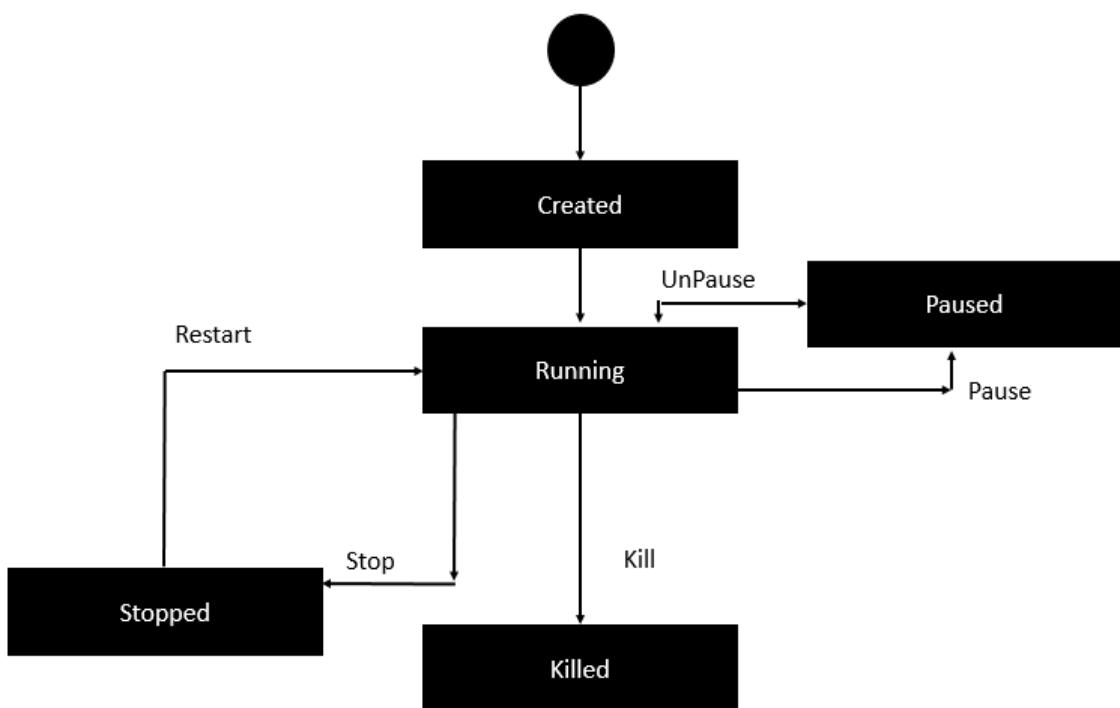
Output

When we run the above command, it will produce the following result:

```
demo@ubuntuserver:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES
07b0b6f434fe        centos:latest      "/bin/bash"        23 minutes ago   Up 23 minutes   cocky_pare
demo@ubuntuserver:~$ sudo docker kill 07b0b6f434fe
07b0b6f434fe
demo@ubuntuserver:~$
```

Docker – Container Lifecycle

The following illustration explains the entire lifecycle of a Docker container.

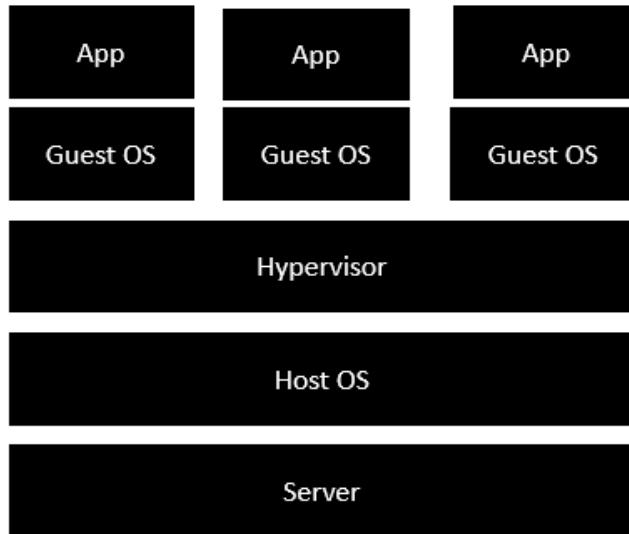


- Initially, the Docker container will be in the **created** state.
- Then the Docker container goes into the **running** state when the Docker **run** command is used.
- The Docker **kill** command is used to kill an existing Docker container.
- The Docker **pause** command is used to pause an existing Docker container.

- The Docker **stop** command is used to pause an existing Docker container.
- The Docker **run** command is used to put a container back from a **stopped** state to a **running** state.

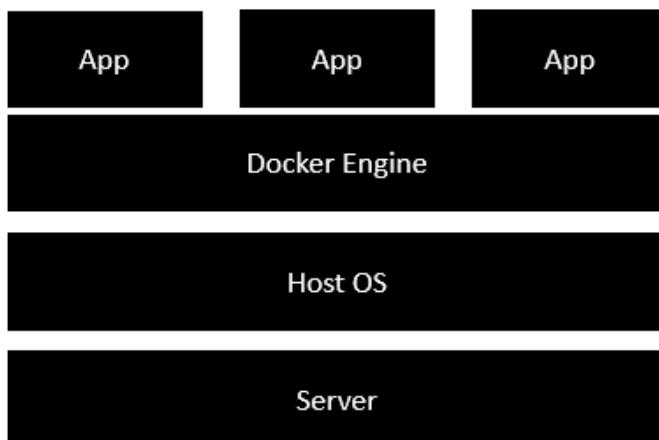
8. Docker – Docker Architecture

The following image shows the standard and traditional architecture of **virtualization**.



- The server is the physical server that is used to host multiple virtual machines.
- The Host OS is the base machine such as Linux or Windows.
- The Hypervisor is either VMWare or Windows Hyper V that is used to host virtual machines.
- You would then install multiple operating systems as virtual machines on top of the existing hypervisor as Guest OS.
- You would then host your applications on top of each Guest OS.

The following image shows the new generation of virtualization that is enabled via Dockers. Let's have a look at the various layers.



- The server is the physical server that is used to host multiple virtual machines. So this layer remains the same.
- The Host OS is the base machine such as Linux or Windows. So this layer remains the same.
- Now comes the new generation which is the Docker engine. This is used to run the operating system which earlier used to be virtual machines as Docker containers.
- All of the Apps now run as Docker containers.

The clear advantage in this architecture is that you don't need to have extra hardware for Guest OS. Everything works as Docker containers.

9. Docker – Container and Hosts

The good thing about the Docker engine is that it is designed to work on various operating systems. We have already seen the installation on Windows and seen all the Docker commands on Linux systems. Now let's see the various Docker commands on the Windows OS.

Docker Images

Let's run the Docker **images** command on the Windows host.

From here, we can see that we have two images: **ubuntu** and **hello-world**.

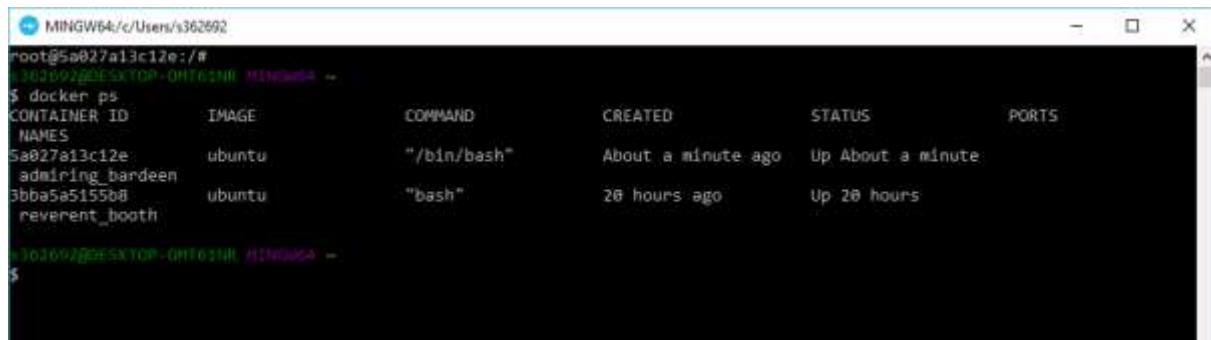
Running a Container

Now let's run a container in the Windows Docker host.

We can see that by running the container, we can now run the Ubuntu container on a Windows host.

Listing All Containers

Let's list all the containers on the Windows host.

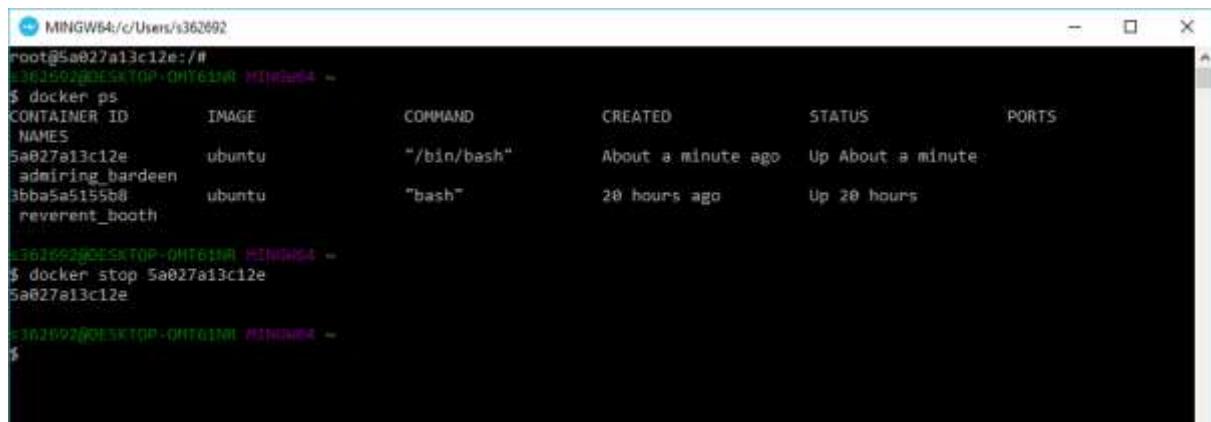


```
MINGW64:/c/Users/s362692
root@5a027a13c12e:#
e362692@DESKTOP-OHT61NR MINGW64 ~
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
 NAMES
5a027a13c12e        ubuntu              "/bin/bash"         About a minute ago   Up About a minute
admiring_bardeen
3bba5a5155b8        ubuntu              "bash"              20 hours ago       Up 20 hours
reverent_booth

e362692@DESKTOP-OHT61NR MINGW64 ~
$
```

Stopping a Container

Let's now stop a running container on the Windows host.



```
MINGW64:/c/Users/s362692
root@5a027a13c12e:#
e362692@DESKTOP-OHT61NR MINGW64 ~
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
 NAMES
5a027a13c12e        ubuntu              "/bin/bash"         About a minute ago   Up About a minute
admiring_bardeen
3bba5a5155b8        ubuntu              "bash"              20 hours ago       Up 20 hours
reverent_booth

e362692@DESKTOP-OHT61NR MINGW64 ~
$ docker stop 5a027a13c12e
5a027a13c12e

e362692@DESKTOP-OHT61NR MINGW64 ~
$
```

So you can see that the Docker engine is pretty consistent when it comes to different Docker hosts and it works on Windows in the same way it works on Linux.

10. Docker – Configuring Docker

In this chapter, we will look at the different options to configure Docker.

service docker stop

This command is used to stop the Docker **daemon** process.

Syntax

```
service docker stop
```

Options

None

Return Value

A message showing that the Docker process has stopped.

Example

```
sudo service docker stop
```

Output

When we run the above command, it will produce the following result:

```
demo@ubuntuserver:~$ sudo service docker stop
[sudo] password for demo:
docker stop/waiting
demo@ubuntuserver:~$
```

service docker start

This command is used to start the Docker daemon process.

Syntax

```
service docker start
```

Options

None

Return Value

A message showing that the Docker process has started.

Example

```
sudo service docker start
```

Output

When we run the above command, it will produce the following result:

```
demo@ubuntuserver:~$ sudo service docker stop
[sudo] password for demo:
docker stop/waiting
demo@ubuntuserver:~$ sudo service docker start
docker start/running, process 1942
demo@ubuntuserver:~$ _
```

11. Docker – Containers and Shells

By default, when you launch a container, you will also use a **shell command** while launching the container as shown below. This is what we have seen in the earlier chapters when we were working with containers.

```
demo@ubuntuserver:~$ sudo docker run -it centos /bin/bash
[root@d76d00fbce4c ~]# demo@ubuntuserver:~$ 
demo@ubuntuserver:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES
d76d00fbce4c        centos:latest      "/bin/bash"        7 seconds ago    Up 6 seconds   boring_goldstine
demo@ubuntuserver:~$
```

In the above screenshot, you can observe that we have issued the following command:

```
sudo docker run -it centos /bin/bash
```

We used this command to create a new container and then used the Ctrl+P+Q command to exit out of the container. It ensures that the container still exists even after we exit from the container.

We can verify that the container still exists with the Docker **ps** command. If we had to exit out of the container directly, then the container itself would be destroyed.

Now there is an easier way to attach to containers and exit them cleanly without the need of destroying them. One way of achieving this is by using the **nsenter** command.

Before we run the **nsenter** command, you need to first install the **nsenter** image. It can be done by using the following command:

```
docker run --rm -v /usr/local/bin:/target jpetazzo/nsenter
```

```
78ddc5c32d95: Pull  complete
f68a85041029: Pull  complete
a82810b57d94: Pull  complete
b7593cf24c20: Pull  complete
0cb2fb8d9656: Pull  complete
f362bdd62dfa: Pull  complete
dc5370e1cb31: Pull  complete
5e3c4cd314f8: Pull  complete
f3e4e1652690: Pull  complete
29c0847bbd97: Pull  complete
5cc9b463abfd: Pull  complete
f36b6bd8d305: Pull  complete
3731cc001b90: Pull  complete
7b05e8e06727: Pull  complete
1a0a7e2eaebc: Pull  complete
a249cf324221: Pull  complete
Digest: sha256:17dc70210e269e76d2f99c575bc9788ba9565aa9cb70b7132f895a6e846db86
Status: Downloaded newer image for jpetazzo/nsenter:latest
Installing nsenter to /target
Installing docker-enter to /target
Installing importenv to /target
demo@ubuntudemo:~$ 
demo@ubuntudemo:~$
```

Before we use the **nsenter** command, we need to get the Process ID of the container, because this is required by the **nsenter** command. We can get the Process ID via the Docker **inspect** command and filtering it via the **Pid**

```
root@ubuntudemo:~# sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES
ef42a4c5e663        centos:latest      "/bin/bash"        2 minutes ago    Up 2 minutes       stoic_banach
root@ubuntudemo:~# sudo docker inspect ef42a4c5e663 | grep Pid
    "PidMode": "",  

    "Pid": 2978,  

root@ubuntudemo:~# _
```

As seen in the above screenshot, we have first used the **docker ps** command to see the running containers. We can see that there is one running container with the ID of ef42a4c5e663.

We then use the Docker **inspect** command to inspect the configuration of this container and then use the **grep** command to just filter the Process ID. And from the output, we can see that the Process ID is 2978.

Now that we have the process ID, we can proceed forward and use the **nsenter** command to attach to the Docker container.

nsenter

This method allows one to attach to a container without exiting the container.

Syntax

```
nsenter -m -u -n -p -i -t containerID command
```

Options

- **-u** is used to mention the **Uts namespace**
- **-m** is used to mention the **mount namespace**
- **-n** is used to mention the **network namespace**
- **-p** is used to mention the **process namespace**
- **-i** is to make the container run in interactive mode.
- **-t** is used to connect the I/O streams of the container to the host OS.
- **containerID** – This is the ID of the container.
- **Command** – This is the command to run within the container.

Return Value

None

Example

```
sudo nsenter -m -u -n -p -i -t 2978 /bin/bash
```

Output

```
root@ubuntudemo:~# sudo nsenter -m -u -n -p -i -t 2978 /bin/bash
bash-4.2# exit
exit
root@ubuntudemo:~# sudo docker ps
CONTAINER ID        IMAGE               COMMAND       CREATED          STATUS           NAMES
ef42a4c5e663        centos:latest      "/bin/bash"   9 minutes ago   Up 9 minutes   stoic_banach
root@ubuntudemo:~# _
```

From the output, we can observe the following points:

- The prompt changes to the **bash shell** directly when we issue the **nsenter** command.
- We then issue the **exit** command. Now normally if you did not use the **nsenter** command, the container would be destroyed. But you would notice that when we run the **nsenter** command, the container is still up and running.

12. Docker – Docker File

In the earlier chapters, we have seen the various Image files such as Centos which get downloaded from **Docker hub** from which you can spin up containers. An example is again shown below.

```
root@ubuntudemo:~# sudo docker images
REPOSITORY          TAG      IMAGE ID      CREATED
centos              latest   97cad5e16cb6  4 weeks ago
jpetazzo/nsenter    latest   a249cf324221  4 months ago
root@ubuntudemo:~# _
```

If we use the Docker **images** command, we can see the existing images in our system. From the above screenshot, we can see that there are two images: **centos** and **nsenter**.

But Docker also gives you the capability to create your own Docker images, and it can be done with the help of **Docker Files**. A Docker File is a simple text file with instructions on how to build your images.

The following steps explain how you should go about creating a Docker File.

Step 1: Create a file called **Docker File** and edit it using **vim**. Please note that the name of the file has to be "Dockerfile" with "D" as capital.

```
root@ubuntudemo:~# sudo vim Dockerfile
```

Step 2: Build your Docker File using the following instructions:

```
#This is a sample Image
FROM ubuntu
MAINTAINER demousr@gmail.com
RUN apt-get update
RUN apt-get install -y nginx
CMD ["echo","Image created"]
```

The following points need to be noted about the above file:

- The first line "#This is a sample Image" is a comment. You can add comments to the Docker File with the help of the **#** command.
- The next line has to start with the **FROM** keyword. It tells docker, from which base image you want to base your image from. In our example, we are creating an image from the **ubuntu** image.

- The next command is the person who is going to maintain this image. Here you specify the **MAINTAINER** keyword and just mention the email ID.
- The **RUN** command is used to run instructions against the image. In our case, we first update our Ubuntu system and then install the **nginx** server on our **ubuntu** image.
- The last command is used to display a message to the user.

Step 3: Save the file. In the next chapter, we will discuss how to build the image.

```
#This is a sample Image
FROM ubuntu
MAINTAINER demousr@gmail.com
RUN apt-get update
RUN apt-get install -y nginx
CMD ["echo",""Image created""]

~
```

13. Docker – Building Docker Files

We created our Docker File in the last chapter. It's now time to build the Docker File. The Docker File can be built with the following command:

```
docker build
```

Let's learn more about this command.

docker build

This method allows the users to build their own Docker images.

Syntax

```
docker build -t ImageName:TagName dir
```

Options

- **-t** is to mention a tag to the image
- **ImageName** – This is the name you want to give to your image
- **TagName** – This is the tag you want to give to your image
- **Dir** – The directory where the Docker File is present.

Return Value

None

Example

```
sudo docker build -t myimage:0.1 .
```

Here, **myimage** is the name we are giving to the Image and **0.1** is the tag number we are giving to our image.

Since the Docker File is in the present working directory, we used **"."** at the end of the command to signify the present working directory.

Output

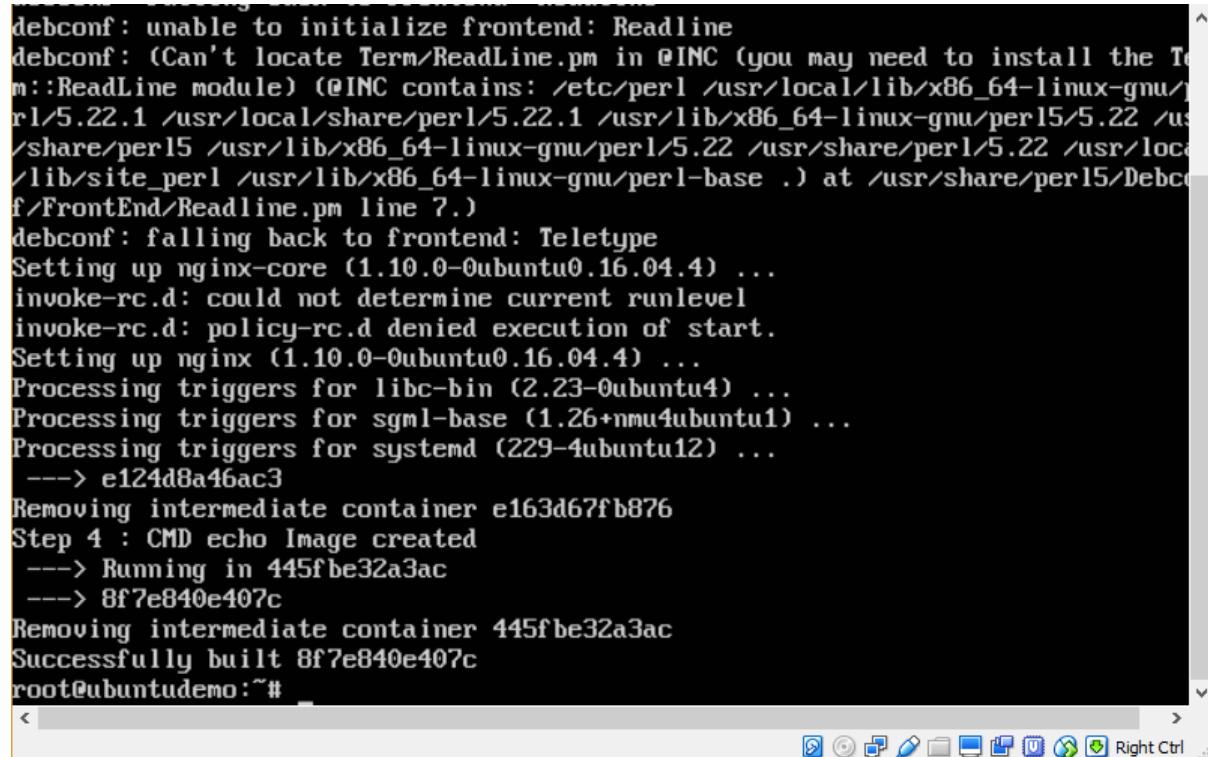
From the output, you will first see that the Ubuntu Image will be downloaded from Docker Hub, because there is no image available locally on the machine.

```
root@ubuntudemo:~# sudo docker build -t myimage:0.1 .
Sending build context to Docker daemon 11.78 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu
latest: Pulling from ubuntu

4d330285ec99: Downloading 2.535 MB/50.1 MB
497dd7934d13: Download complete
bca38844f775: Download complete
e711979f32e2: Download complete
f38e3ca2b73a: Download complete
103d358b91a9: Download complete
```

Finally, when the build is complete, all the necessary commands would have run on the image.

```
debconf: unable to initialize frontend: Readline
debconf: (Can't locate Term/ReadLine.pm in @INC (you may need to install the Term::ReadLine module) (@INC contains: /etc/perl /usr/local/lib/x86_64-linux-gnu/perl/5.22.1 /usr/local/share/perl/5.22.1 /usr/lib/x86_64-linux-gnu/perl5/5.22 /usr/share/perl5 /usr/lib/x86_64-linux-gnu/perl/5.22 /usr/share/perl/5.22 /usr/local/lib/site_perl /usr/lib/x86_64-linux-gnu/perl-base .) at /usr/share/perl5/Debian/DebianFrontEnd/Readline.pm line 7.)
debconf: falling back to frontend: Teletype
Setting up nginx-core (1.10.0-0ubuntu0.16.04.4) ...
invoke-rc.d: could not determine current runlevel
invoke-rc.d: policy-rc.d denied execution of start.
Setting up nginx (1.10.0-0ubuntu0.16.04.4) ...
Processing triggers for libc-bin (2.23-0ubuntu4) ...
Processing triggers for sgml-base (1.26+nmu4ubuntu1) ...
Processing triggers for systemd (229-4ubuntu12) ...
--> e124d8a46ac3
Removing intermediate container e163d67fb876
Step 4 : CMD echo Image created
--> Running in 445fbe32a3ac
--> 8f7e840e407c
Removing intermediate container 445fbe32a3ac
Successfully built 8f7e840e407c
root@ubuntudemo:~#
```



You will then see the successfully built message and the ID of the new Image. When you run the Docker **images command**, you would then be able to see your new image.

```
root@ubuntudemo:~# sudo docker images
REPOSITORY          TAG      IMAGE ID      CREATED
myimage              0.1      8f7e840e407c    About a minute ago
ubuntu               latest   103d358b91a9    6 days ago
centos               latest   97cad5e16cb6    4 weeks ago
jpetazzo/nsenter     latest   a249cf324221    4 months ago
root@ubuntudemo:~#
```

You can now build containers from your new Image.

14. Docker – Public Repositories

Public repositories can be used to host Docker images which can be used by everyone else. An example is the images which are available in Docker Hub. Most of the images such as Centos, Ubuntu, and Jenkins are all publicly available for all. We can also make our images available by publishing it to the public repository on Docker Hub.

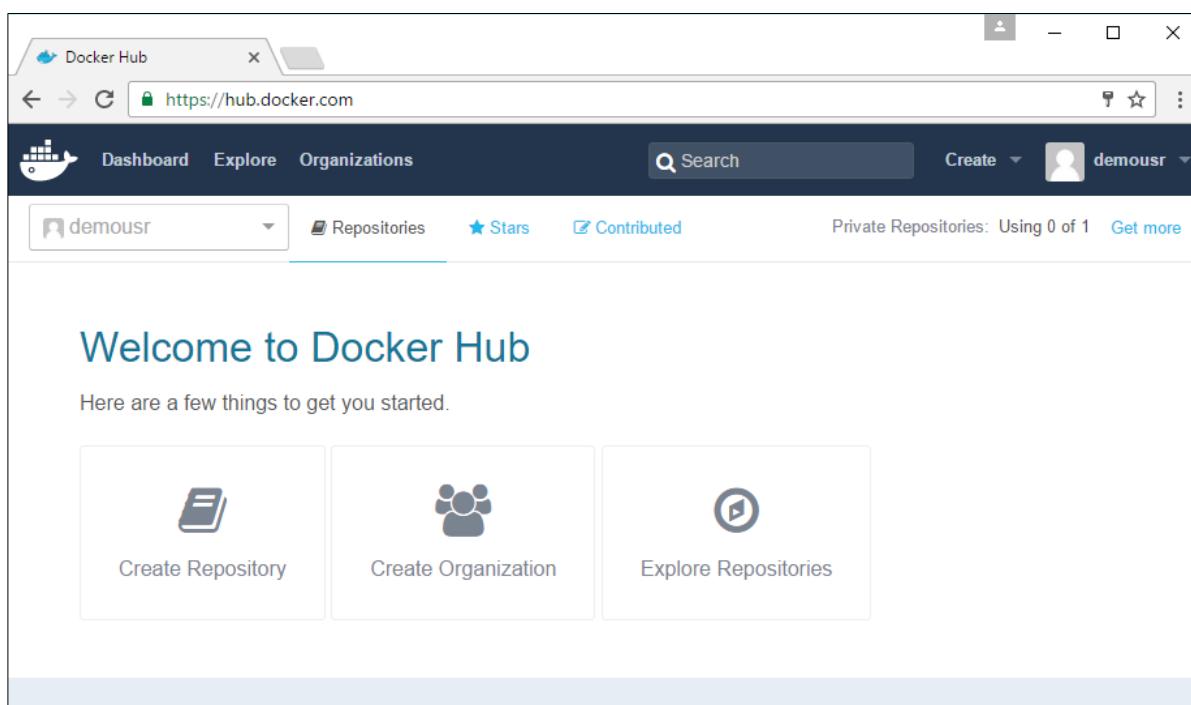
For our example, we will use the **myimage** repository built in the "Building Docker Files" chapter and upload that image to Docker Hub. Let's first review the images on our Docker host to see what we can push to the Docker registry.

```
demo@ubuntudemo:~$ sudo docker images
REPOSITORY          TAG      IMAGE ID      CREATED
SIZE
myimage              0.1      7738aa0e1b36   9 minutes ago
225.3 MB
centos               latest    67591570dd29   2 days ago
191.8 MB
ubuntu               latest    104bec311bcd   2 days ago
129 MB
demo@ubuntudemo:~$ _
```

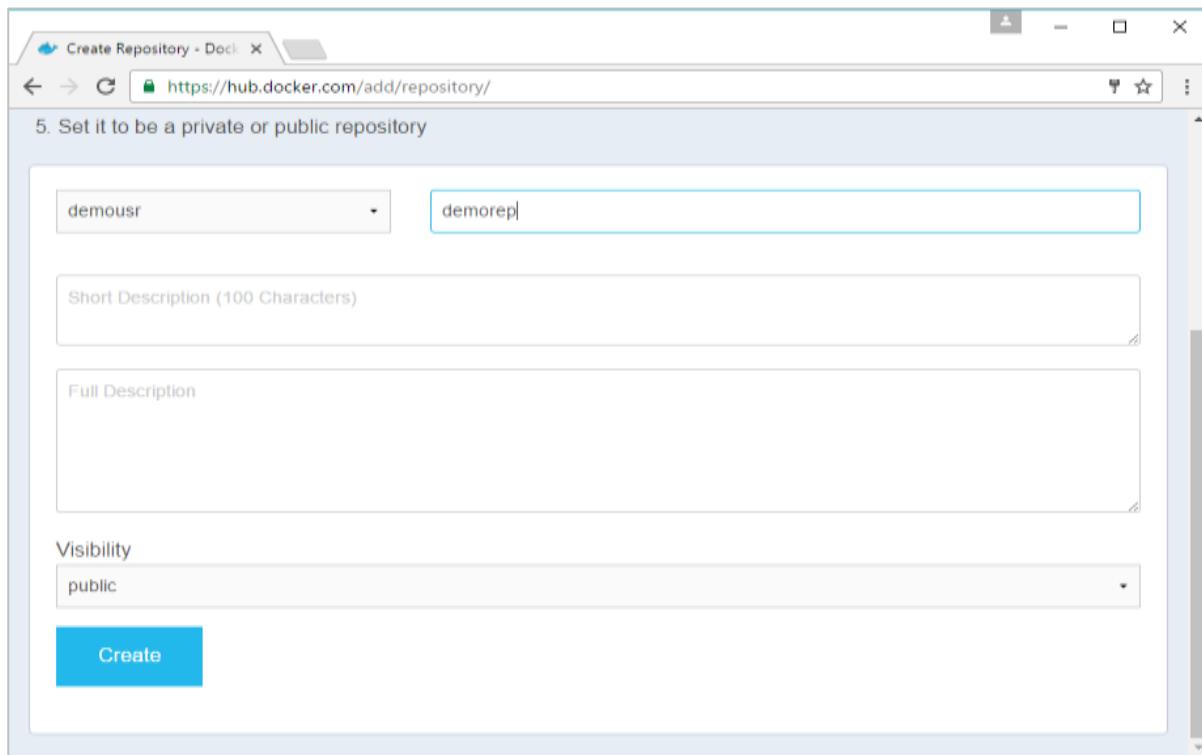
Here, we have our **myimage:0.1** image which was created as a part of the "Building Docker Files" chapter. Let's use this to upload to the Docker public repository.

The following steps explain how you can upload an image to public repository.

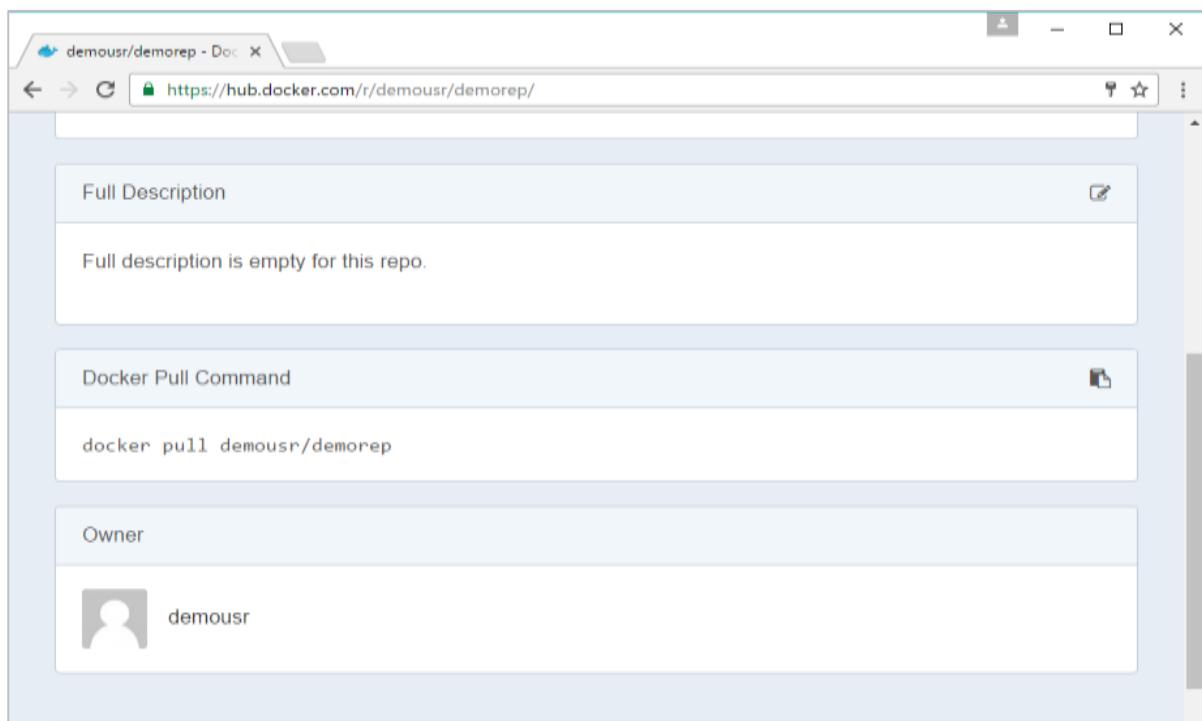
Step 1: Log into Docker Hub and create your repository. This is the repository where your image will be stored. Go to <https://hub.docker.com/> and log in with your credentials.



Step 2: Click the button "Create Repository" on the above screen and create a repository with the name **demorep**. Make sure that the visibility of the repository is public.



Once the repository is created, make a note of the **pull** command which is attached to the repository.



The **pull** command which will be used in our repository is as follows:

```
docker pull demousr/demorep
```

Step 3: Now go back to the Docker Host. Here we need to tag our **myimage** to the new repository created in Docker Hub. We can do this via the Docker **tag command**.

We will learn more about this **tag** command later in this chapter.

Step 4: Issue the Docker login command to login into the Docker Hub repository from the command prompt. The Docker login command will prompt you for the username and password to the Docker Hub repository.

```
demo@ubuntudemo:~$ sudo docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't
have a Docker ID, head over to https://hub.docker.com to create one.
Username: demousr
Password:
Login Succeeded
demo@ubuntudemo:~$
```

Step 5: Once the image has been tagged, it's now time to push the image to the Docker Hub repository. We can do this via the Docker **push** command. We will learn more about this command later in this chapter.

docker tag

This method allows one to tag an image to the relevant repository.

Syntax

```
docker tag imageID Repositoryname
```

Options

- **imageID** – This is the ImageID which needs to be tagged to the repository.
- **Repositoryname** – This is the repository name to which the ImageID needs to be tagged to.

Return Value

None

Example

```
sudo docker tag ab0c1d3744dd demousr/demorep:1.0
```

Output

A sample output of the above example is given below.

```
demo@ubuntudemo:~$ sudo docker images
REPOSITORY          TAG      IMAGE ID      CREATED
SIZE
myimage              0.1      ab0c1d3744dd   6 minutes ago
225.3 MB
centos               latest    67591570dd29   2 days ago
191.8 MB
ubuntu               latest    104bec311bcd   2 days ago
129 MB
demo@ubuntudemo:~$ sudo docker tag ab0c1d3744dd demousr/demorep:1.0
demo@ubuntudemo:~$
```

docker push

This method allows one to push images to the Docker Hub.

Syntax

```
docker push Repositoryname
```

Options

- **Repositoryname** – This is the repository name which needs to be pushed to the Docker Hub.

Return Value

The long ID of the repository pushed to Docker Hub.

Example

```
sudo docker push demousr/demorep:1.0
```

Output

```
demo@ubuntudemo:~$ sudo docker push demousr/demorep:1.0
The push refers to a repository [docker.io/demousr/demorep]
2fa3ddba4e69: Layer already exists
ef84b80e23cc: Layer already exists
5972ebe5b524: Layer already exists
3d515508d4eb: Layer already exists
bbe6cef52379: Layer already exists
87f743c24123: Pushed
32d75bc97c41: Layer already exists
1.0: digest: sha256:1bcdae3a9270a95798f02cd287b91956c5a6cf9fae08d82eb3d11f3a22d4
8d42 size: 1781
demo@ubuntudemo:~$
```

If you go back to the Docker Hub page and go to your repository, you will see the tag name in the repository.

The screenshot shows the Docker Hub interface for the repository 'demousr/domorep'. At the top, there's a navigation bar with links for 'Dashboard', 'Explore', and 'Organizations'. A search bar and a 'Create' button are also present. The main area is titled 'PUBLIC REPOSITORY' and shows the repository name 'demousr/domorep' with a star icon. Below this, it says 'Last pushed: a few seconds ago'. There are tabs for 'Repo Info', 'Tags', 'Collaborators', 'Webhooks', and 'Settings', with 'Tags' currently selected. The 'Tags' section displays a table with three columns: 'Tag Name', 'Compressed Size', and 'Last Updated'. One row is visible, showing '1.0', '103 MB', and 'a few seconds ago'. A small trash can icon is next to the '1.0' entry.

Now let's try to pull the repository we uploaded onto our Docker host. Let's first delete the images, **myimage:0.1** and **demousr/domorep:1.0**, from the local Docker host. Let's use the Docker **pull command** to pull the repository from the Docker Hub.

```
demo@ubuntudemo:~$ sudo docker images
REPOSITORY          TAG      IMAGE ID      CREATED
SIZE
centos              latest   67591570dd29    2 days ago
191.8 MB
ubuntu               latest   104bec311bcd    2 days ago
129 MB
demo@ubuntudemo:~$ sudo docker pull demousr/domorep:1.0
1.0: Pulling from demousr/domorep
b3e1c725a85f: Already exists
4daad8bdde31: Already exists
63fe8c0068a8: Already exists
4a70713c436f: Already exists
bd842a2105a8: Already exists
9b0dd3bf5478: Pull complete
6d3c35e0a8a2: Pull complete
Digest: sha256:1bcdae3a9270a95798f02cd287b91956c5a6cf9fae08d82eb3d11f3a22d48d42
Status: Downloaded newer image for demousr/domorep:1.0
demo@ubuntudemo:~$
```

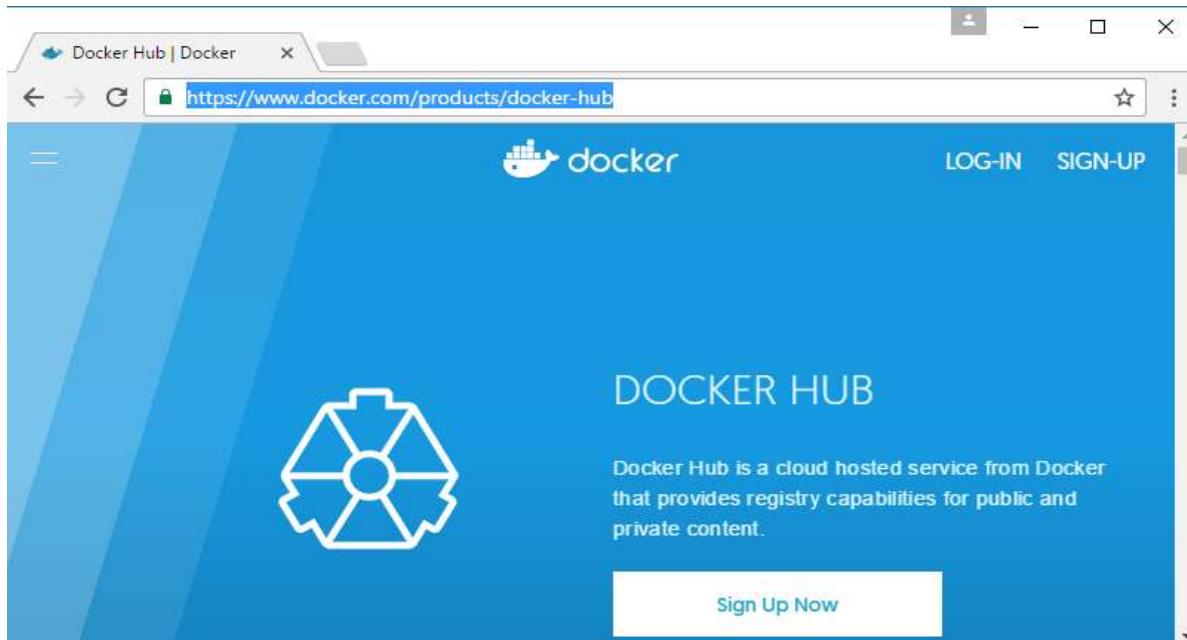
From the above screenshot, you can see that the Docker **pull** command has taken our new repository from the Docker Hub and placed it on our machine.

15. Docker – Managing Ports

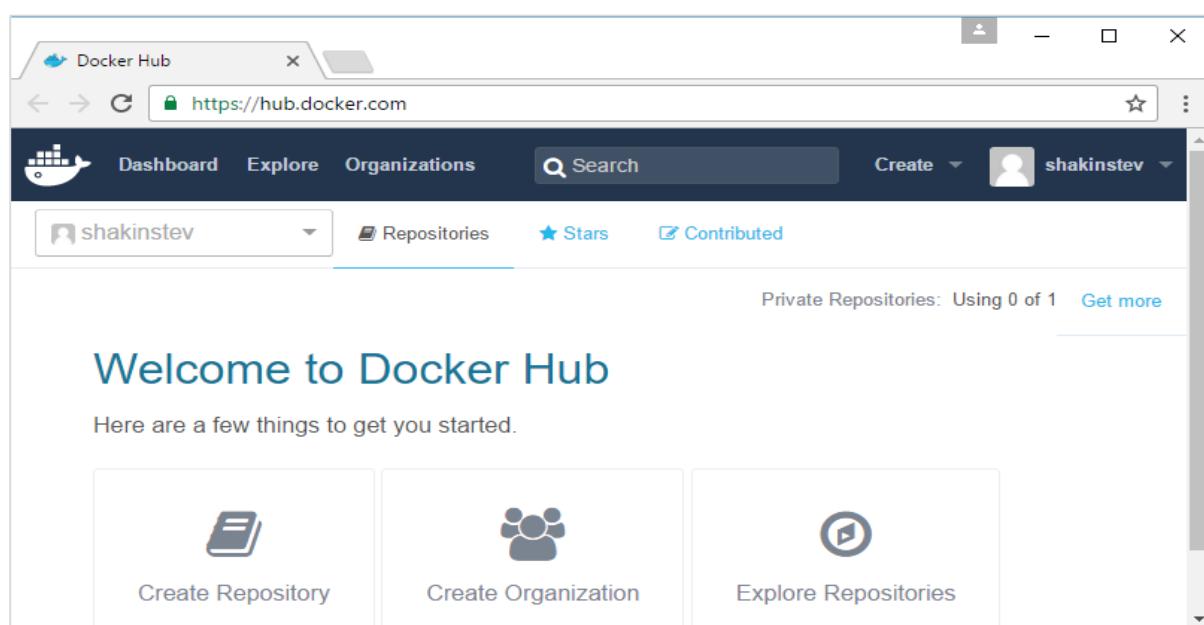
In Docker, the containers themselves can have applications running on ports. When you run a container, if you want to access the application in the container via a port number, you need to map the port number of the container to the port number of the Docker host. Let's look at an example of how this can be achieved.

In our example, we are going to download the Jenkins container from Docker Hub. We are then going to map the Jenkins port number to the port number on the Docker host.

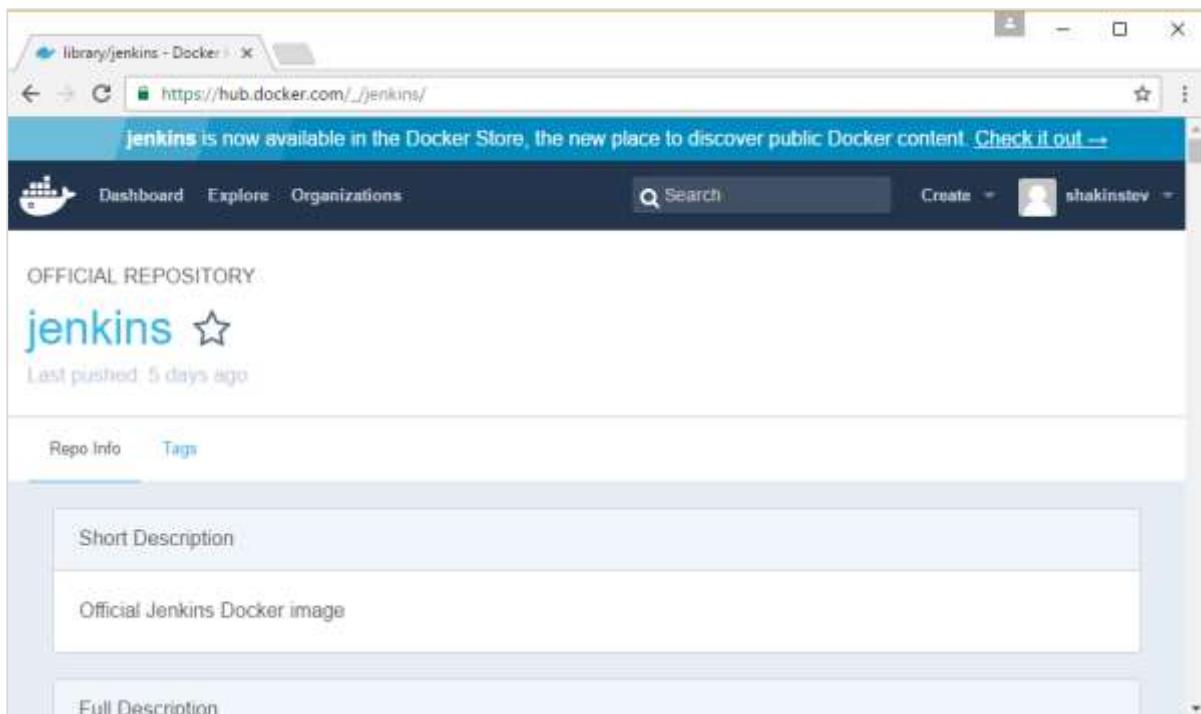
Step 1: First, you need to do a simple sign-up on Docker Hub.



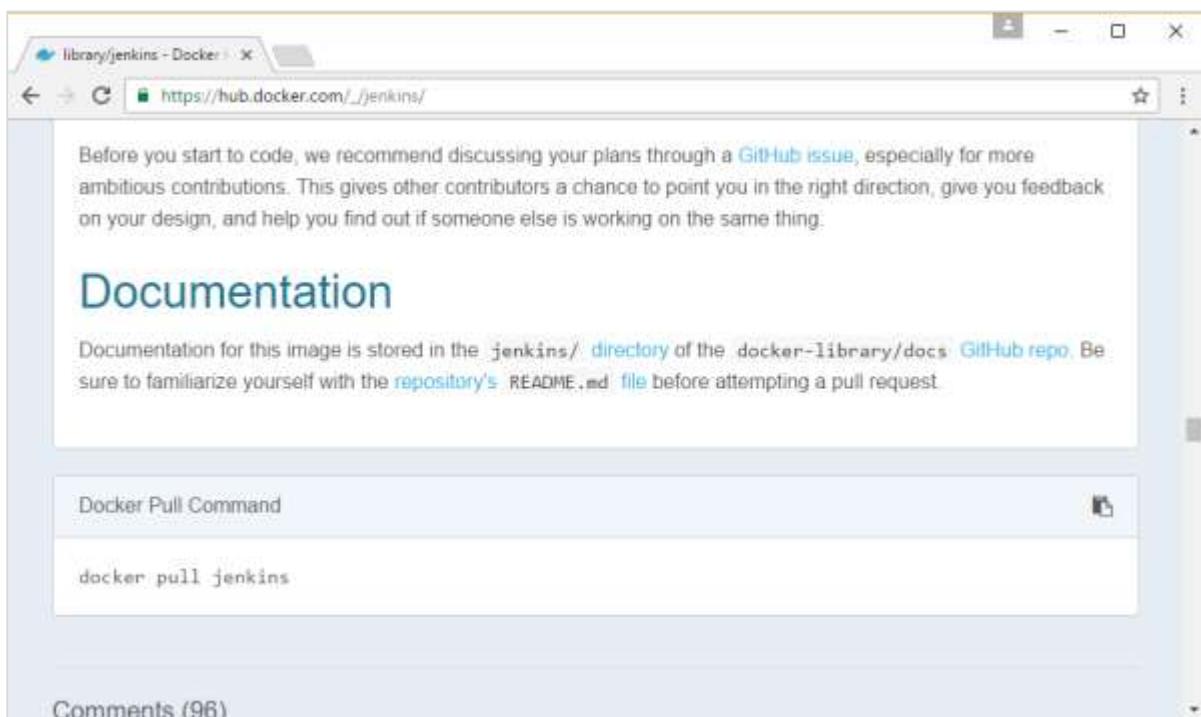
Step 2: Once you have signed up, you will be logged into Docker Hub.



Step 3: Next, let's browse and find the Jenkins image.



Step 4: If you scroll down on the same page, you can see the Docker **pull** command. This will be used to download the Jenkins Image onto the local Ubuntu server.



Step 5: Now go to the Ubuntu server and run the command:

```
sudo docker pull jenkins
```

```
a079defbaeff: Pull complete
66181a89effa: Pull complete
f4d8f7d94b9c: Pull complete
98e5c3e08215: Pull complete
992fde8f3336: Pull complete
65b58e072756: Pull complete
0b0b6d6525a1: Pull complete
4e7171e4505a: Pull complete
469745638476: Pull complete
49d5aaafff78: Pull complete
c01281524fd6: Pull complete
00a759703a0b: Pull complete
da411a858795: Pull complete
7b8a0b4fd7d0: Pull complete
cbd9e145ea6b: Pull complete
700f8f527cd7: Pull complete
88d27231965c: Pull complete
a067af206313: Pull complete
211049e028a4: Pull complete
7249723069d8: Pull complete
6465c437f020: Pull complete
954c67861e66: Pull complete
6a14c8afbb3a: Pull complete
ec070f7e511e: Pull complete
983246da862f: Pull complete
998d1854867e: Pull complete
Digest: sha256:878e055f96c90af9281fd859f7c69ac289e0178594ff36bbb85e53b789691bec
Status: Downloaded newer image for jenkins:latest
demo@ubuntuserver:~$
```

Step 6: To understand what ports are exposed by the container, you should use the Docker **inspect** command to inspect the image.

Let's now learn more about this **inspect** command.

docker inspect

This method allows one to return low-level information on the container or image.

Syntax

```
docker inspect Container/Image
```

Options

- Container/Image – The container or image to inspect.

Return Value

The low-level information of the image or container in JSON format.

Example

```
sudo docker inspect jenkins
```

Output

```
[{"Id": "sha256:ff6f0851ef574309ccd37c29e024f4d2a475d2436c8ebfa1180d45d8ebc9d1f2", "RepoTags": ["jenkins:latest"], "RepoDigests": ["jenkins@sha256:8d39e83e2e97f4f5f1ff6980f9bda7f7b3e7fb9208d94b377d475a7e3054a5b"], "Parent": "", "Comment": "", "Created": "2016-12-01T20:17:24.232532333Z", "Container": "34804931e11a95400d6c40263174f32d498fd3ffff160f6deae807a323365c76", "ContainerConfig": {"Hostname": "6b3797ab1e90", "Domainname": "", "User": "jenkins", "AttachStdin": false, "AttachStdout": false, "AttachStderr": false, "ExposedPorts": {"50000/tcp": {}, "8080/tcp": {}}, "Tty": false, "OpenStdin": false}, --More-- (14%)
```

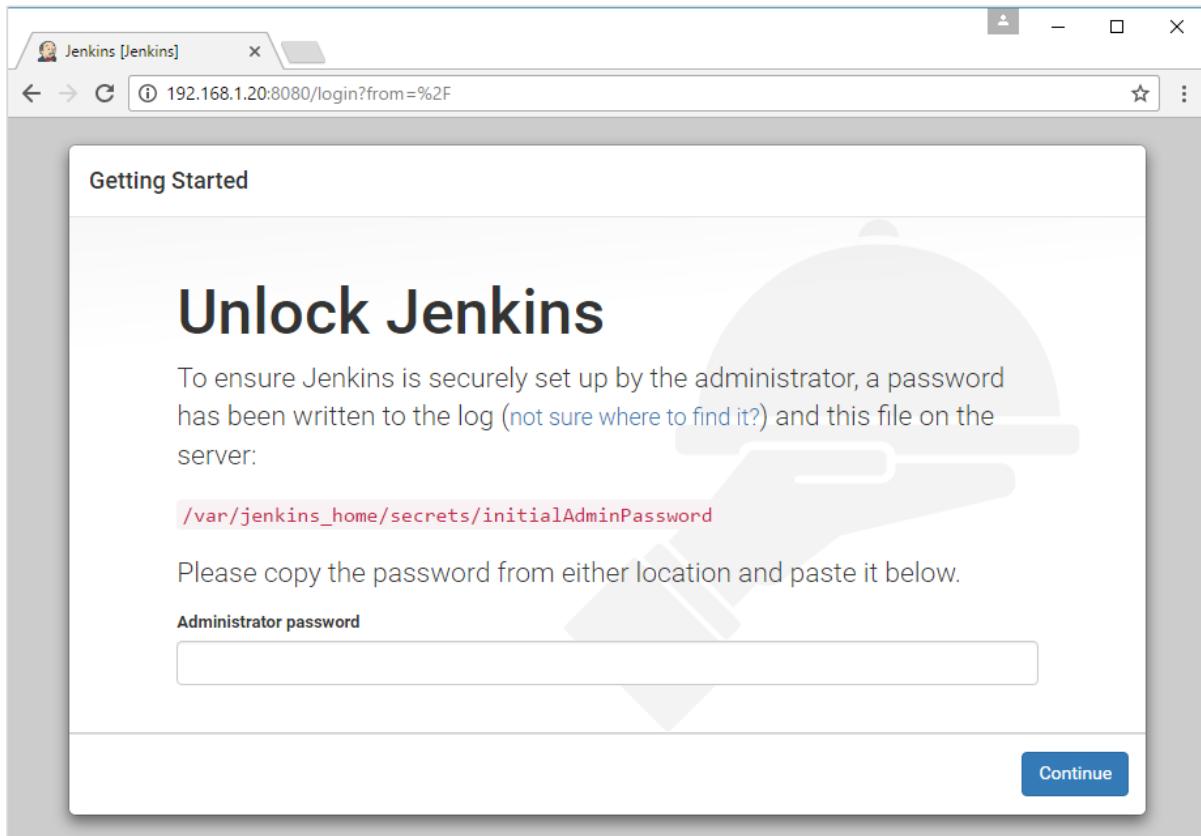
The output of the **inspect** command gives a JSON output. If we observe the output, we can see that there is a section of "ExposedPorts" and see that there are two ports mentioned. One is the **data port** of 8080 and the other is the **control port** of 50000.

To run Jenkins and map the ports, you need to change the Docker **run** command and add the 'p' option which specifies the port mapping. So, you need to run the following command:

```
sudo docker run -p 8080:8080 -p 50000:50000 jenkins
```

The left-hand side of the port number mapping is the Docker host port to map to and the right-hand side is the Docker container port number.

When you open the browser and navigate to the Docker host on port 8080, you will see Jenkins up and running.



16. Docker – Private Registries

You might have the need to have your own private repositories. You may not want to host the repositories on Docker Hub. For this, there is a repository container itself from Docker. Let's see how we can download and use the container for registry.

Step 1: Use the Docker **run** command to download the private registry. This can be done using the following command:

```
sudo docker run -d -p 5000:5000 --name registry registry:2
```

The following points need to be noted about the above command:

- **Registry** is the container managed by Docker which can be used to host private repositories.
- The port number exposed by the container is 5000. Hence with the **-p command**, we are mapping the same port number to the 5000 port number on our localhost.
- We are just tagging the registry container as "2", to differentiate it on the Docker host.
- The **-d** option is used to run the container in detached mode. This is so that the container can run in the background.

```
demo@ubuntudemo:~$ sudo docker run -d -p 5000:5000 --name registry registry:2
Unable to find image 'registry:2' locally
2: Pulling from library/registry

3690ec4760f9: Already exists
930045f1e8fb: Already exists
feeaa90cbdbc: Already exists
61f85310d350: Already exists
b6082c239858: Already exists
Digest: sha256:1152291c7f93a4ea2ddc95e46d142c31e743b6dd70e194af9e6ebe530f782c17
Status: Downloaded newer image for registry:2
bf47ab9f46963746d8686994339a6a7fc33b4ac889c8f92ffe392cdfa8414068
demo@ubuntudemo:~$
```

Step 2: Let's do a **docker ps** to see that the registry container is indeed running.

```
demo@ubuntudemo:~$ sudo docker run -d -p 5000:5000 --name registry registry:2
Unable to find image 'registry:2' locally
2: Pulling from library/registry

3690ec4760f9: Already exists
930045f1e8fb: Already exists
feeaa90cbdbc: Already exists
61f85310d350: Already exists
b6082c239858: Already exists
Digest: sha256:1152291c7f93a4ea2ddc95e46d142c31e743b6dd70e194af9e6ebe530f782c17
Status: Downloaded newer image for registry:2
bf47ab9f46963746d8686994339a6a7fc33b4ac889c8f92ffe392cdfa8414068
demo@ubuntudemo:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES
          STATUS        PORTS              NAMES
bf47ab9f4696        registry:2         "/entrypoint.sh /etc/"   About a minute ago   Up About a minute   0.0.0.0:5000->5000/tcp   registry
demo@ubuntudemo:~$ _
```

We have now confirmed that the registry container is indeed running.

Step 3: Now let's tag one of our existing images so that we can push it to our local repository. In our example, since we have the **centos** image available locally, we are going to tag it to our private repository and add a tag name of **centos**.

```
sudo docker tag 67591570dd29 localhost:5000/centos
```

The following points need to be noted about the above command:

- **67591570dd29** refers to the Image ID for the **centos** image.
- **localhost:5000** is the location of our private repository.
- We are tagging the repository name as **centos** in our private repository.

```
demo@ubuntudemo:~$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED           SIZE
demousr/demorep    1.0                ab0c1d3744dd      24 hours ago   225.3 MB
centos              latest              67591570dd29      3 days ago     191.8 MB
jenkins              latest              ff6f0851ef57      2 weeks ago    714.1 MB
registry              2                  c9bd19d022f6      8 weeks ago    33.3 MB
demo@ubuntudemo:~$ sudo docker tag 67591570dd29 localhost:5000/centos
demo@ubuntudemo:~$ _
```

Step 4: Now let's use the Docker **push** command to push the repository to our private repository.

```
sudo docker push localhost:5000/centos
```

Here, we are pushing the **centos** image to the private repository hosted at **localhost:5000**.

```

demo@ubuntudemo:~$ sudo docker images
REPOSITORY          TAG      IMAGE ID      CREATED
SIZE
demousr/demorep    1.0      ab0c1d3744dd  24 hours ago
225.3 MB
centos             latest   67591570dd29  3 days ago
191.8 MB
jenkins            latest   ff6f0851ef57  2 weeks ago
714.1 MB
registry           2        c9bd19d022f6  8 weeks ago
33.3 MB

demo@ubuntudemo:~$ sudo docker tag 67591570dd29 localhost:5000/centos
demo@ubuntudemo:~$ sudo docker push localhost:5000/centos
The push refers to a repository [localhost:5000/centos]
34e7b85d83e4: Pushed
latest: digest: sha256:c577af3197aacedf79c5a204cd7f493c8e07ff bce7f88f7600bf19c68
8c38799 size: 529
demo@ubuntudemo:~$ _

```

Step 5: Now let's delete the local images we have for **centos** using the **docker rmi** commands. We can then download the required **centos** image from our private repository.

```

sudo docker rmi centos:latest
sudo docker rmi 67591570dd29

```

```

demo@ubuntudemo:~$ sudo docker images
REPOSITORY          TAG      IMAGE ID      CREATED
SIZE
demousr/demorep    1.0      ab0c1d3744dd  24 hours ago
225.3 MB
centos             latest   67591570dd29  3 days ago
191.8 MB
localhost:5000/centos  latest   67591570dd29  3 days ago
191.8 MB
jenkins            latest   ff6f0851ef57  2 weeks ago
714.1 MB
registry           2        c9bd19d022f6  8 weeks ago
33.3 MB

demo@ubuntudemo:~$ sudo docker rmi centos:latest
Untagged: centos:latest
Untagged: centos@sha256:c577af3197aacedf79c5a204cd7f493c8e07ff bce7f88f7600bf19c68c38799
demo@ubuntudemo:~$ sudo docker rmi 67591570dd29
Untagged: localhost:5000/centos:latest
Untagged: localhost:5000/centos@sha256:c577af3197aacedf79c5a204cd7f493c8e07ff bce7f88f7600bf19c68c38799
Deleted: sha256:67591570dd29de0e124ee89d50458b098dbd83b12d73e5fdaf8b4dcdb4ea50f8
Deleted: sha256:34e7b85d83e48a22bd5dfa2b6b9ee9565b7ef672f09b3d2409c61635f9bca4db
demo@ubuntudemo:~$ _

```

Step 6: Now that we don't have any **centos** images on our local machine, we can now use the following Docker **pull** command to pull the **centos** image from our private repository.

```

sudo docker pull localhost:5000/centos

```

Here, we are pulling the **centos** image to the private repository hosted at **localhost:5000**.

```
demo@ubuntudemo:~$ sudo docker pull localhost:5000/centos
Using default tag: latest
latest: Pulling from centos

45a2e645736c: Pull complete
Digest: sha256:c577af3197aacedf79c5a204cd7f493c8e07ffbcce7f88f7600bf19c688c38799
Status: Downloaded newer image for localhost:5000/centos:latest
demo@ubuntudemo:~$ sudo docker images
REPOSITORY          TAG      IMAGE ID      CREATED     SIZE
demouser/demorep    1.0      ab0c1d3744dd  24 hours ago  225.3 MB
localhost:5000/centos  latest   67591570dd29  3 days ago   191.8 MB
jenkins             latest   ff6f0851ef57  2 weeks ago   714.1 MB
registry             2       c9bd19d022f6  8 weeks ago   33.3 MB
demo@ubuntudemo:~$
```

If you now see the images on your system, you will see the **centos** image as well.

17. Docker – Building a Web Server Docker File

We have already learnt how to use Docker File to build our own custom images. Now let's see how we can build a web server image which can be used to build containers.

In our example, we are going to use the Apache Web Server on Ubuntu to build our image. Let's follow the steps given below, to build our web server Docker file.

Step 1: The first step is to build our Docker File. Let's use **vim** and create a Docker File with the following information.

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y apache2
RUN apt-get install -y apache2-utils
RUN apt-get clean
EXPOSE 80
CMD ["apache2ctl", "-D", "FOREGROUND"]
```

The following points need to be noted about the above statements:

- We are first creating our image to be from the Ubuntu base image.
- Next, we are going to use the RUN command to update all the packages on the Ubuntu system.
- Next, we use the RUN command to install apache2 on our image.
- Next, we use the RUN command to install the necessary utility apache2 packages on our image.
- Next, we use the RUN command to clean any unnecessary files from the system.
- The EXPOSE command is used to expose port 80 of Apache in the container to the Docker host.
- Finally, the CMD command is used to run apache2 in the background.

```
FROM ubuntu
MAINTAINER demousr@gmail.com
RUN apt-get update
RUN apt-get install -y apache2
RUN apt-get install -y apache2-utils
RUN apt-get clean
EXPOSE 80
CMD ["apache2ctl", "-D", "FOREGROUND"]
```

Now that the file details have been entered, just save the file.

Step 2: Run the Docker **build** command to build the Docker file. It can be done using the following command:

```
sudo docker build -t="mywebserver" .
```

We are tagging our image as **mywebserver**. Once the image is built, you will get a successful message that the file has been built.

```
Processing triggers for libc-bin (2.23-0ubuntu5) ...
Processing triggers for systemd (229-4ubuntu12) ...
Processing triggers for sgml-base (1.26+nmu4ubuntu1) ...
---> 3deecdb58eea
Removing intermediate container a34fbe45c6f0
Step 5 : RUN apt-get install -y apache2-utils
--> Running in 3924b32e72c0
Reading package lists...
Building dependency tree...
Reading state information...
apache2-utils is already the newest version (2.4.18-2ubuntu3.1).
apache2-utils set to manually installed.
0 upgraded, 0 newly installed, 0 to remove and 4 not upgraded.
--> 9ddc59d1764b
Removing intermediate container 3924b32e72c0
Step 6 : RUN apt-get clean
--> Running in cb73b67c8109
--> 4a13c4c36e57
Removing intermediate container cb73b67c8109
Step 7 : EXPOSE 80
--> Running in 85245722be33
--> e4d2eb0fc674
Removing intermediate container 85245722be33
Step 8 : CMD apache2ctl -D FOREGROUND
--> Running in 49d3437f799f
--> 5ca8134b8d87
Removing intermediate container 49d3437f799f
Successfully built 5ca8134b8d87
demo@ubuntudemo:~$
```

Step 3: Now that the web server file has been built, it's now time to create a container from the image. We can do this with the Docker **run** command.

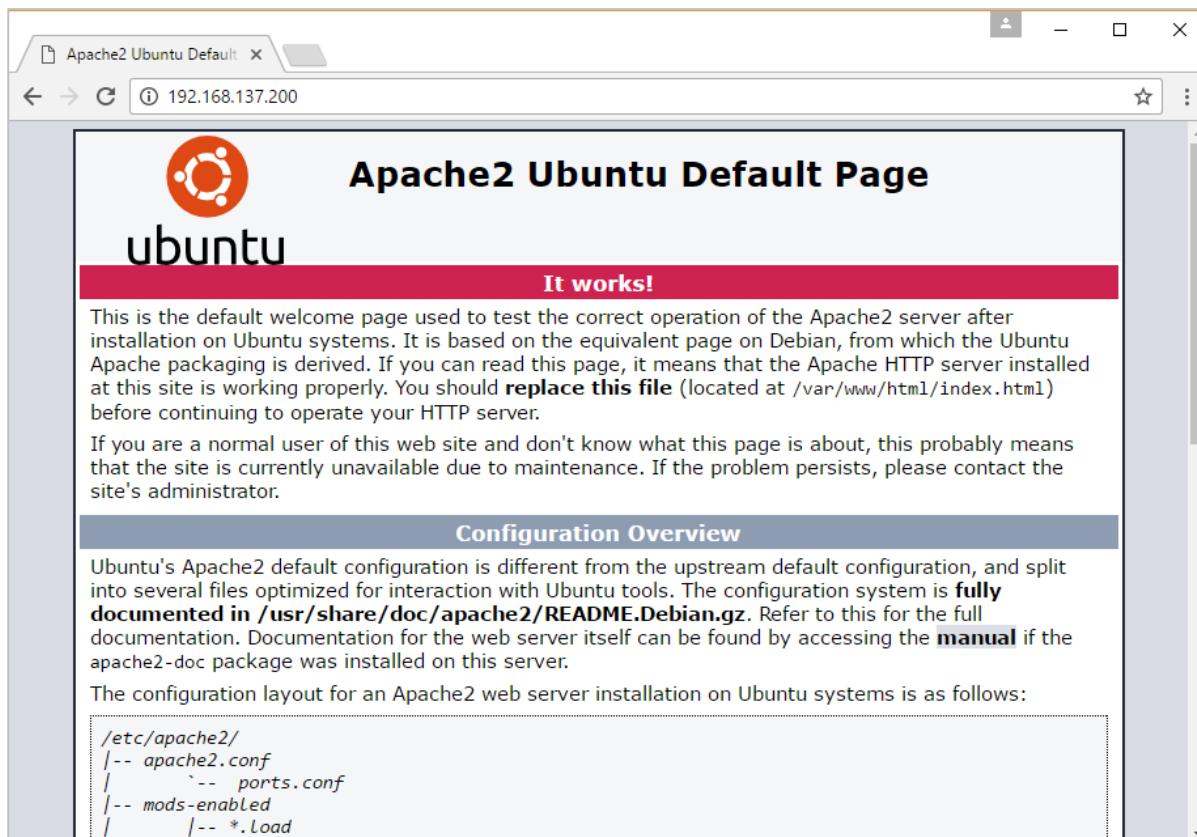
```
sudo docker run -d -p 80:80 mywebserver
```

```
demo@ubuntudemo:~$ sudo docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
mywebserver        latest   5ca8134b8d87  4 minutes ago  267.6 MB
demousr/demorep    1.0     ab0c1d3744dd  26 hours ago  225.3 MB
ubuntu              latest   104bec311bcd  3 days ago   129 MB
jenkins             latest   ff6f0851ef57  2 weeks ago   714.1 MB
registry            2        c9bd19d022f6  8 weeks ago   33.3 MB
demo@ubuntudemo:~$ sudo docker run -d -p 80:80 mywebserver
42c70f5e90a2915d1954af2207de75657231c906feb9366f15a4e5c128c0675a
demo@ubuntudemo:~$ _
```

The following points need to be noted about the above command:

- The port number exposed by the container is 80. Hence with the **-p** command, we are mapping the same port number to the 80 port number on our localhost.
- The **-d** option is used to run the container in detached mode. This is so that the container can run in the background.

If you go to port 80 of the Docker host in your web browser, you will now see that Apache is up and running.



18. Docker – Instruction Commands

Docker has a host of instruction commands. These are commands that are put in the Docker File. Let's look at the ones which are available.

CMD Instruction

This command is used to execute a command at runtime when the container is executed.

Syntax

```
CMD command param1
```

Options

- **command** – This is the command to run when the container is launched.
- **param1** – This is the parameter entered to the command.

Return Value

The command will execute accordingly.

Example

In our example, we will enter a simple **Hello World** echo in our Docker File and create an image and launch a container from it.

Step 1: Build the Docker File with the following commands:

```
FROM ubuntu
MAINTAINER demousr@gmail.com
CMD ["echo" , "hello world"]
```

Here, the CMD is just used to print **hello world**.

```
FROM ubuntu
MAINTAINER demousr@gmail.com
CMD ["echo", "hello world"]
~
```

Step 2: Build the image using the Docker **build** command.

```
demo@ubuntudemo:~$ sudo docker build -t="mynewdemo" .
Sending build context to Docker daemon 21.5 kB
Step 1 : FROM ubuntu
--> 104bec311bcd
Step 2 : MAINTAINER demousr@gmail.com
--> Using cache
--> 429c19673474
Step 3 : CMD echo hello world
--> Running in 6589f66cffd4
--> 90ab8626a009
Removing intermediate container 6589f66cffd4
Successfully built 90ab8626a009
demo@ubuntudemo:~$ _
```

Step 3: Run a container from the image.

```
demo@ubuntudemo:~$ sudo docker run mynewdemo
hello world
demo@ubuntudemo:~$ _
```

ENTRYPOINT

This command can also be used to execute commands at runtime for the container. But we can be more flexible with the ENTRYPOINT command.

Syntax

```
ENTRYPOINT command param1
```

Options

- **command** – This is the command to run when the container is launched.
- **param1** – This is the parameter entered into the command.

Return Value

The command will execute accordingly.

Example

Let's take a look at an example to understand more about ENTRYPOINT. In our example, we will enter a simple **echo** command in our Docker File and create an image and launch a container from it.

Step 1: Build the Docker File with the following commands:

```
FROM ubuntu
MAINTAINER demousr@gmail.com
ENTRYPOINT ["echo"]
```

```
FROM ubuntu
MAINTAINER demousr@gmail.com
ENTRYPOINT ["echo"]
```

Step 2: Build the image using the Docker **build** command.

```
demo@ubuntudemo:~$ sudo docker build -t="entrydemo" .
Sending build context to Docker daemon 22.53 kB
Step 1 : FROM ubuntu
--> 104bec311bcd
Step 2 : MAINTAINER demousr@gmail.com
--> Using cache
--> 429c19673474
Step 3 : ENTRYPOINT echo
--> Running in 4a06da685d12
--> c26b8ef5a8c9
Removing intermediate container 4a06da685d12
Successfully built c26b8ef5a8c9
demo@ubuntudemo:~$
```

Step 3: Run a container from the image.

```
demo@ubuntudemo:~$ sudo docker build -t="entrydemo" .
Sending build context to Docker daemon 22.53 kB
Step 1 : FROM ubuntu
--> 104bec311bcd
Step 2 : MAINTAINER demousr@gmail.com
--> Using cache
--> 429c19673474
Step 3 : ENTRYPOINT echo
--> Running in 4a06da685d12
--> c26b8ef5a8c9
Removing intermediate container 4a06da685d12
Successfully built c26b8ef5a8c9
demo@ubuntudemo:~$ sudo docker run entrydemo Hello World
Hello World
demo@ubuntudemo:~$ _
```

ENV

This command is used to set environment variables in the container.

Syntax

```
ENV key value
```

Options

- **Key** – This is the key for the environment variable.
- **value** – This is the value for the environment variable.

Return Value

The command will execute accordingly.

Example

In our example, we will enter a simple **echo** command in our Docker File and create an image and launch a container from it.

Step 1: Build the Docker File with the following commands:

```
FROM ubuntu
MAINTAINER demousr@gmail.com
ENV var1=Tutorial var2=point
```

```
FROM ubuntu
MAINTAINER demousr@gmail.com
ENV var1=Tutorial var2=point_
~
~
```

Step 2: Build the image using the Docker **build** command.

```
demo@ubuntudemo:~$ sudo docker build -t="envdemo" .
Sending build context to Docker daemon 23.04 kB
Step 1 : FROM ubuntu
--> 104bec311bcd
Step 2 : MAINTAINER demousr@gmail.com
--> Using cache
--> 429c19673474
Step 3 : ENV var1 Tutorial var2 point
--> Running in 8bd8eccb5986
--> 1def7e9aa854
Removing intermediate container 8bd8eccb5986
Successfully built 1def7e9aa854
demo@ubuntudemo:~$ _
```

Step 3: Run a container from the image.

```
demo@ubuntudemo:~$ sudo docker build -t="envdemo" .
Sending build context to Docker daemon 23.04 kB
Step 1 : FROM ubuntu
--> 104bec311bcd
Step 2 : MAINTAINER demousr@gmail.com
--> Using cache
--> 429c19673474
Step 3 : ENV var1 Tutorial var2 point
--> Running in 8bd8eccb5986
--> 1def7e9aa854
Removing intermediate container 8bd8eccb5986
Successfully built 1def7e9aa854
demo@ubuntudemo:~$ sudo docker run -it envdemo /bin/bash
root@b48d9e69cc34:/# _
```

Step 4: Finally, execute the **env** command to see the environment variables.

```
demo@ubuntudemo:~$ sudo docker run -it envdemo /bin/bash
root@b48d9e69cc34:/# env
var1=Tutorial
var2=point
HOSTNAME=b48d9e69cc34
TERM=xterm
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33:01:cd
=40;33:01:or=40;31:01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;4
4:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;
31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7
z=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=0
1;31:*.lzo=01;31:*.xz=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz
=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*
.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;3
1:*.cab=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm
=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:
*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=0
1;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*
.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;
35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=
01;35:*.dl=01;35:*.xcf=01;35:*.xdw=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.o
gv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.f lac=00;36:*.m4a=00;36:*.mid=00;36
:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=0
0;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
SHLVL=1
HOME=/root
_=usr/bin/env
```

WORKDIR

This command is used to set the working directory of the container.

Syntax

```
WORKDIR dirname
```

Options

- **dirname** – The new working directory. If the directory does not exist, it will be added.

Return Value

The command will execute accordingly.

Example

In our example, we will enter a simple **echo** command in our Docker File and create an image and launch a container from it.

Step 1: Build the Docker File with the following commands:

```
FROM ubuntu
MAINTAINER demousr@gmail.com
WORKDIR /newtemp
CMD pwd
```

```
FROM ubuntu
MAINTAINER demousr@gmail.com
WORKDIR /newtemp
CMD pwd
```

Step 2: Build the image using the Docker **build** command.

```
demo@ubuntudemo:~$ sudo docker build -t="tempdemo" .
Sending build context to Docker daemon 23.55 kB
Step 1 : FROM ubuntu
--> 104bec311bcd
Step 2 : MAINTAINER demousr@gmail.com
--> Using cache
--> 429c19673474
Step 3 : WORKDIR /newtemp
--> Using cache
--> e09e6378e765
Step 4 : CMD pwd
--> Using cache
--> c7bedf4e3158
Successfully built c7bedf4e3158
demo@ubuntudemo:~$
```

Step 3: Run a container from the image.

```
demo@ubuntudemo:~$ sudo docker build -t="tempdemo" .
Sending build context to Docker daemon 23.55 kB
Step 1 : FROM ubuntu
--> 104bec311bcd
Step 2 : MAINTAINER demousr@gmail.com
--> Using cache
--> 429c19673474
Step 3 : WORKDIR /newtemp
--> Using cache
--> e09e6378e765
Step 4 : CMD pwd
--> Using cache
--> c7bedf4e3158
Successfully built c7bedf4e3158
demo@ubuntudemo:~$ sudo docker run tempdemo
/newtemp
demo@ubuntudemo:~$ _
```

19. Docker – Container Linking

Container Linking allows multiple containers to link with each other. It is a better option than exposing ports. Let's go step by step and learn how it works.

Step 1: Download the Jenkins image, if it is not already present, using the Jenkins **pull** command.

```
demo@ubuntudemo:~$ sudo docker jenkins pull_
```

Step 2: Once the image is available, run the container, but this time, you can specify a name to the container by using the **--name** option. This will be our **source container**.

```
demo@ubuntudemo:~$ sudo docker run --name=jenkinsa -d jenkins_
```

Step 3: Next, it is time to launch the destination container, but this time, we will link it with our source container. For our destination container, we will use the standard Ubuntu image.

```
demo@ubuntudemo:~$ sudo docker run --name=reca --link=jenkinsa:alias-src -it ubu^  
ntu:latest /bin/bash_
```

When you do a **docker ps**, you will see both the containers running.

Step 4: Now, attach to the receiving container.

```
demo@ubuntudemo:~$ sudo docker ps  
[sudo] password for demo:  
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES  
13ea6d68a149        ubuntu:latest       "/bin/bash"  
                   Up 32 minutes          reca  
9f55a4963c44        jenkins             "/bin/tini -- /usr/lo"  
                   Up 33 minutes          8080/tcp, 50000/tcp   jenkinsa  
demo@ubuntudemo:~$ sudo docker attach reca  
root@13ea6d68a149:/# _
```

Then run the **env** command. You will notice new variables for linking with the source container.

```
35:*.rm=01:35:*.rmvb=01:35:*.flc=01:35:*.avi=01:35:*.fli=01:35:*.flv=01:35:*.gl=^
01:35:*.dl=01:35:*.xcf=01:35:*.xwd=01:35:*.yuv=01:35:*.cgm=01:35:*.emf=01:35:*.o
gv=01:35:*.ogx=01:35:*.aac=00:36:*.au=00:36:*.f lac=00:36:*.m4a=00:36:*.mid=00:36
:*.midi=00:36:*.mka=00:36:*.mp3=00:36:*.mpc=00:36:*.ogg=00:36:*.ra=00:36:*.wav=0
0:36:*.oga=00:36:*.opus=00:36:*.spx=00:36:*.xspf=00:36:
ALIAS_SRC_PORT_50000_TCP=tcp://172.17.0.2:50000
ALIAS_SRC_ENV_JENKINS_SLAVE_AGENT_PORT=50000
ALIAS_SRC_PORT_8080_TCP_PORT=8080
ALIAS_SRC_ENV_JAVA_VERSION=8u111
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
ALIAS_SRC_ENV_LANG=C.UTF-8
ALIAS_SRC_ENV_JAVA_DEBIAN_VERSION=8u111-b14-2~bpo8+1
ALIAS_SRC_PORT_50000_TCP_PORT=50000
ALIAS_SRC_PORT_8080_TCP_ADDR=172.17.0.2
ALIAS_SRC_ENV_TINI_VERSION=0.9.0
ALIAS_SRC_PORT_8080_TCP_PROTO=tcp
ALIAS_SRC_ENV_COPY_REFERENCE_FILE_LOG=/var/jenkins_home/copy_reference_file.log
SHLVL=1
HOME=/root
ALIAS_SRC_PORT_50000_TCP_ADDR=172.17.0.2
ALIAS_SRC_ENV_JENKINS_UC=https://updates.jenkins.io
ALIAS_SRC_ENV_TINI_SHA=fa23d1e20732501c3bb8eeeca423c89ac80ed452
ALIAS_SRC_ENV_JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
ALIAS_SRC_ENV_JENKINS_HOME=/var/jenkins_home
ALIAS_SRC_PORT_8080_TCP=tcp://172.17.0.2:8080
_=usr/bin/env
root@13ea6d68a149:#
root@13ea6d68a149:#

```

```
demo@ubuntudemo:~$ sudo docker ps
[sudo] password for demo:
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
13ea6d68a149        ubuntu:latest       "/bin/bash"            32 minutes ago    Up 32 minutes      "reca"
9f55a4963c44        jenkins           "/bin/tini -- /usr/lo"   33 minutes ago    Up 33 minutes      "jenkinsa"
demo@ubuntudemo:~$ _
```

20. Docker – Docker Storage

Storage Drivers

Docker has multiple storage drivers that allow one to work with the underlying storage devices. The following table shows the different storage drivers along with the technology used for the storage drivers.

Technology	Storage Driver
OverlayFS	overlay or overlay2
AUFS	aufs
Btrfs	brtfs
Device Manager	devicemanager
VFS	vfs
ZFS	zfs

Let us now discuss some of the instances in which you would use the various storage drivers:

AUFS

- This is a stable driver; can be used for production-ready applications.
- It has good memory usage and is good for ensuring a smooth Docker experience for containers.
- There is a high-write activity associated with this driver which should be considered.
- It's good for systems which are of Platform as a service type work.

Devicemapper

- This is a stable driver; ensures a smooth Docker experience.
- This driver is good for testing applications in the lab.
- This driver is in line with the main Linux kernel functionality.

Btrfs

- This driver is in line with the main Linux kernel functionality.
- There is a high-write activity associated with this driver which should be considered.
- This driver is good for instances where you maintain multiple build pools.

Ovelay

- This is a stable driver and it is in line with the main Linux kernel functionality.
- It has a good memory usage.
- This driver is good for testing applications in the lab.

ZFS

- This is a stable driver and it is good for testing applications in the lab.
- It's good for systems which are of Platform-as-a-Service type work.

To see the storage driver being used, issue the **docker info** command.

Syntax

```
docker info
```

Options

None

Return Value

The command will provide all relative information on the Docker component installed on the Docker Host.

Example

```
sudo docker info
```

Output

The following output shows that the main driver used is the **aufs** driver and that the root directory is stored in **/var/lib/docker/aufs**.

```
Root Dir: /var/lib/docker/aufs
Backing Filesystem: extfs
Dirs: 75
Dirperm1 Supported: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
  Volume: local
  Network: bridge host overlay null
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Security Options: apparmor
Kernel Version: 4.2.0-27-generic
Operating System: Ubuntu 14.04.4 LTS
OSType: linux
Architecture: x86_64
CPUs: 1
Total Memory: 993.1 MiB
Name: ubuntudemo
ID: ECDA:IFR3:2CQJ:FNXL:APJR:BT6Y:JJ75:FUE6:DNP5:PD7B:AOAD:YUB4
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Username: demouser
Registry: https://index.docker.io/v1/
WARNING: No swap limit support
Insecure Registries:
  127.0.0.0/8
```

Data Volumes

In Docker, you have a separate volume that can be shared across containers. These are known as **data volumes**. Some of the features of data volume are:

- They are initialized when the container is created.
 - They can be shared and also reused amongst many containers.
 - Any changes to the volume itself can be made directly.
 - They exist even after the container is deleted.

Let's look at our Jenkins container. Let's do a **docker inspect** to see the details of this image. We can issue the following command to write the output of the **docker inspect** command to a text file and then view the file accordingly.

```
sudo docker inspect Jenkins > tmp.txt
```

When you view the text file using the **more command**, you will see an entry as **JENKINS_HOME=/var/Jenkins_home**.

This is the mapping that is done within the container via the Jenkins image.

```
"JAVA_DEBIAN_VERSION=8u111-b14-2~bpo8+1",
"CA_CERTIFICATES_JAVA_VERSION=20140324",
"JENKINS_HOME=/var/jenkins_home",
"JENKINS_SLAVE_AGENT_PORT=50000",
"TINI_VERSION=0.9.0",
"TINI_SHA=fab23d1e20732501c3bb8eeeca423c89ac80ed452",
"JENKINS_VERSION=2.19.4",
"JENKINS_UC=https://updates.jenkins.io",
"COPY_REFERENCE_FILE_LOG=/var/jenkins_home/copy_reference_file.log"
],
"Cmd": [
"/bin/sh",
"-c",
"#(nop) COPY file:2a6a3e16202b8ddab5edef50f712c16fe8f6980f5aea8
0c8c76b5db4f903913 in /usr/local/bin/install-plugins.sh "
],
"ArgsEscaped": true,
"Image": "sha256:86cb73edfb8b3a4681961047f1ca654c81586c7d8fdeaf7904ec
cbcba09ae4d720",
"Volumes": {
"/var/jenkins_home": {}
},
"WorkingDir": "",
"Entrypoint": [
"/bin/tini",
"--",
"/usr/local/bin/jenkins.sh"
--More-- (37%)

```

Now suppose you wanted to map the volume in the container to a local volume, then you need to specify the **-v** option when launching the container. An example is shown below:

```
sudo docker run -d -v /home/demo:/var/jenkins_home -p 8080:8080 -p 50000:50000
jenkins
```

The **-v** option is used to map the volume in the container which is **/var/jenkins_home** to a location on our Docker Host which is **/home/demo**.

```
demo@ubuntudemo:~$ sudo docker run -d -v /home/demo:/var/jenkins_home -p 8080:80
80 -p 50000:50000 jenkins
```

Now if you go to the **/home/demo** location on your Docker Host after launching your container, you will see all the container files present there.

```
demo@ubuntudemo:~$ pwd
/home/demo
demo@ubuntudemo:~$ ls
config.xml           jobs
copy_reference_file.log logs
Dockerfile            nodeMonitors.xml
hudson.model.UpdateCenter.xml   nodes
identity.key.enc      plugins
init.groovy.d          secret.key
jenkins.install.UpgradeWizard.state secret.key.not-so-secret
demo@ubuntudemo:~$
```

Changing the Storage Driver for a Container

If you wanted to change to the storage driver used for a container, you can do so when launching the container. This can be done by using the **-volume-driver** parameter when using the **docker run** command. An example is given below:

```
sudo docker run -d --volume-driver=flocker -v /home/demo:/var/jenkins_home -p 8080:8080 -p 50000:50000 jenkins
```

The **-volume-driver** option is used to specify another storage driver for the container.

```
demo@ubuntudemo:~$ sudo docker run -d --volume-driver=flocker -v /home/demo:/var/jenkins_home -p 8080:8080 -p 50000:50000 jenkins
9bffb1bfebee3fd15bc58ebade534bfbf18bf996d0052af7190aef5fdc37180
demo@ubuntudemo:~$
```

To confirm that the driver has been changed, first let's use the **docker ps** command to see the running containers and get the container ID. So, issue the following command first:

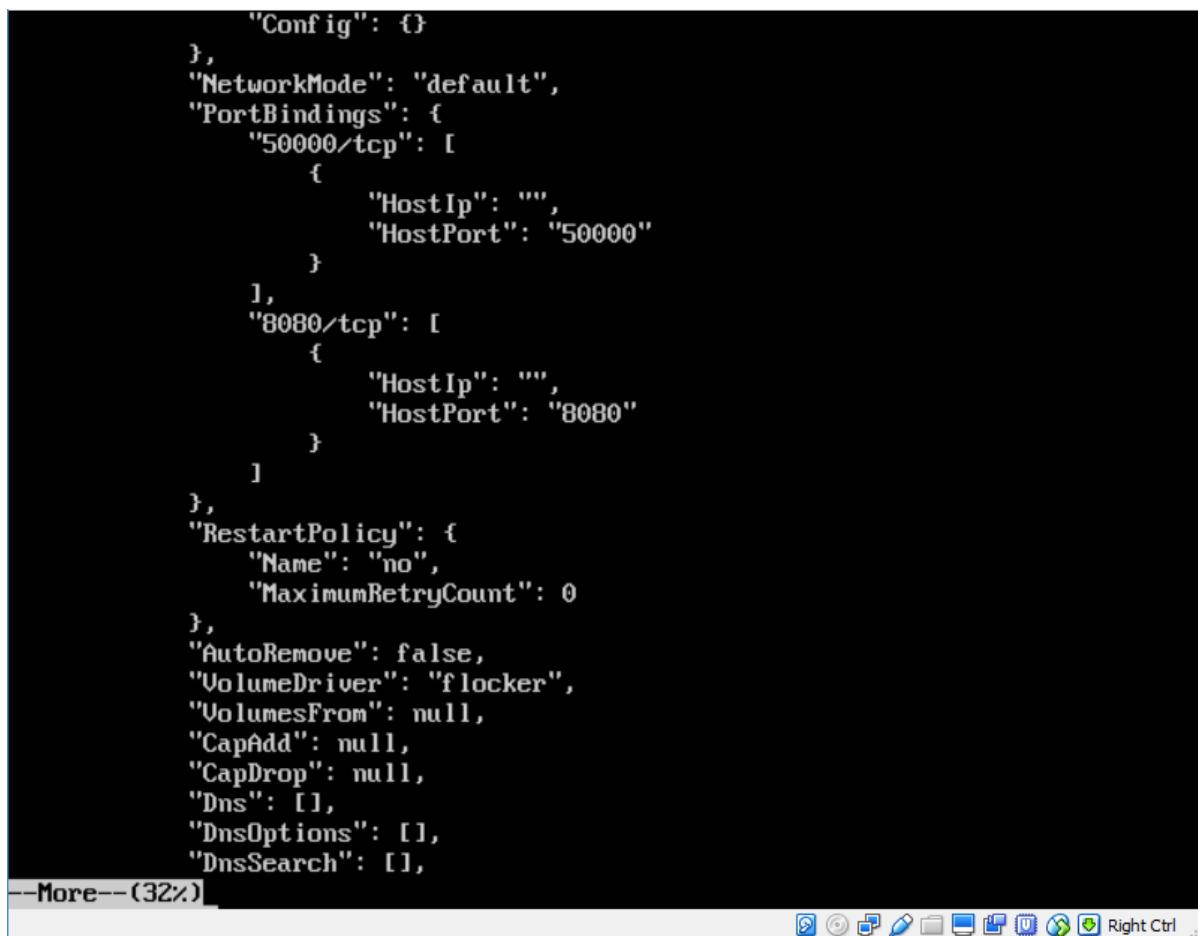
```
sudo docker ps
```

Then issue a **docker inspect** against the container and put the output in a text file using the command.

```
sudo docker inspect 9bffb1bfebee > temp.txt
```

```
demo@ubuntudemo:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             NAME
S
9bffb1bfebee      jenkins            "/bin/tini -- /usr/lo"   2 minutes ago     gloomy_wing
Up 2 minutes        0.0.0.0:8080->8080/tcp, 0.0.0.0:50000->50000/tcp
demo@ubuntudemo:~$ sudo docker inspect 9bffb1bfebee > temp.txt_
```

If you browse through the text file and go to the line which says **VolumeDriver**, you will see that the driver name has been changed.



```

    "Config": {},
    "NetworkMode": "default",
    "PortBindings": {
        "50000/tcp": [
            {
                "HostIp": "",
                "HostPort": "50000"
            }
        ],
        "8080/tcp": [
            {
                "HostIp": "",
                "HostPort": "8080"
            }
        ]
    },
    "RestartPolicy": {
        "Name": "no",
        "MaximumRetryCount": 0
    },
    "AutoRemove": false,
    "VolumeDriver": "flocker",
    "VolumesFrom": null,
    "CapAdd": null,
    "CapDrop": null,
    "Dns": [],
    "DnsOptions": [],
    "DnsSearch": []
--More--(32%)

```

Creating a Volume

A volume can be created beforehand using the **docker** command. Let's learn more about this command.

Syntax

```
docker volume create --name=volumename --opt options
```

Options

- **name** – This is the name of the volume which needs to be created.
- **opt** – These are options you can provide while creating the volume.

Return Value

The command will output the name of the volume created.

Example

```
sudo docker volume create --name=demo --opt o=size=100m
```

In the above command, we are creating a volume of size 100MB and with a name of demo.

Output

The output of the above command is shown below:

```
demo@ubuntudemo:~$ sudo docker volume create --name=demo --opt o=size=100m
demo
demo@ubuntudemo:~$ _
```

Listing all the Volumes

You can also list all the **docker volumes** on a **docker host**. More details on this command is given below:

Syntax

```
docker volume ls
```

Options

None

Return Value

The command will output all the volumes on the **docker host**.

Example

```
sudo docker volume ls
```

Output

The output of the above command is shown below:

```
demo@ubuntudemo:~$ sudo docker volume ls
DRIVER          VOLUME NAME
local           0329aedc9cb821481d4a6c05619839294af86cfae3494a44b7aee23b1bc8
682c
local           0457e437c2496560355bb02e856d4443ec7e70dd6ceece12044b1cf4d40b
a037
local           3405fca247666c62a05ec15988534c0d385444bc7d5475457bf108a10eb2
f334
local           3cf320ee8bd98f558c25aff2803b300815da575bcc0e5a319e18316618e1
f959
local           8a32b991086de55f3869ae1be7057f14dbc29c3aba70db6726a416670747
d74c
local           9c7e3f37b4f5483c0550f6122b2e8f053d025b6174aecf14e0a12d96081e
c450
local           demo
local           e94311df64b7ad609f851c5c66d0ec04b680c83539cc2721d32697f048fb
fd0f
local           myvolume
demo@ubuntudemo:~$
```

21. Docker – Docker Networking

Docker takes care of the networking aspects so that the containers can communicate with other containers and also with the Docker Host. If you do an **ifconfig** on the Docker Host, you will see the Docker Ethernet adapter. This adapter is created when Docker is installed on the Docker Host.

```
demo@ubuntudemo:~$ sudo ifconfig
docker0  Link encap:Ethernet HWaddr 02:42:b4:a4:43:59
          inet addr:172.17.0.1 Bcast:0.0.0.0 Mask:255.255.0.0
              inet6 addr: fe80::42:b4ff:fea4:4359/64 Scope:Link
                  UP BROADCAST MULTICAST MTU:1500 Metric:1
                  RX packets:55 errors:0 dropped:0 overruns:0 frame:0
                  TX packets:28 errors:0 dropped:0 overruns:0 carrier:0
                  collisions:0 txqueuelen:0
                  RX bytes:3448 (3.4 KB) TX bytes:2576 (2.5 KB)

eth0      Link encap:Ethernet HWaddr 08:00:27:f5:15:76
          inet addr:192.168.137.200 Bcast:192.168.137.255 Mask:255.255.255.0
              inet6 addr: fe80::a00:27ff:fe5:1576/64 Scope:Link
                  UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
                  RX packets:199 errors:0 dropped:0 overruns:0 frame:0
                  TX packets:70 errors:0 dropped:0 overruns:0 carrier:0
                  collisions:0 txqueuelen:1000
                  RX bytes:13734 (13.7 KB) TX bytes:5238 (5.2 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
              inet6 addr: ::1/128 Scope:Host
                  UP LOOPBACK RUNNING MTU:65536 Metric:1
                  RX packets:40 errors:0 dropped:0 overruns:0 frame:0
                  TX packets:40 errors:0 dropped:0 overruns:0 carrier:0
                  collisions:0 txqueuelen:0
                  RX bytes:3184 (3.1 KB) TX bytes:3184 (3.1 KB)

demo@ubuntudemo:~$
```



This is a bridge between the Docker Host and the Linux Host. Now let's look at some commands associated with networking in Docker.

Listing All Docker Networks

This command can be used to list all the networks associated with Docker on the host.

Syntax

```
docker network ls
```

Options

None

Return Value

The command will output all the networks on the Docker Host.

Example

```
sudo docker network ls
```

Output

The output of the above command is shown below

```
demo@ubuntudemo:~$ sudo docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
f07aad6ccadf   bridge    bridge      local
faae6bf679ea   host      host       local
54a2d37e7e00   none     null       local
demo@ubuntudemo:~$
```

Inspecting a Docker network

If you want to see more details on the network associated with Docker, you can use the Docker **network inspect** command.

Syntax

```
docker network inspect networkname
```

Options

- **networkname** – This is the name of the network you need to inspect.

Return Value

The command will output all the details about the network.

Example

```
sudo docker network inspect bridge
```

Output

The output of the above command is shown below:

```

{
    "Name": "bridge",
    "Id": "f07aad6ccadf388082ccf9ad37db43f78adec85fb96ae0b2e9e8390c6d674242"
    ,
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
        "Driver": "default",
        "Options": null,
        "Config": [
            {
                "Subnet": "172.17.0.0/16",
                "Gateway": "172.17.0.1"
            }
        ]
    },
    "Internal": false,
    "Containers": {},
    "Options": {
        "com.docker.network.bridge.default_bridge": "true",
        "com.docker.network.bridge.enable_icc": "true",
        "com.docker.network.bridge.enable_ip_masquerade": "true",
        "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
        "com.docker.network.bridge.name": "docker0",
        "com.docker.network.driver.mtu": "1500"
    },
    "Labels": {}
}
]
demo@ubuntudemo:~$ 

```



Now let's run a container and see what happens when we inspect the network again. Let's spin up an Ubuntu container with the following command:

```
sudo docker run -it ubuntu:latest /bin/bash
```

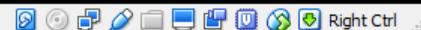
```
demo@ubuntudemo:~$ sudo docker run -it ubuntu:latest /bin/bash
```

Now if we inspect our network name via the following command, you will now see that the container is attached to the bridge.

```
sudo docker network inspect bridge
```

```

        "Subnet": "172.17.0.0/16",
        "Gateway": "172.17.0.1"
    }
]
},
"Internal": false,
"Containers": [
    "8e7b9a6dc121ba1c9a9fe48542db0149ee87b5efe031f518fb15751741ea0447": {
        "Name": "suspicious_blackwell",
        "EndpointID": "d30971d663e91ec2439355bb43c99613d500e35fbaae1957a7f74cb650f40723",
        "MacAddress": "02:42:ac:11:00:02",
        "IPv4Address": "172.17.0.2/16",
        "IPv6Address": ""
    }
],
"Options": {
    "com.docker.network.bridge.default_bridge": "true",
    "com.docker.network.bridge.enable_icc": "true",
    "com.docker.network.bridge.enable_ip_masquerade": "true",
    "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
    "com.docker.network.bridge.name": "docker0",
    "com.docker.network.driver.mtu": "1500"
},
"Labels": {}
]
demo@ubuntudemo:~$
```



Creating Your Own New Network

One can create a network in Docker before launching containers. This can be done with the following command:

Syntax

```
docker network create --driver drivername name
```

Options

- **drivername** – This is the name used for the network driver.
- **name** – This is the name given to the network.

Return Value

The command will output the long ID for the new network.

Example

```
sudo docker network create --driver bridge new_nw
```

Output

The output of the above command is shown below:

```
demo@ubuntudemo:~$ sudo docker network create --driver bridge new_nw
f01b64dc09425cc4906e20b5e17765e3248ea727068e0e2172bf4aec42586fe
demo@ubuntudemo:~$ _
```

You can now attach the new network when launching the container. So let's spin up an Ubuntu container with the following command:

```
sudo docker run -it --network=new_nw ubuntu:latest /bin/bash
```

```
demo@ubuntudemo:~$ sudo docker run -it --network=new_nw ubuntu:latest /bin/bash
```

And now when you inspect the network via the following command, you will see the container attached to the network.

```
sudo docker network inspect new_nw
```

```
"Scope": "local",
"Driver": "bridge",
"EnableIPv6": false,
"IPAM": {
    "Driver": "default",
    "Options": {},
    "Config": [
        {
            "Subnet": "172.18.0.0/16",
            "Gateway": "172.18.0.1/16"
        }
    ]
},
"Internal": false,
"Containers": {
    "38604fc42bcb5f78d42a8f40f34fa245301b2020a84c9e602786d2103ca6b847": {
        "Name": "boring_dubinsky",
        "EndpointID": "74d6b14a6393bf3081d5d9ec012b5b76b2ead49e85a5f664ca621761a9e69612",
        "MacAddress": "02:42:ac:12:00:02",
        "IPv4Address": "172.18.0.2/16",
        "IPv6Address": ""
    },
    "Options": {},
    "Labels": {}
}
]
```

```
demo@ubuntudemo:~$ _
```

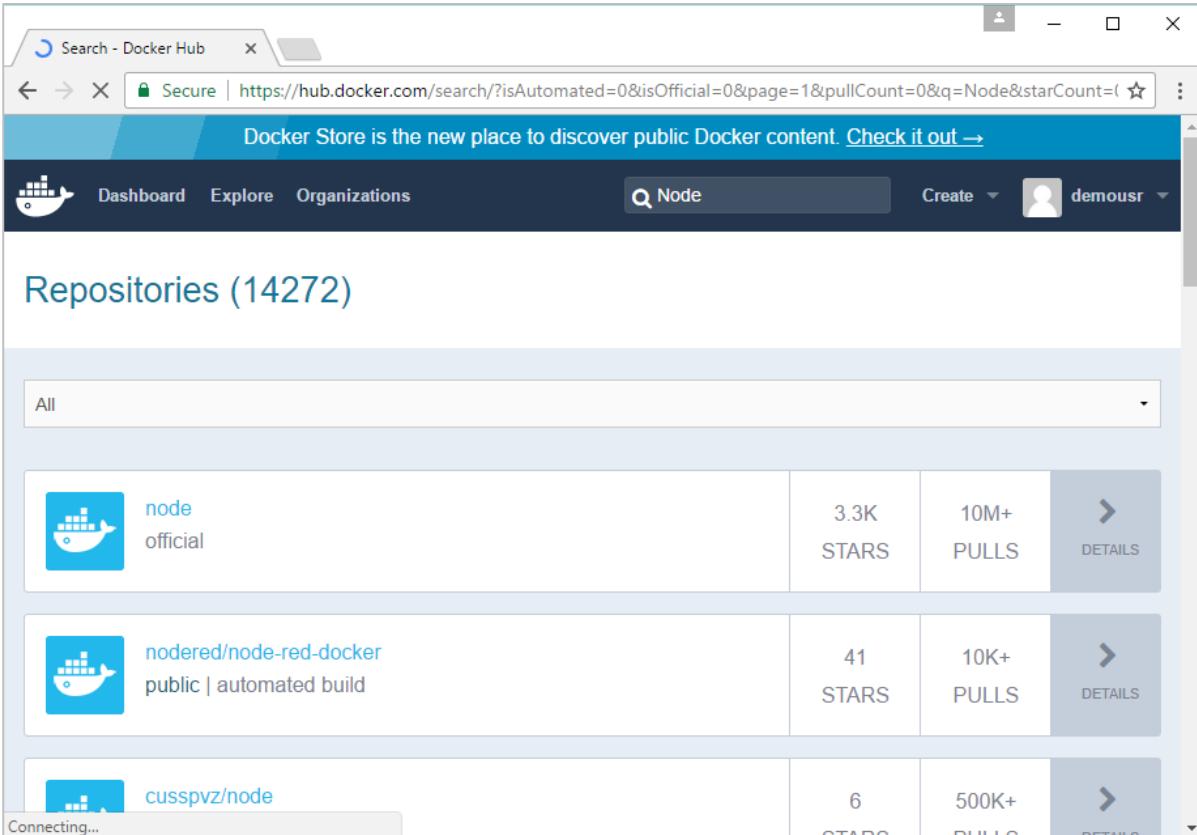


22. Docker – Setting Node.js

Node.js is a JavaScript framework that is used for developing server-side applications. It is an open source framework that is developed to run on a variety of operating systems. Since Node.js is a popular framework for development, Docker has also ensured it has support for Node.js applications.

We will now see the various steps for getting the Docker container for Node.js up and running.

Step 1: The first step is to pull the image from Docker Hub. When you log into Docker Hub, you will be able to search and see the image for Node.js as shown below. Just type in Node in the search box and click on the node (official) link which comes up in the search results.



The screenshot shows the Docker Hub interface. The search bar at the top contains the query 'Node'. Below the search bar, the heading 'Repositories (14272)' is visible. The main area displays three repository cards:

Repository	Owner	Stars	Pulls	Action
node	official	3.3K	10M+	DETAILS
nodered/node-red-docker	public automated build	41	10K+	DETAILS
cusspvz/node	Connecting...	6	500K+	DETAILS

Step 2: You will see that the Docker **pull** command for node in the details of the repository in Docker Hub.

The screenshot shows the Docker Hub repository page for 'library/node'. At the top, there is a message encouraging users to discuss their plans through GitHub issues. Below this, the word 'Documentation' is prominently displayed. A note below it says: 'Documentation for this image is stored in the `node/` directory of the `docker-library/docs` GitHub repo. Be sure to familiarize yourself with the repository's `README.md` file before attempting a pull request.' Under the documentation, there is a section titled 'Docker Pull Command' containing the command 'docker pull node'. At the bottom of the page, there is a section for 'Comments' with a count of 52 and a blue 'Add Comment' button. The page has a standard header with back, forward, and search buttons, and a URL bar showing 'https://hub.docker.com/_/node/'.

Step 3: On the Docker Host, use the Docker **pull** command as shown above to download the latest node image from Docker Hub.

```
demo@ubuntudemo:~$ sudo docker pull node_
```

Once the **pull** is complete, we can then proceed with the next step.

```
demo@ubuntudemo:~$ sudo docker pull node
Using default tag: latest
latest: Pulling from library/node

75a822cd7888: Downloading 31.54 MB/39.73 MB
75a822cd7888: Pull complete
57de64c72267: Pull complete
4306be1e8943: Pull complete
871436ab7225: Pull complete
0110c26a367a: Pull complete
1f04fe713f1b: Pull complete
723bac39028e: Pull complete
Digest: sha256:08d77f1984cf79739ba7c987636cb871fd69745754200e5891a0c7ee2d9965b0
Status: Downloaded newer image for node:latest
demo@ubuntudemo:~$
```

Step 4: On the Docker Host, let's use the **vim** editor and create one Node.js example file. In this file, we will add a simple command to display "HelloWorld" to the command prompt.

```
demo@ubuntudemo:~$ vim HelloWorld.js
```

In the Node.js file, let's add the following statement:

```
Console.log('Hello World');
```

This will output the "Hello World" phrase when we run it through Node.js.

```
console.log('Hello World');_  
~  
~  
~
```

Ensure that you save the file and then proceed to the next step.

Step 5: To run our Node.js script using the Node Docker container, we need to execute the following statement:

```
sudo docker run -it -rm --name>HelloWorld -v "$PWD":/usr/src/app -w /usr/src/app  
node node HelloWorld.js
```

The following points need to be noted about the above command:

- The **-rm** option is used to remove the container after it is run.
- We are giving a name to the container called "HelloWorld"
- We are mentioning to map the volume in the container which is **/usr/src/app** to our current present working directory. This is done so that the node container will pick up our HelloWorld.js script which is present in our working directory on the Docker Host.
- The **-w** option is used to specify the working directory used by Node.js.
- The first node option is used to specify to run the node image.
- The second node option is used to mention to run the node command in the node container.
- And finally we mention the name of our script.

We will then get the following output. And from the output, we can clearly see that the Node container ran as a container and executed the HelloWorld.js script.

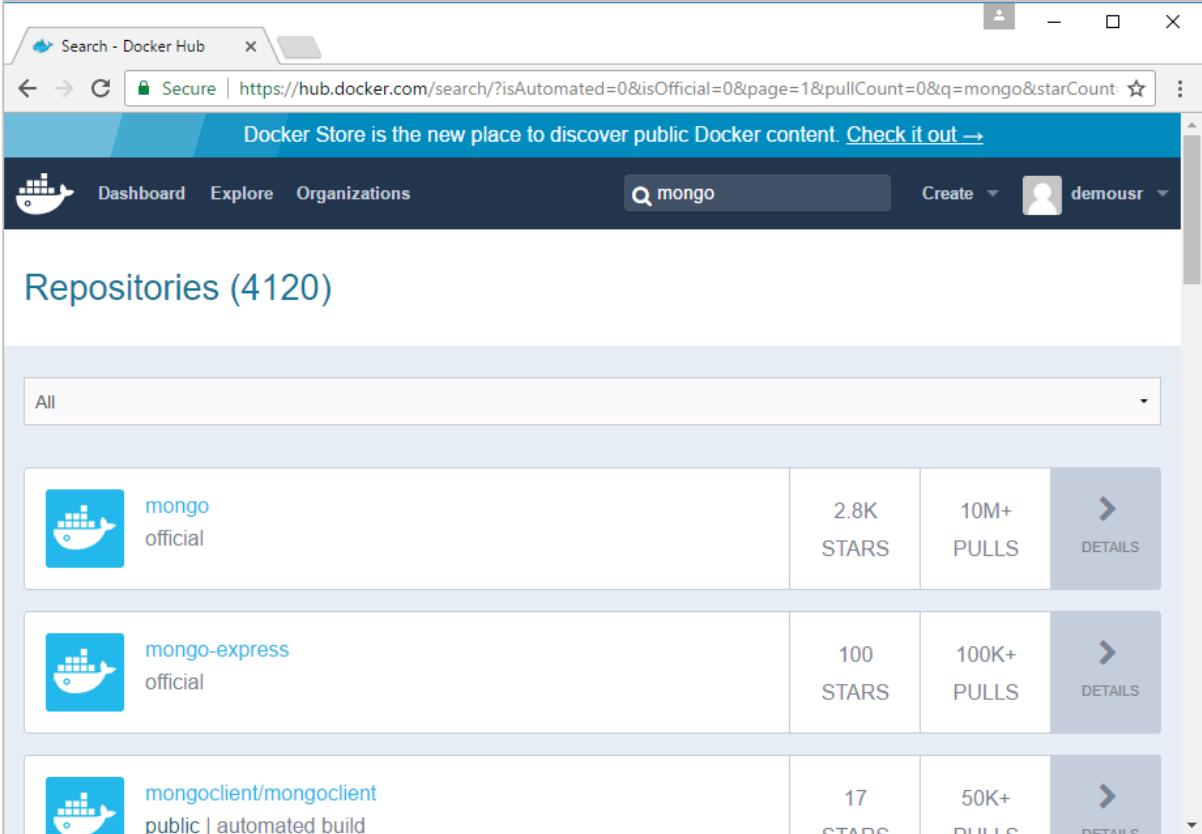
```
demo@ubuntudemo:~$ sudo docker run -it --rm --name>HelloWorld -v "$PWD":/usr/src/app -w /usr/src/app node node HelloWorld.js  
Hello World  
demo@ubuntudemo:~$
```

23. Docker – Setting MongoDB

MongoDB is a famous document-oriented database that is used by many modern-day web applications. Since MongoDB is a popular database for development, Docker has also ensured it has support for MongoDB.

We will now see the various steps for getting the Docker container for MongoDB up and running.

Step 1: The first step is to pull the image from Docker Hub. When you log into Docker Hub, you will be able to search and see the image for Mongo as shown below. Just type in Mongo in the search box and click on the Mongo (official) link which comes up in the search results.



The screenshot shows the Docker Hub interface with a search query of "mongo" entered in the search bar. The results page displays three official MongoDB Docker images:

Image Name	Type	Stars	Pulls	Action
mongo	official	2.8K	10M+	DETAILS
mongo-express	official	100	100K+	DETAILS
mongoclient/mongoclient	public automated build	17	50K+	DETAILS

Step 2: You will see that the Docker **pull** command for Mongo in the details of the repository in Docker Hub.

The screenshot shows a web browser window with the URL https://hub.docker.com/_/mongo/. The page title is "library/mongo - Docker". The main content is titled "Documentation" and states: "Documentation for this image is stored in the `mongo/` directory of the `docker-library/docs` GitHub repo. Be sure to familiarize yourself with the repository's `README.md` file before attempting a pull request." Below this is a "Docker Pull Command" section containing the command `docker pull mongo`. At the bottom, there is a "Comments (51)" section with a "Add Comment" button and a pagination link "1 2 »". A user comment by "xiegh" from 8 days ago is visible.

Step 3: On the Docker Host, use the Docker **pull** command as shown above to download the latest Mongo image from Docker Hub.

```
demo@ubuntudemo:~$ sudo docker pull mongo
```



```
demo@ubuntudemo:~$ sudo docker pull mongo
[sudo] password for demo:
Using default tag: latest
latest: Pulling from library/mongo

75a822cd7888: Already exists
3bf369f658b6: Pull complete
7d7cb343d20e: Pull complete
73a933a908f7: Pull complete
658569c28c55: Pull complete
124a8bf940da: Pull complete
7c19551df503: Pull complete
a18347fe18d9: Pull complete
53e710c6ec29: Pull complete
Digest: sha256:23e5cdbd9bc26a6d1ae4db8252a295d6bdba8332dec68483816d5b7bb2438d7
Status: Downloaded newer image for mongo:latest
demo@ubuntudemo:~$
```

Step 4: Now that we have the image for Mongo, let's first run a MongoDB container which will be our instance for MongoDB. For this, we will issue the following command:

```
sudo docker -it -d mongo
```

The following points can be noted about the above command:

- The **-it** option is used to run the container in interactive mode.
- The **-d** option is used to run the container as a daemon process.
- And finally we are creating a container from the Mongo image.

You can then issue the **docker ps** command to see the running containers:

```
demo@ubuntudemo:~$ sudo docker run -it -d mongo
ec086eec7416e368614de631b8356fcf68eec978b01b620251cb55d8b7ec7189
demo@ubuntudemo:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES
STATUS              PORTS              NAMES
ec086eec7416        mongo              "/entrypoint.sh mongo"   3 seconds ago   Up 2 seconds      tender_poitras
demo@ubuntudemo:~$ _
```

Take a note of the following points:

- The name of the container is **tender_poitras**. This name will be different since the name of the containers keep on changing when you spin up a container. But just make a note of the container which you have launched.
- Next, also notice the port number it is running on. It is listening on the TCP port of 27017.

Step 5: Now let's spin up another container which will act as our client which will be used to connect to the MongoDB database. Let's issue the following command for this:

```
sudo docker run -it --link=tender_poitras:mongo mongo /bin/bash
```

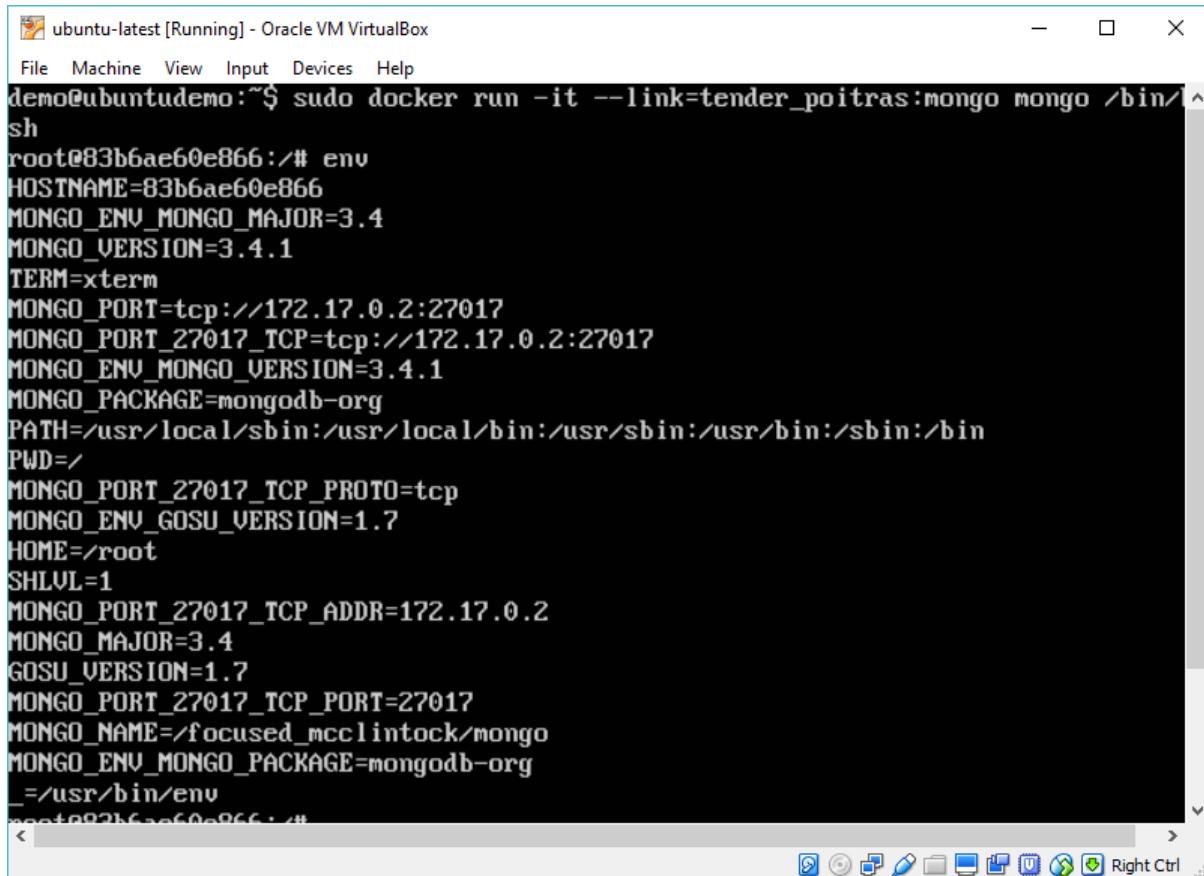
The following points can be noted about the above command:

- The **-it** option is used to run the container in interactive mode.
- We are now linking our new container to the already launched MongoDB server container. Here, you need to mention the name of the already launched container.
- We are then specifying that we want to launch the Mongo container as our client and then run the **bin/bash** shell in our new container.

```
demo@ubuntudemo:~$ sudo docker run -it --link=tender_poitras:mongo mongo /bin/
sh
root@83b6ae60e866:/#
```

You will now be in the new container.

Step 5: Run the **env** command in the new container to see the details of how to connect to the MongoDB server container.



```
ubuntu-latest [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
demo@ubuntudemo:~$ sudo docker run -it --link=tender_poitras:mongo mongo /bin/sh
root@83b6ae60e866:/# env
HOSTNAME=83b6ae60e866
MONGO_ENV_MONGO_MAJOR=3.4
MONGO_VERSION=3.4.1
TERM=xterm
MONGO_PORT=tcp://172.17.0.2:27017
MONGO_PORT_27017_TCP=tcp://172.17.0.2:27017
MONGO_ENV_MONGO_VERSION=3.4.1
MONGO_PACKAGE=mongodb-org
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
MONGO_PORT_27017_TCP_PROTO=tcp
MONGO_ENV_GOSU_VERSION=1.7
HOME=/root
SHLVL=1
MONGO_PORT_27017_TCP_ADDR=172.17.0.2
MONGO_MAJOR=3.4
GOSU_VERSION=1.7
MONGO_PORT_27017_TCP_PORT=27017
MONGO_NAME=/focused_mcclintock/mongo
MONGO_ENV_MONGO_PACKAGE=mongodb-org
_=~/usr/bin/env
root@83b6ae60e866:~#
```

Step 6: Now it's time to connect to the MongoDB server from the client container. We can do this via the following command:

```
mongo 172.17.0.2:27017
```

The following points need to be noted about the above command

- The **mongo** command is the client **mongo** command that is used to connect to a MongoDB database.
- The IP and port number is what you get when you use the **env** command.

Once you run the command, you will then be connected to the MongoDB database.

```
http://groups.google.com/group/mongodb-user
Server has startup warnings:
2017-01-07T15:26:23.769+0000 I STORAGE [initandlisten]
2017-01-07T15:26:23.769+0000 I STORAGE [initandlisten] ** WARNING: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine
2017-01-07T15:26:23.769+0000 I STORAGE [initandlisten] ** See http://dochub.mongodb.org/core/prodnotes-filesystem
2017-01-07T15:26:23.873+0000 I CONTROL [initandlisten]
2017-01-07T15:26:23.874+0000 I CONTROL [initandlisten] ** WARNING: Access control is not enabled for the database.
2017-01-07T15:26:23.874+0000 I CONTROL [initandlisten] ** Read and write access to data and configuration is unrestricted.
2017-01-07T15:26:23.874+0000 I CONTROL [initandlisten]
2017-01-07T15:26:23.874+0000 I CONTROL [initandlisten]
2017-01-07T15:26:23.875+0000 I CONTROL [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/enabled is 'always'.
2017-01-07T15:26:23.875+0000 I CONTROL [initandlisten] ** We suggest setting it to 'never'
2017-01-07T15:26:23.875+0000 I CONTROL [initandlisten]
2017-01-07T15:26:23.875+0000 I CONTROL [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/defrag is 'always'.
2017-01-07T15:26:23.875+0000 I CONTROL [initandlisten] ** We suggest setting it to 'never'
2017-01-07T15:26:23.875+0000 I CONTROL [initandlisten]
>
< [  ] Right Ctrl .
```

You can then run any MongoDB command in the command prompt. In our example, we are running the following command:

```
use demo
```

This command is a MongoDB command which is used to switch to a database name **demo**. If the database is not available, it will be created.

```
2017-01-07T15:26:23.769+0000 I STORAGE [initandlisten]
2017-01-07T15:26:23.769+0000 I STORAGE [initandlisten] ** WARNING: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine
2017-01-07T15:26:23.769+0000 I STORAGE [initandlisten] ** See http://dochub.mongodb.org/core/prodnotes-filesystem
2017-01-07T15:26:23.873+0000 I CONTROL [initandlisten]
2017-01-07T15:26:23.874+0000 I CONTROL [initandlisten] ** WARNING: Access control is not enabled for the database.
2017-01-07T15:26:23.874+0000 I CONTROL [initandlisten] ** Read and write access to data and configuration is unrestricted.
2017-01-07T15:26:23.874+0000 I CONTROL [initandlisten]
2017-01-07T15:26:23.874+0000 I CONTROL [initandlisten]
2017-01-07T15:26:23.875+0000 I CONTROL [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/enabled is 'always'.
2017-01-07T15:26:23.875+0000 I CONTROL [initandlisten] ** We suggest setting it to 'never'
2017-01-07T15:26:23.875+0000 I CONTROL [initandlisten]
2017-01-07T15:26:23.875+0000 I CONTROL [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/defrag is 'always'.
2017-01-07T15:26:23.875+0000 I CONTROL [initandlisten] ** We suggest setting it to 'never'
2017-01-07T15:26:23.875+0000 I CONTROL [initandlisten]
> use demo
switched to db demo
>
```

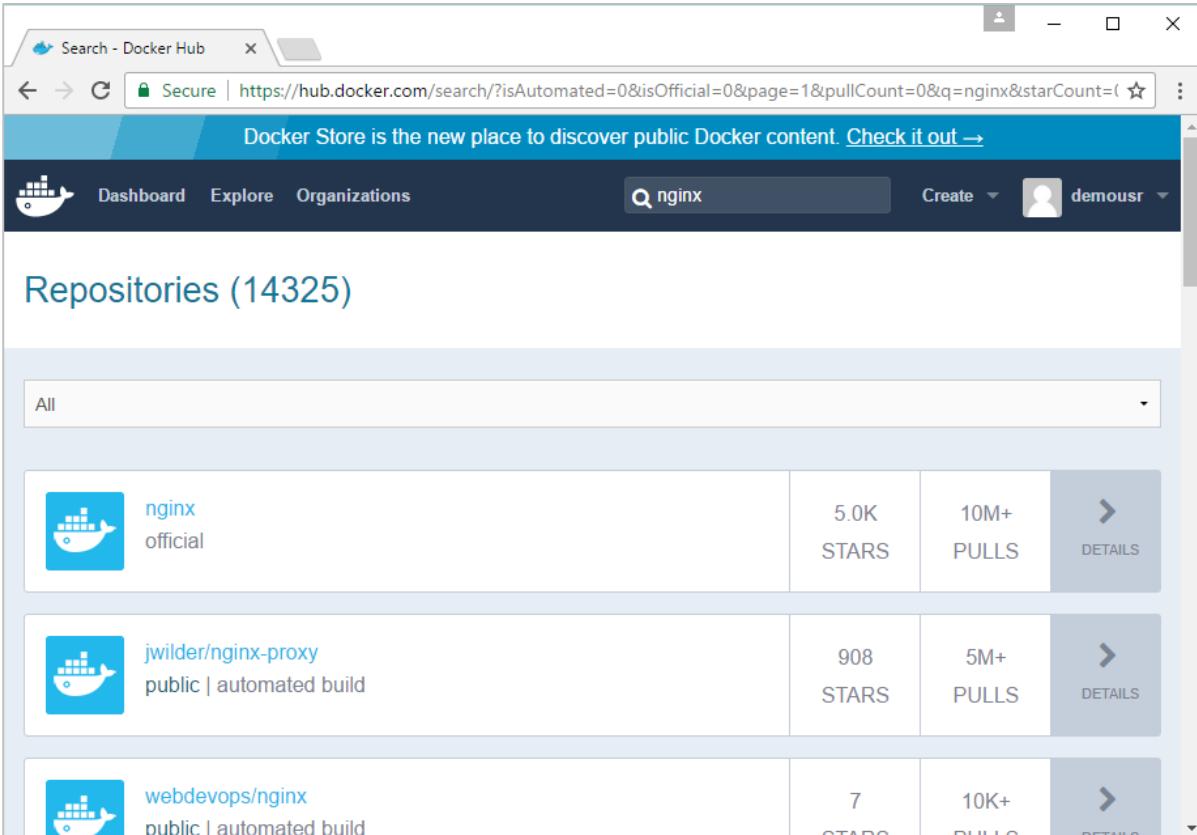
Now you have successfully created a client and server MongoDB container.

24. Docker – Setting NGINX

NGINX is a popular lightweight web application that is used for developing server-side applications. It is an open-source web server that is developed to run on a variety of operating systems. Since **nginx** is a popular web server for development, Docker has ensured that it has support for **nginx**.

We will now see the various steps for getting the Docker container for **nginx** up and running.

Step 1: The first step is to pull the image from Docker Hub. When you log into Docker Hub, you will be able to search and see the image for **nginx** as shown below. Just type in **nginx** in the search box and click on the **nginx (official)** link which comes up in the search results.



The screenshot shows the Docker Hub interface. In the search bar at the top, the word "nginx" is typed. Below the search bar, the text "Docker Store is the new place to discover public Docker content. Check it out →" is displayed. Underneath, there are tabs for "Dashboard", "Explore", and "Organizations". On the right side of the header, there is a search icon and a user profile icon for "demousr". The main area is titled "Repositories (14325)". A dropdown menu shows "All" selected. Below this, three repository cards are listed:

Repository	Description	Stars	Pulls	Action
nginx official		5.0K STARS	10M+ PULLS	DETAILS
jwilder/nginx-proxy public automated build		908 STARS	5M+ PULLS	DETAILS
webdevops/nginx public automated build		7 STARS	10K+ PULLS	DETAILS

Step 2: You will see that the Docker **pull** command for **nginx** in the details of the repository in Docker Hub.

The screenshot shows a web browser window with the URL https://hub.docker.com/_/nginx/. The page content is as follows:

- Documentation:** A note about contributing to the GitHub repository.
- Docker Pull Command:** A text input field containing the command `docker pull nginx`.
- Comments (73):** A button labeled "Add Comment".

Step 3: On the Docker Host, use the Docker **pull** command as shown above to download the latest **nginx** image from Docker Hub.

```
demo@ubuntudemo:~$ sudo docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx

75a822cd7888: Already exists
0aefb9dc4a57: Pull complete
046e44ee6057: Pull complete
Digest: sha256:fab482910aae9630c93bd24fc6fcecb9f9f792c24a8974f5e46d8ad625ac235
Status: Downloaded newer image for nginx:latest
demo@ubuntudemo:~$ _
```

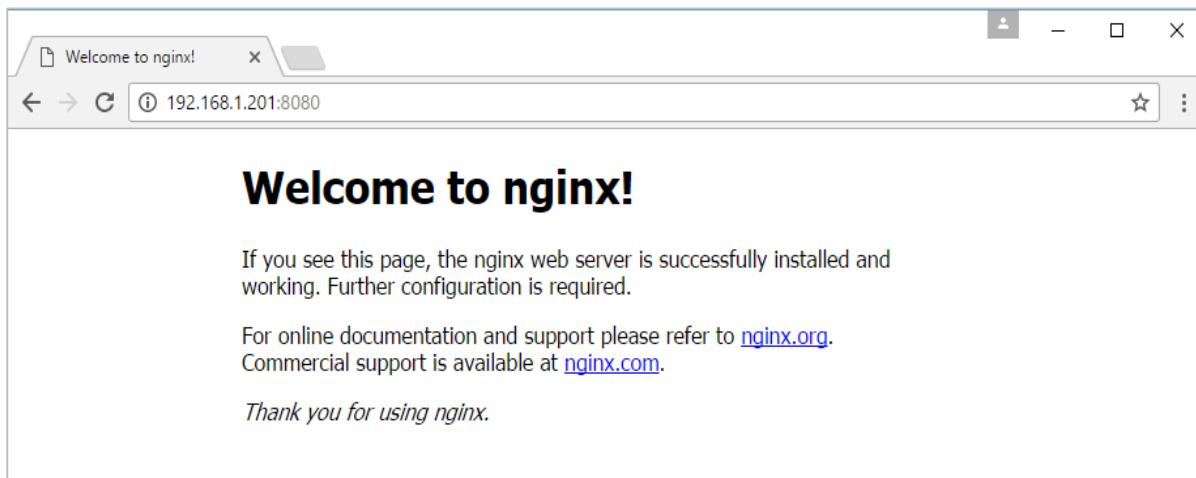
Step 4: Now let's run the **nginx** container via the following command:

```
sudo docker run -p 8080:80 -d nginx
```

We are exposing the port on the **nginx** server which is port 80 to the port 8080 on the Docker Host.

```
demo@ubuntudemo:~$ sudo docker run -p 8080:80 -d nginx
[sudo] password for demo:
605d088b4d044f40b558fd321df2b40cadfcad5d0c59a947bddfffdc0c5151e4f
demo@ubuntudemo:~$
```

Once you run the command, you will get the following output if you browse to the URL <http://dockerhost:8080>. This shows that the **nginx** container is up and running.



Step 5: Let's look at another example where we can host a simple web page in our **nginx** container. In our example, we will create a simple **HelloWorld.html** file and host it in our **nginx** container.

Let's first create an HTML file called **HelloWorld.html**

```
demo@ubuntudemo:~$ sudo vim HelloWorld.html_
```

Let's add a simple line of Hello World in the HTML file.

```
Hello World
```

Let's then run the following Docker command.

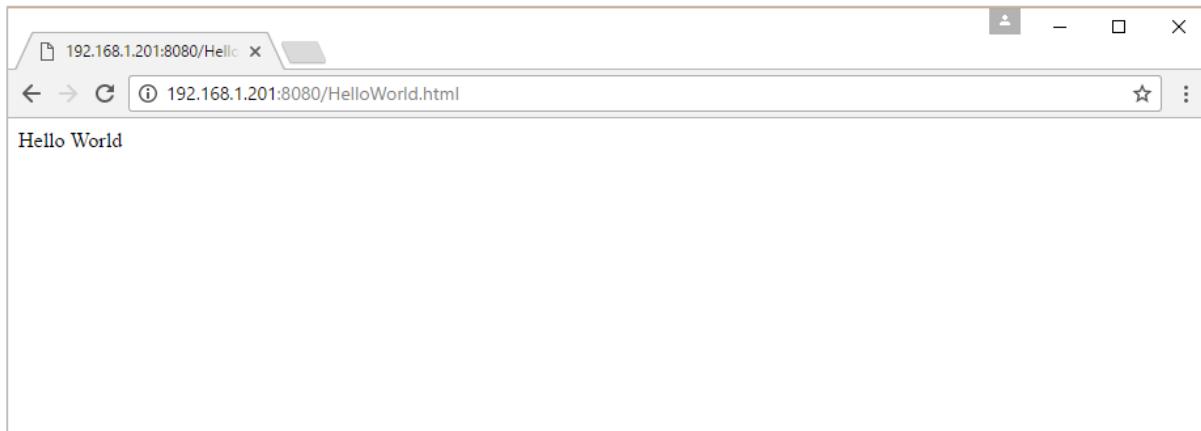
```
sudo docker run -p 8080:80 -v "$PWD":/usr/share/nginx/html:ro -d nginx
```

The following points need to be noted about the above command:

- We are exposing the port on the **nginx** server which is port 80 to the port 8080 on the Docker Host.
- Next, we are attaching the volume on the container which is **/usr/share/nginx/html** to our present working directory. This is where our **HelloWorld.html** file is stored.

```
demo@ubuntudemo:~$ sudo docker run -p 8080:80 -v "$PWD":/usr/share/nginx/html:~ -d nginx
bee9ef3fb04f2f3a219fc60123989351c63d4074a32a765340491f28ad3dbc44
demo@ubuntudemo:~$ 
demo@ubuntudemo:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES
STATUS              PORTS              NAMES
bee9ef3fb04f        nginx              "nginx -g 'daemon off'"   About a minute ago      ago    Up 50 seconds          443/tcp, 0.0.0.0:8080->80/tcp   agitated_leavitt
ec086eec7416        mongo              "/entrypoint.sh mongo"  About an hour ago     go    Up About an hour       27017/tcp           tender_poitras
demo@ubuntudemo:~$ _
```

Now if we browse to the URL <http://dockerhost:8080>HelloWorld.html> we will get the following output as expected:



25. Docker – Docker Toolbox

In the introductory chapters, we have seen the installation of Docker toolbox on Windows. The Docker toolbox is developed so that Docker containers can be run on Windows and MacOS. The site for toolbox on Windows is <https://docs.docker.com/docker-for-windows/>



For Windows, you need to have Windows 10 or Windows Server 2016 with Hyper-V enabled.

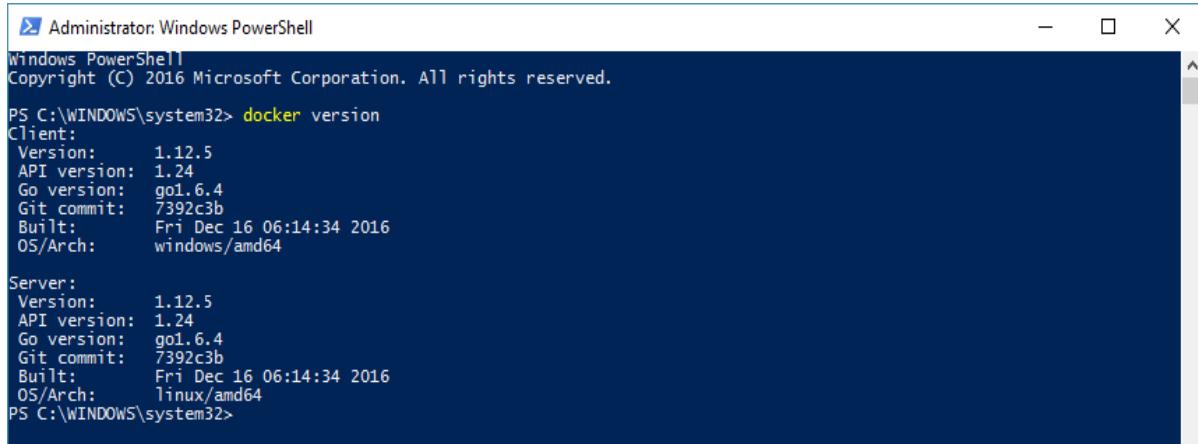
The toolbox consists of the following components:

- **Docker Engine** – This is used as the base engine or Docker daemon that is used to run Docker containers.
- **Docker Machine** for running Docker machine commands.
- **Docker Compose** for running Docker compose commands.
- **Kinematic** – This is the Docker GUI built for Windows and Mac OS.
- **Oracle virtualbox**

Let's now discuss the different types of activities that are possible with Docker toolbox.

Running in Powershell

With Docker toolbox on Windows 10, you can now run Docker commands off **powershell**. If you open powershell on Windows and type in the command of Docker version, you will get all the required details about the Docker version installed.



```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

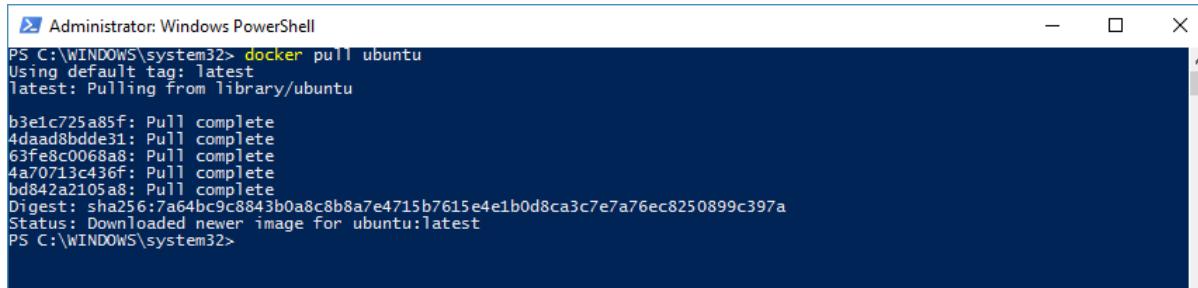
PS C:\WINDOWS\system32> docker version
Client:
  Version: 1.12.5
  API version: 1.24
  Go version: go1.6.4
  Git commit: 7392c3b
  Built: Fri Dec 16 06:14:34 2016
  OS/Arch: windows/amd64

Server:
  Version: 1.12.5
  API version: 1.24
  Go version: go1.6.4
  Git commit: 7392c3b
  Built: Fri Dec 16 06:14:34 2016
  OS/Arch: linux/amd64
PS C:\WINDOWS\system32>
```

Pulling Images and Running Containers

You can also now pull Images from Docker Hub and run containers in powershell as you would do in Linux. The following example will show in brief the downloading of the Ubuntu image and running of the container off the image.

The first step is to use the Docker **pull** command to pull the Ubuntu image from Docker Hub.

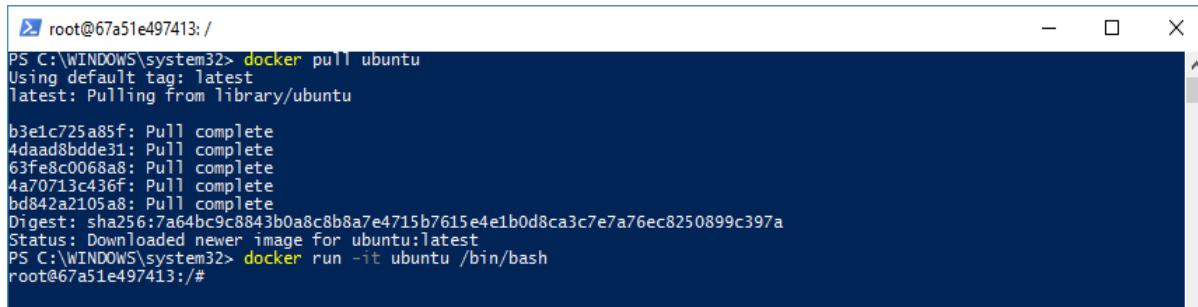


```
Administrator: Windows PowerShell
PS C:\WINDOWS\system32> docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
b3e1c725a85f: Pull complete
4daad8bddde31: Pull complete
63fe8c0068a8: Pull complete
4a70713c436f: Pull complete
bd842a2105a8: Pull complete
Digest: sha256:7a64bc9c8843b0a8c8b8a7e4715b7615e4e1b0d8ca3c7e7a76ec8250899c397a
Status: Downloaded newer image for ubuntu:latest
PS C:\WINDOWS\system32>
```

The next step is to run the Docker image using the following **run** command:

```
docker run -it ubuntu /bin/bash
```

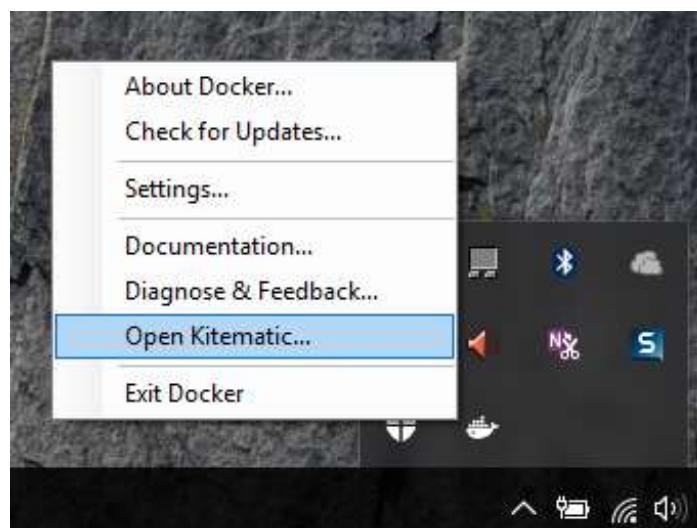
You will notice that the command is the same as it was in Linux.



```
root@67a51e497413:/ 
PS C:\WINDOWS\system32> docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
b3e1c725a85f: Pull complete
4daad8bddde31: Pull complete
63fe8c0068a8: Pull complete
4a70713c436f: Pull complete
bd842a2105a8: Pull complete
Digest: sha256:7a64bc9c8843b0a8c8b8a7e4715b7615e4e1b0d8ca3c7e7a76ec8250899c397a
Status: Downloaded newer image for ubuntu:latest
PS C:\WINDOWS\system32> docker run -it ubuntu /bin/bash
root@67a51e497413:/#
```

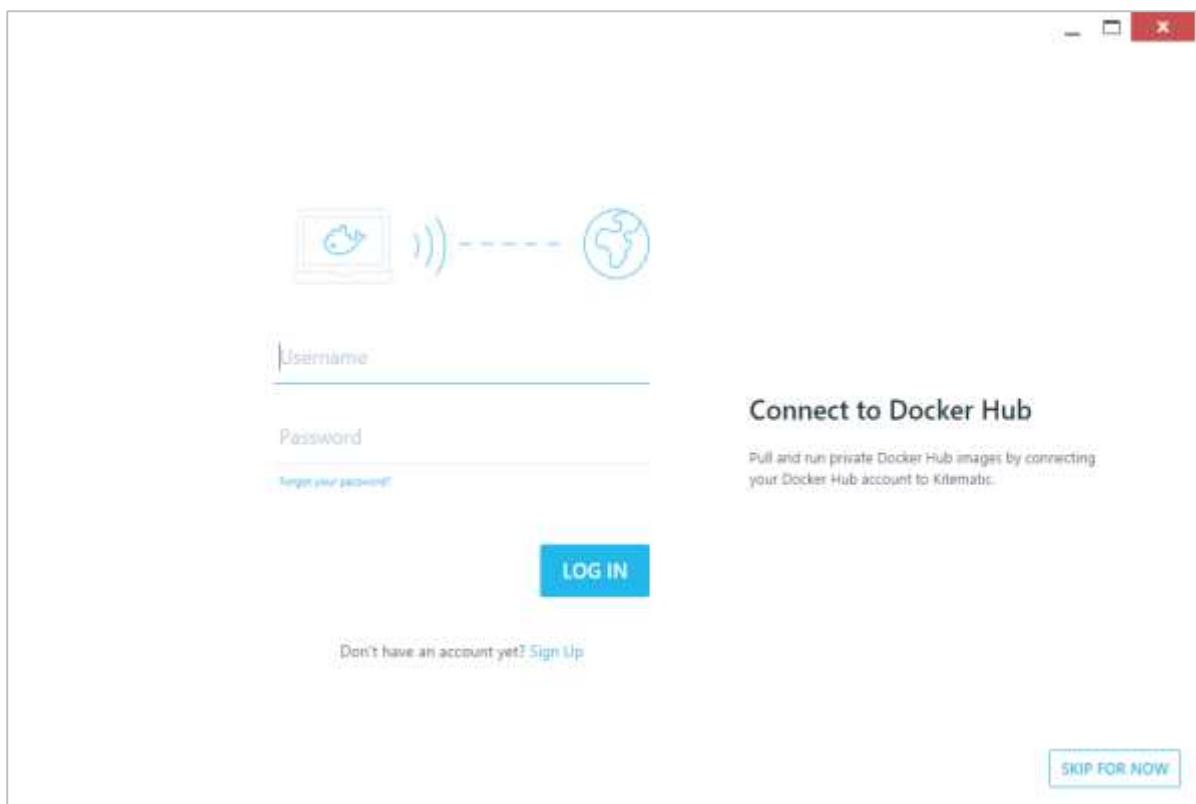
Kitematic

This is the GUI equivalent of Docker on Windows. To open this GUI, go to the taskbar and on the Docker icon, right-click and choose to open Kitematic.

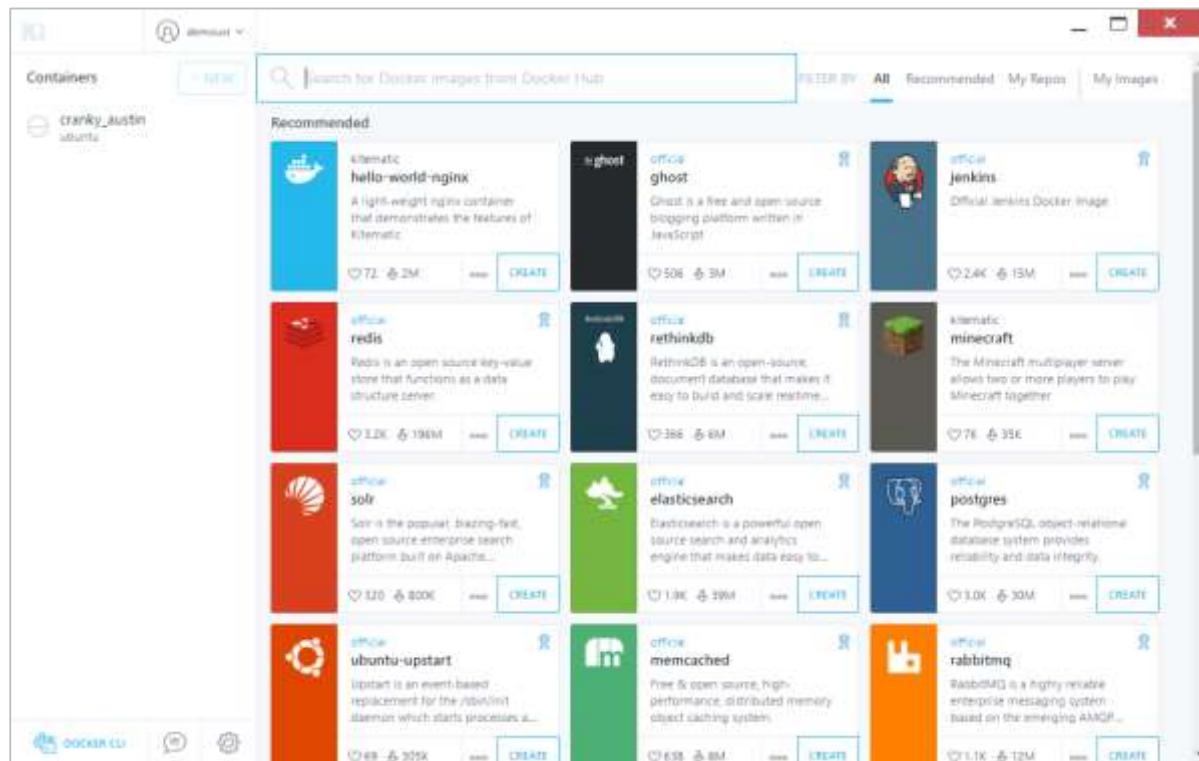


It will prompt you to download Kitematic GUI. Once downloaded, just unzip the contents. There will be a file called **Kitematic.exe**. Double-click this **exe** file to open the GUI interface.

You will then be requested to log into Docker Hub, enter through the GUI. Just enter the required username and password and then click the Login button.



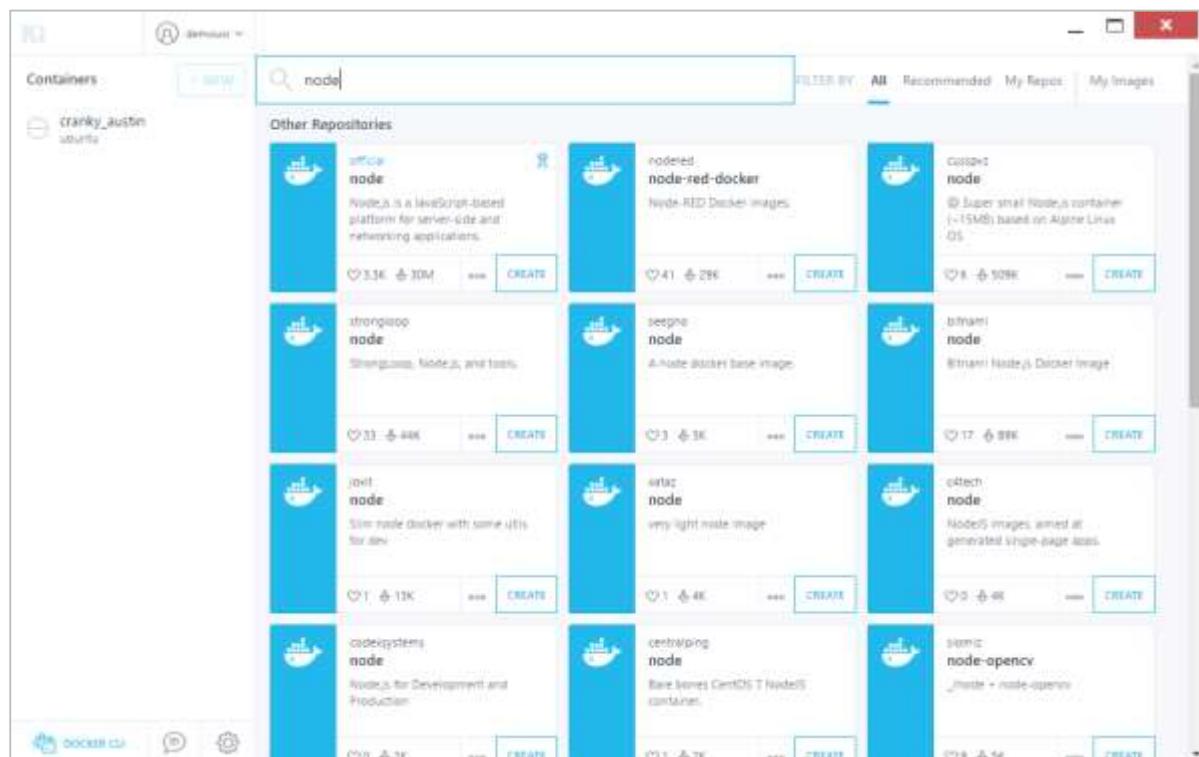
Once logged in, you will be able to see all the images downloaded on the system on the left-hand side of the interface.



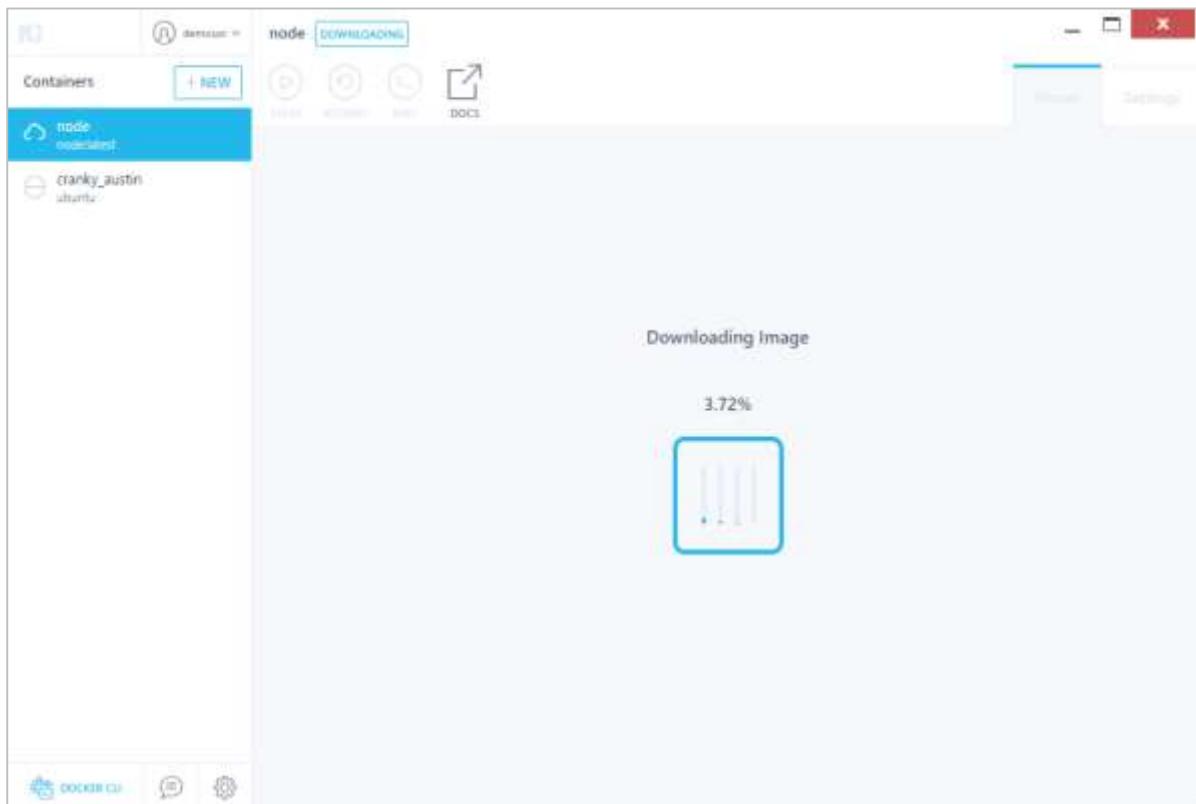
On the right-hand side, you will find all the images available on Docker Hub.

Let's take an example to understand how to download the Node image from Docker Hub using Kitematic.

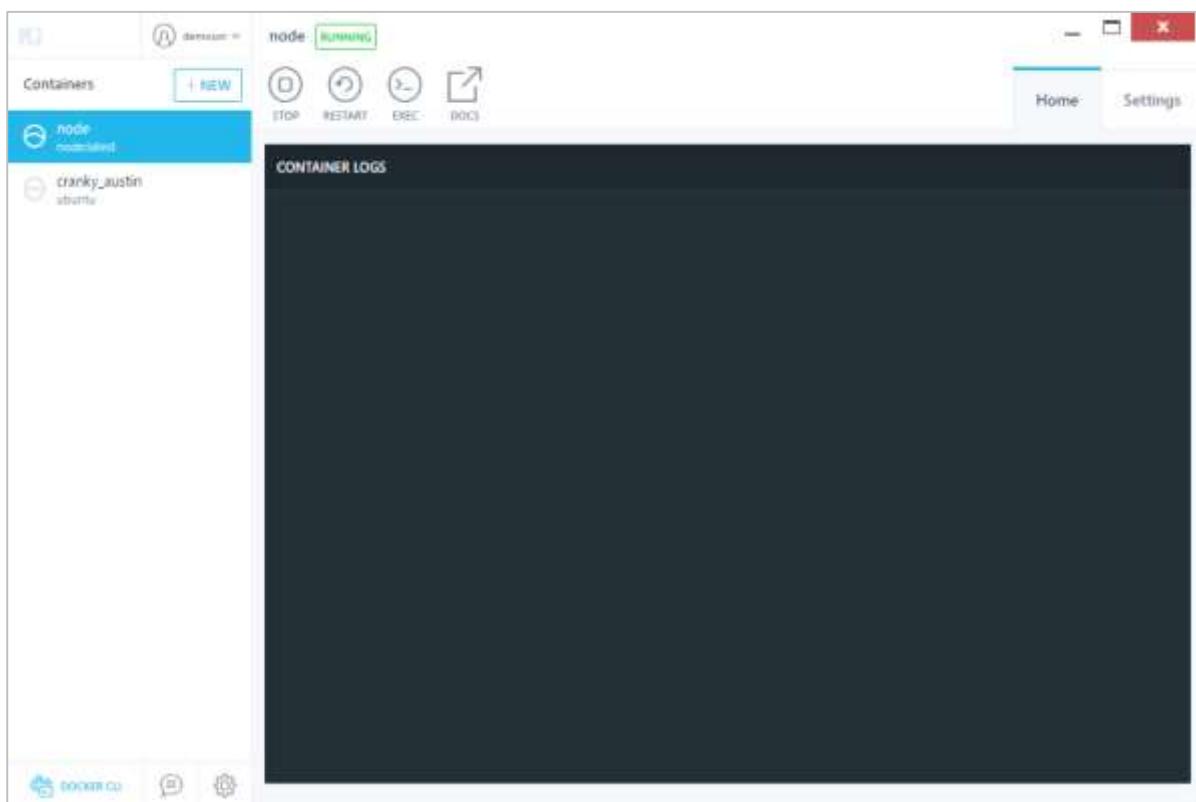
Step 1: Enter the keyword of node in the search criteria.



Step 2: Click the **create** button on official Node image. You will then see the image being downloaded.

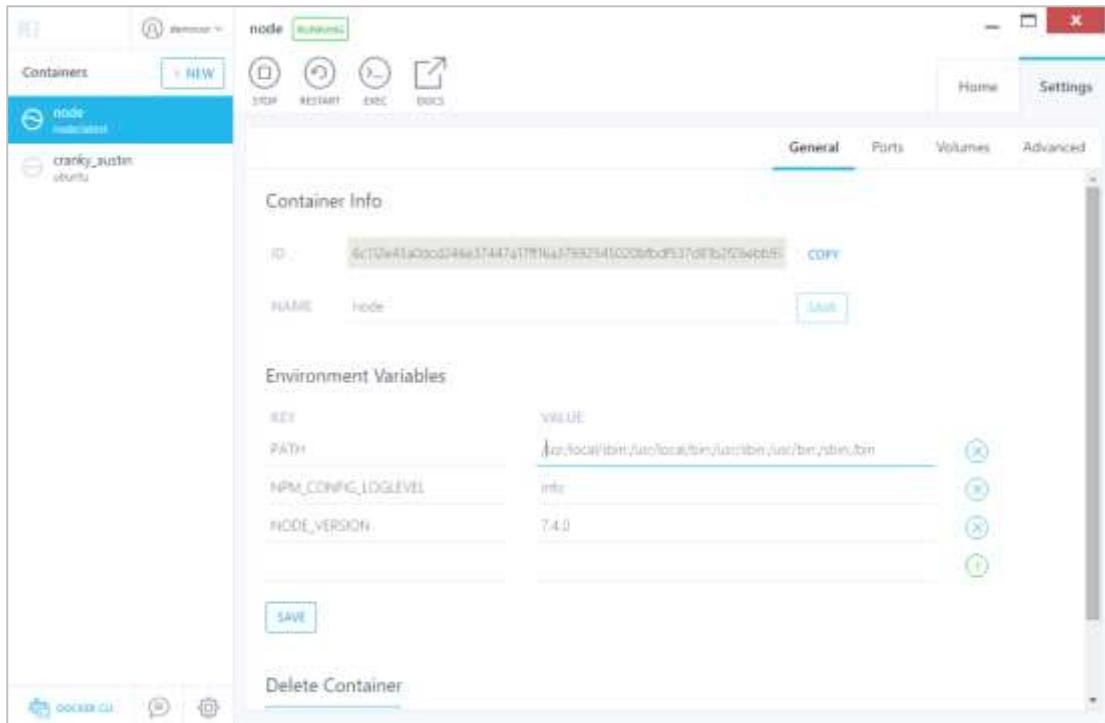


Once the image has been downloaded, it will then start running the Node container.

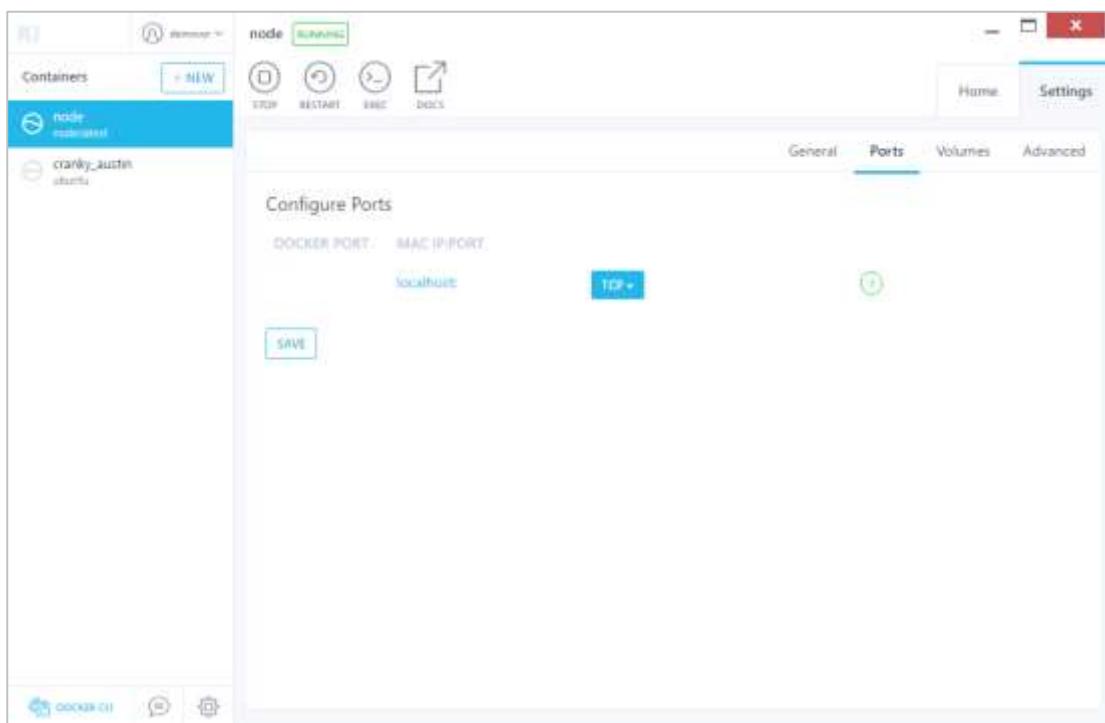


Step 3: If you go to the **settings** tab, you can drill-down to further settings options, as shown below.

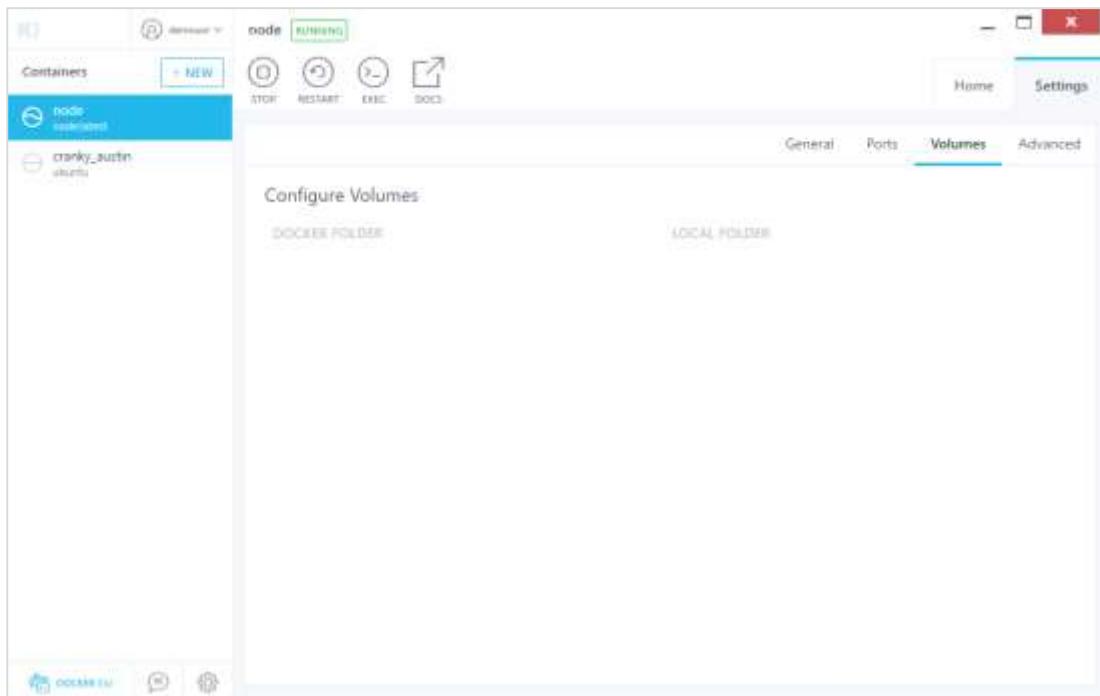
- **General settings** – In this tab, you can name the container, change the path settings, and delete the container.



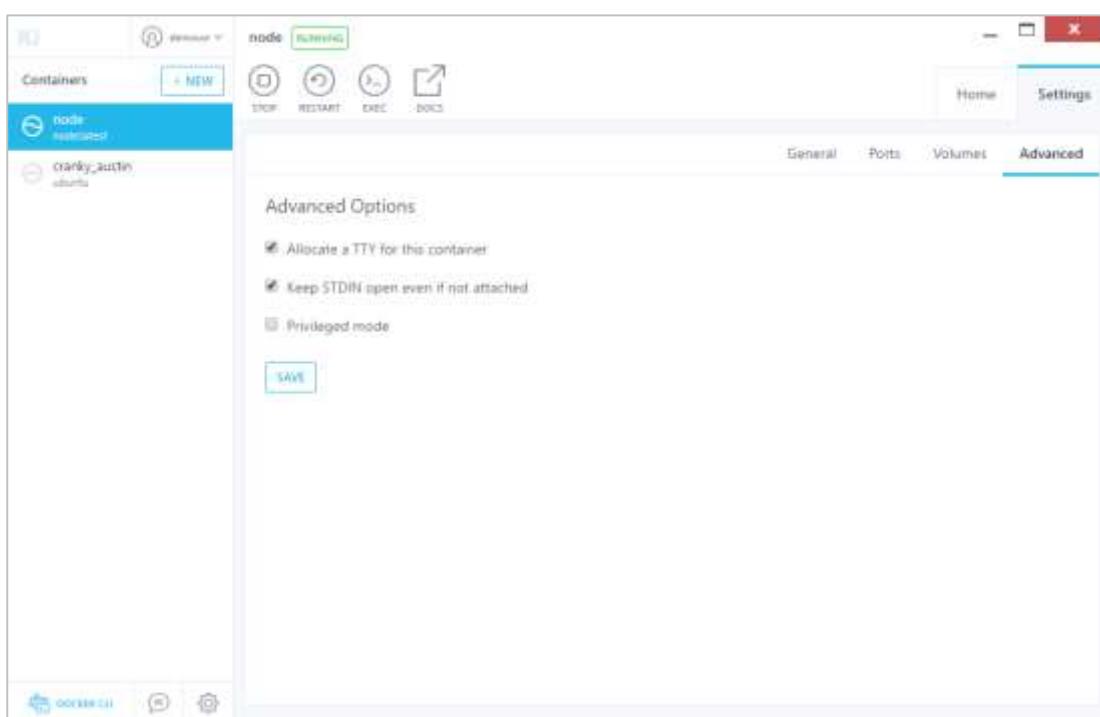
- **Ports** – Here you can see the different port mappings. If you want, you can create your own port mappings.



- **Volumes** - Here you can see the different volume mappings.



- **Advanced** – It contains the advanced settings for the container.



26. Docker – Setting ASP.Net

ASP.Net is the standard web development framework that is provided by Microsoft for developing server-side applications. Since ASP.Net has been around for quite a long time for development, Docker has ensured that it has support for ASP.Net.

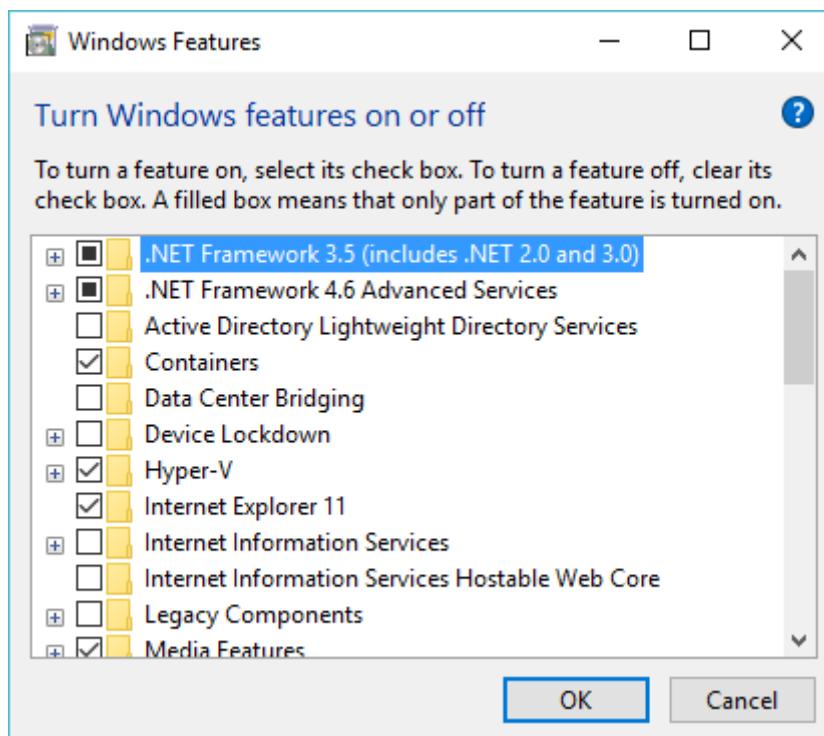
In this chapter, we will see the various steps for getting the Docker container for ASP.Net up and running.

Prerequisites

The following steps need to be carried out first for running ASP.Net.

Step 1: Since this can only run on Windows systems, you first need to ensure that you have either Windows 10 or Window Server 2016.

Step 2: Next, ensure that Hyper-V is and Containers are installed on the Windows system. To install Hyper-V and Containers, you can go to Turn Windows Features ON or OFF. Then ensure the Hyper-V option and Containers is checked and click the OK button.



The system might require a restart after this operation.

Step 3: Next, you need to use the following Powershell command to install the **1.13.0-rc4** version of Docker. The following command will download this and store it in the temp location.

```
Invoke-WebRequest "https://test.docker.com/builds/Windows/x86_64/docker-1.13.0-rc4.zip" -OutFile "$env:TEMP\docker-1.13.0-rc4.zip" -UseBasicParsing
```

```

Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\WINDOWS\system32> Invoke-WebRequest "https://test.docker.com/builds/Windows/x86_64/docker-1.13.0-rc4.zip" -OutFile $env:TEMP\docker-1.13.0-rc4.zip -UseBasicParsing
PS C:\WINDOWS\system32>

```

Step 4: Next, you need to expand the archive using the following **powershell** command.

```
Expand-Archive -Path "$env:TEMP\docker-1.13.0-rc4.zip" -DestinationPath $env:ProgramFiles
```

```

Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\WINDOWS\system32> Invoke-WebRequest "https://test.docker.com/builds/Windows/x86_64/docker-1.13.0-rc4.zip" -OutFile $env:TEMP\docker-1.13.0-rc4.zip -UseBasicParsing
PS C:\WINDOWS\system32> Expand-Archive -Path "$env:TEMP\docker-1.13.0-rc4.zip" -DestinationPath $env:ProgramFiles
PS C:\WINDOWS\system32>

```

Step 5: Next, you need to add the Docker Files to the environment variable using the following **powershell** command.

```
$env:path += ";$env:ProgramFiles\Docker"
```

Step 6: Next, you need to register the Docker Daemon Service using the following **powershell** command.

```
dockerd --register-service
```

Step 7: Finally, you can start the **docker daemon** using the following command.

```
Start-Service Docker
```

Use the **docker version** command in **powershell** to verify that the **docker daemon** is working.

```

Administrator: Windows PowerShell
PS C:\WINDOWS\system32> docker version
Client:
Version: 1.13.0-rc4
API version: 1.25
Go version: go1.7.3
Git commit: 88862e7
Built: Sat Dec 17 01:34:17 2016
OS/Arch: windows/amd64

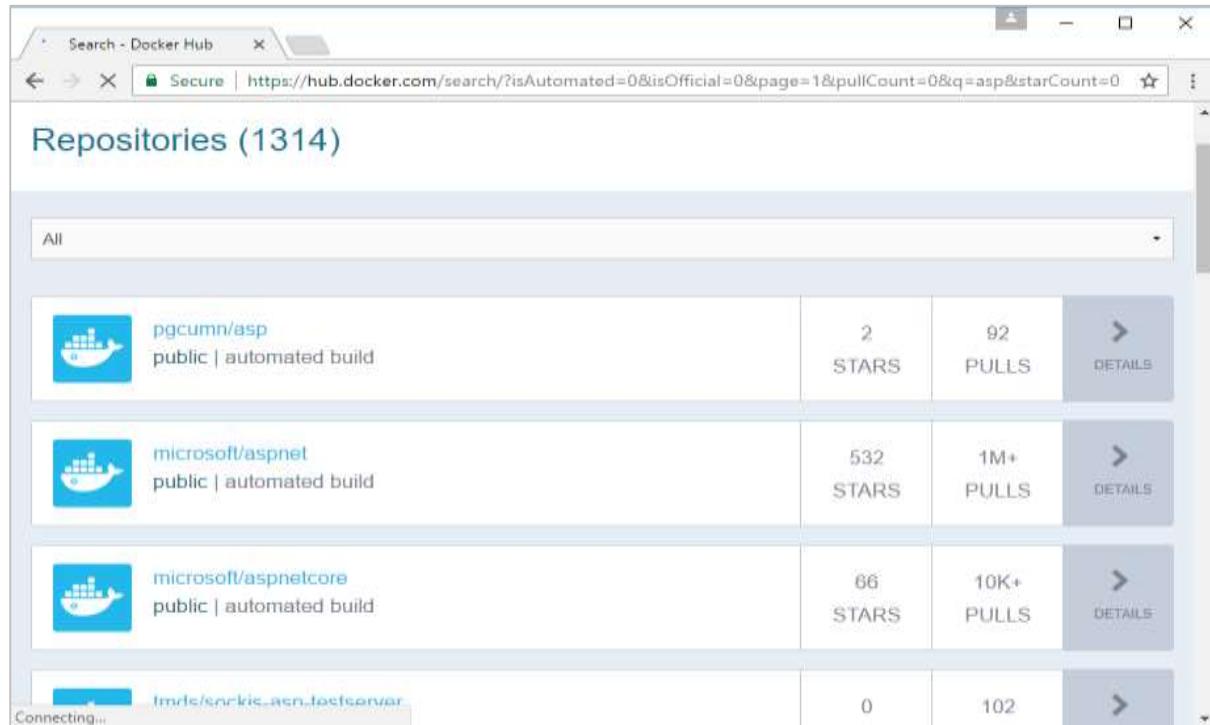
Server:
Version: 1.13.0-rc4
API version: 1.25 (minimum version 1.24)
Go version: go1.7.3
Git commit: 88862e7
Built: Sat Dec 17 01:34:17 2016
OS/Arch: windows/amd64
Experimental: false
PS C:\WINDOWS\system32>

```

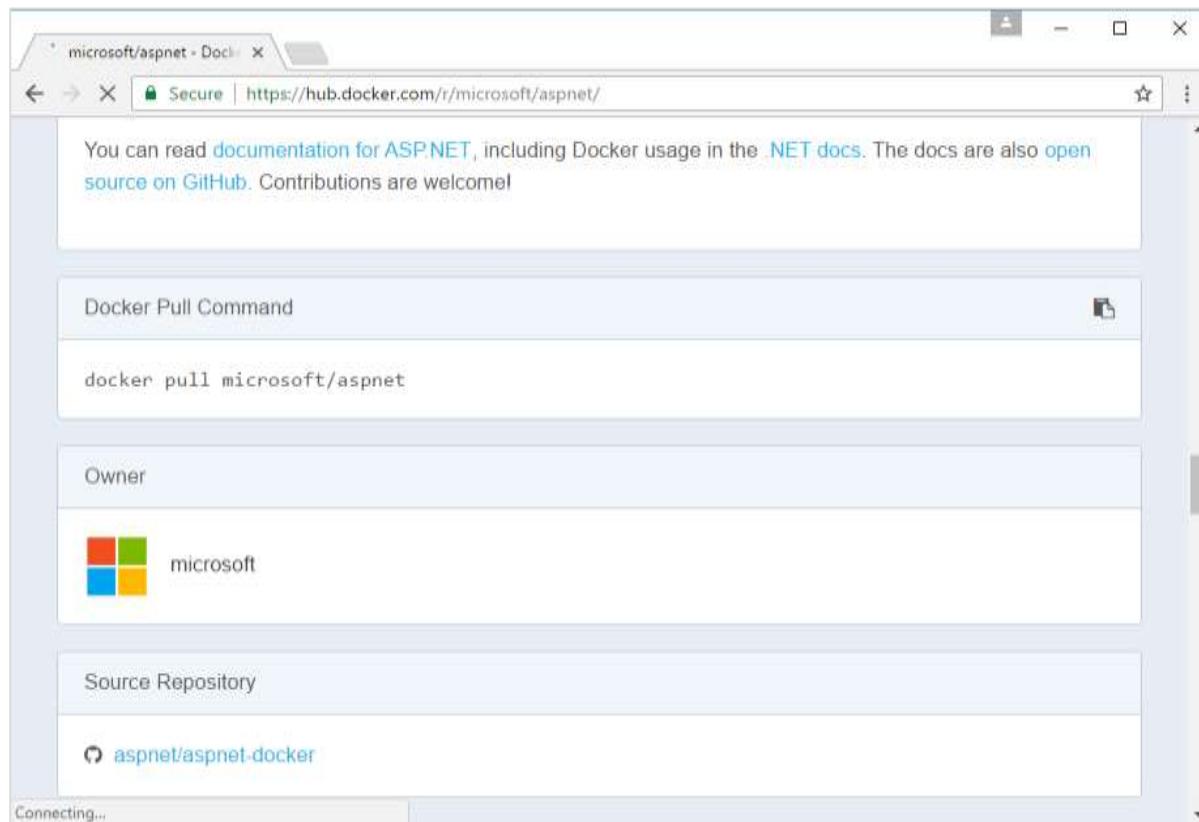
Installing the ASP.Net Container

Let's see how to install the ASP.Net container.

Step 1: The first step is to pull the image from Docker Hub. When you log into Docker Hub, you will be able to search and see the image for **Microsoft/aspnet** as shown below. Just type in **asp** in the search box and click on the Microsoft/aspnet link which comes up in the search results.



Step 2: You will see that the Docker **pull** command for ASP.Net in the details of the repository in Docker Hub.



Step 3: Go to Docker Host and run the Docker **pull** command for the microsoft/aspnet image. Note that the image is pretty large, somewhere close to 4.2 GB.

```
Administrator: Windows PowerShell
PS C:\WINDOWS\system32> docker pull microsoft/aspnet
Using default tag: latest
latest: Pulling from microsoft/aspnet
3889bb8d808b: Pull complete
04ee5d718c7a: Pull complete
c0931dd15237: Pull complete
61784b745c20: Pull complete
d05122f129ca: Pull complete
c31417523e9: Pull complete
Digest: sha256:48e6b45a43bc58e71da3d20d73839c29cc1c7ce87ed06ad257c4150e5286ade3
Status: Downloaded newer image for microsoft/aspnet:latest
PS C:\WINDOWS\system32>
```

Step 4: Now go to the following location <https://github.com/Microsoft/aspnet-docker> and download the entire **Git** repository.

Step 5: Create a folder called **App** in your C drive. Then copy the contents from the **4.6.2/sample** folder to your C drive. Go the Docker File in the sample directory and issue the following command:

```
docker build -t aspnet-site-new -build-arg site_root=
```

The following points need to be noted about the above command:

- It builds a new image called **aspnet-site-new** from the Docker File.
- The root path is set to the localpath folder.

```

Administrator: Windows PowerShell
-a--- 11/3/2016 11:54 AM          72 Dockerfile

PS C:\App> cd 4.6.2
PS C:\App\4.6.2> dir

Directory: C:\App\4.6.2

Mode                LastWriteTime      Length Name
----                -              -           -
d----       1/15/2017   8:52 AM          sample
-a---       11/3/2016  11:54 AM         216 Dockerfile

PS C:\App\4.6.2> cd sample
PS C:\App\4.6.2\sample> dir

Directory: C:\App\4.6.2\sample

Mode                LastWriteTime      Length Name
----                -              -           -
-a---       11/3/2016  11:54 AM         72 Dockerfile

PS C:\App\4.6.2\sample> docker build -t aspnet-site-new --build-arg site_root=/
.
Sending build context to Docker daemon 2.048 kB
Step 1/3 : FROM microsoft/aspnet
--> 08897a3b116a
Step 2/3 : ARG site_root=.
--> Using cache
--> 4c89cb1bef2d
Step 3/3 : ADD ${site_root} /inetpub/wwwroot
--> d1ce3a0ca1bc
Removing intermediate container 8652c844e52a
Successfully built d1ce3a0ca1bc
PS C:\App\4.6.2\sample>

```

Step 6: Now it's time to run the container. It can be done using the following command:

```
docker run -d -p 8000:80 -name my-running-site-new aspnet-site-new
```

```

Administrator: Windows PowerShell
PS C:\App\4.6.2\sample> docker run -d -p 8000:80 --name my-running-site-new aspnet-site-new
31cfe55892ba5059330ae5084532cc3cb8a31a64b69d9478a899c33f1c2ec629
PS C:\App\4.6.2\sample>

```

Step 7: You will now have IIS running in the Docker container. To find the IP Address of the Docker container, you can issue the Docker **inspect** command as shown below.

```

Administrator: Windows PowerShell
PS C:\App\4.6.2\sample> docker run -d -p 8000:80 --name my-running-site-new aspnet-site-new
31cfe55892ba5059330ae5084532cc3cb8a31a64b69d9478a899c33f1c2ec629
PS C:\App\4.6.2\sample> docker inspect -f "{{ .NetworkSettings.Networks.nat.IPAddress }}" my-running-site-new
172.20.130.68
PS C:\App\4.6.2\sample>

```

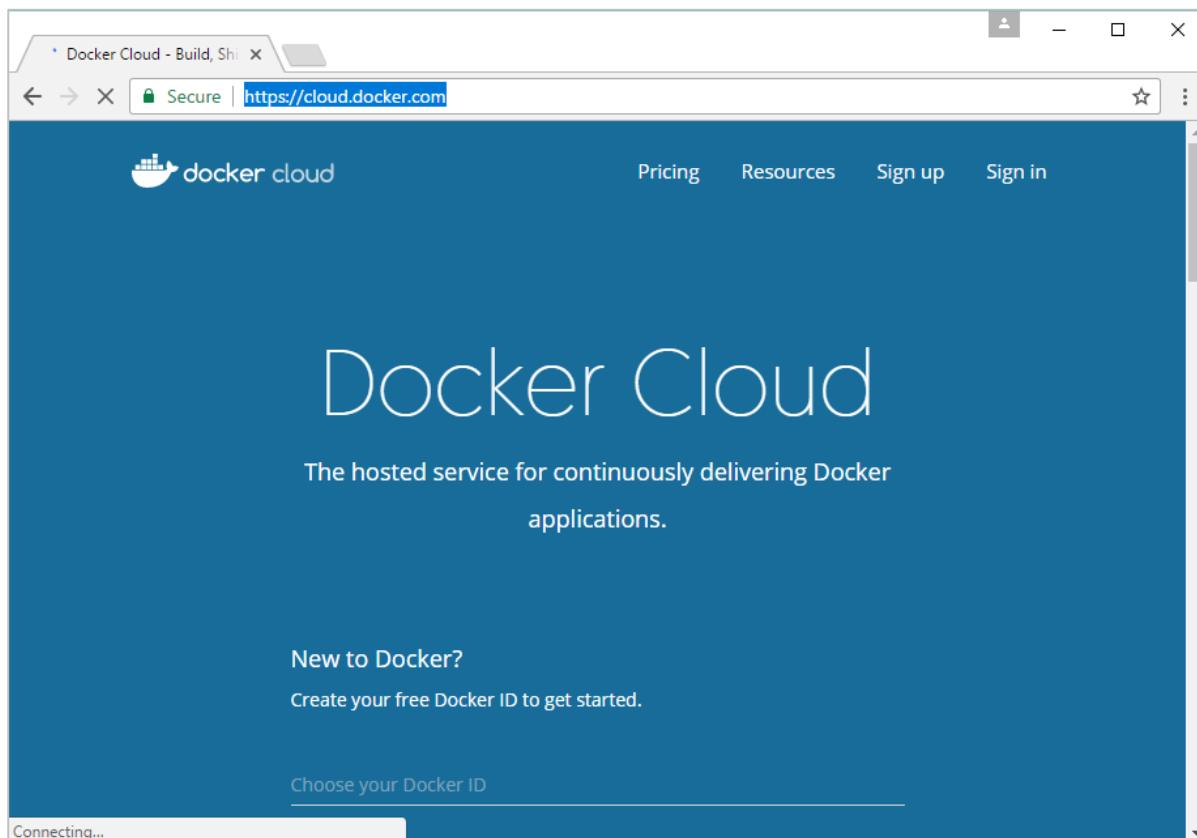
27. Docker – Docker Cloud

The Docker Cloud is a service provided by Docker in which you can carry out the following operations:

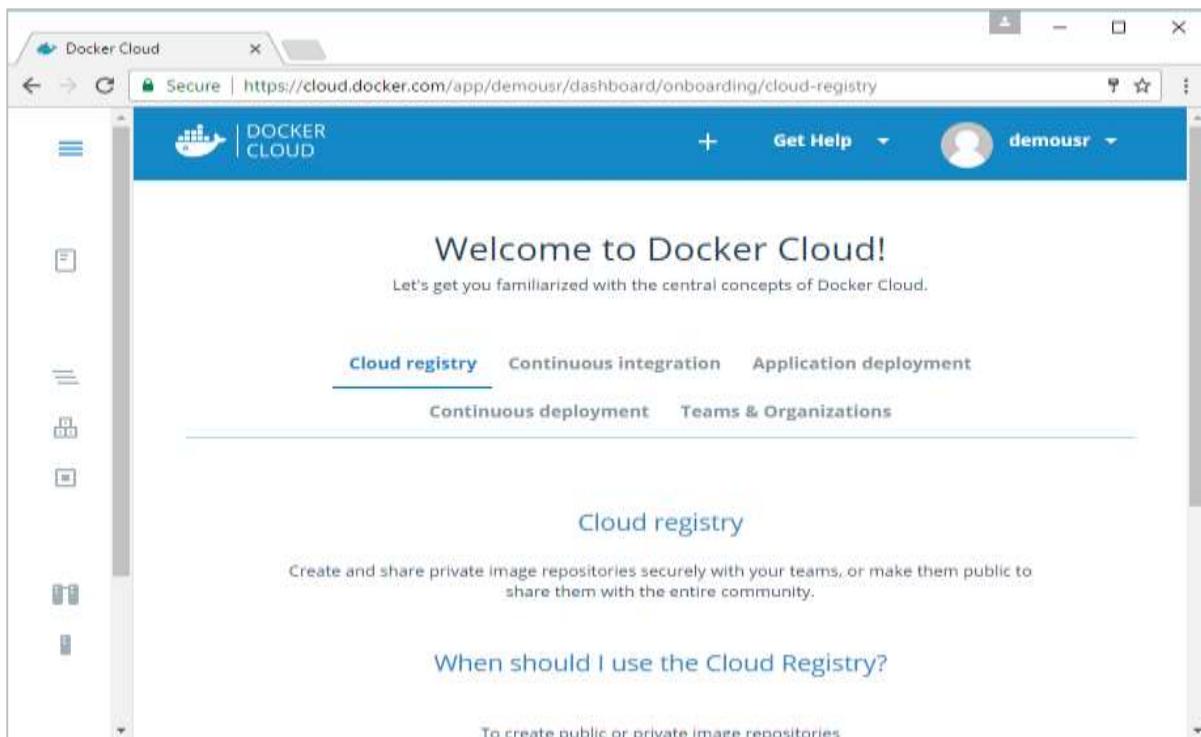
- **Nodes** – You can connect the Docker Cloud to your existing cloud providers such as Azure and AWS to spin up containers on these environments.
- **Cloud Repository** – Provides a place where you can store your own repositories.
- **Continuous Integration** – Connect with **Github** and build a continuous integration pipeline.
- **Application Deployment** – Deploy and scale infrastructure and containers.
- **Continuous Deployment** – Can automate deployments.

Getting started

You can go to the following link to getting started with Docker Cloud:
<https://cloud.docker.com/>



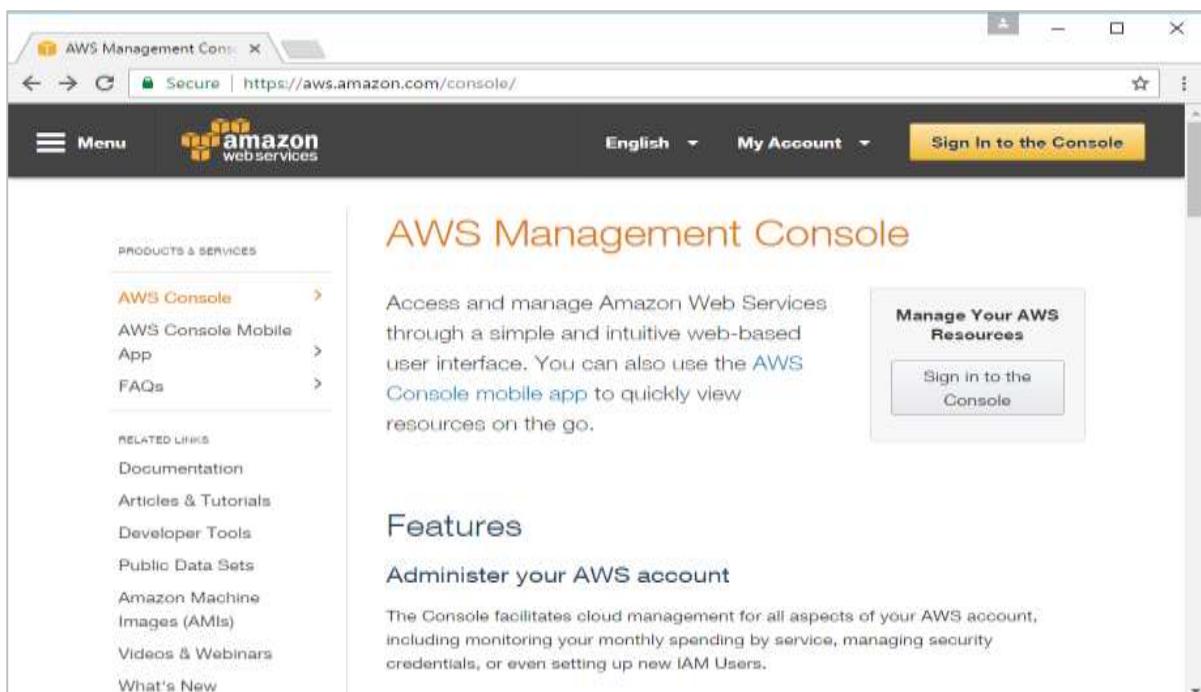
Once logged in, you will be provided with the following basic interface:



Connecting to the Cloud Provider

The first step is to connect to an existing cloud provider. The following steps will show you how to connect with an Amazon Cloud provider.

Step 1: The first step is to ensure that you have the right AWS keys. This can be taken from the **aws** console. Log into your **aws** account using the following link - <https://aws.amazon.com/console/>



Step 2: Once logged in, go to the Security Credentials section. Make a note of the access keys which will be used from Docker Hub.

Created	Deleted	Access Key ID	Last Used	Last Used Region	Last Used Service	Status	Actions
Jan 7th 2017		AKIAJMK4PHE5JUMBY7JQ	N/A	N/A	N/A	Active	Make Inactive Delete
Dec 13th 2016		AKIAJ5G7VMJ6TSKJOGKA	N/A	N/A	N/A	Active	Make Inactive Delete
Dec 12th 2016	Jan 7th 2017	AKIAIRAK6UZ7VQPALXA	2016-12-12 03:04 PST	us-west-2	ec2	Deleted	
Jul 27th 2016	Dec 13th 2016	AKIAIR37RFGT07BPMF3A	N/A	N/A	N/A	Deleted	

Create New Access Key

Important Change - Managing Your AWS Secret Access Keys

As described in a previous announcement, you cannot retrieve the existing secret access keys for your AWS root account, though you can still create a new root access key at any time. As a best practice, we recommend creating an IAM user that has access keys rather than relying on root access keys.

Step 3: Next, you need to create a policy in **aws** that will allow Docker to view EC2 instances. Go to the profiles section in **aws**. Click the **Create Policy** button.

	Policy Name	Attached Entities	Creation Time	Edited Time
<input type="checkbox"/>	AmazonS3FullAccess	1	2015-02-06 10:40 PST	2015-02-06 10:40...
<input type="checkbox"/>	Cloudwatchdemo	1	2016-12-13 08:43 PST	2016-12-13 08:43...
<input type="checkbox"/>	AdministratorAccess	0	2015-02-06 10:39 PST	2015-02-06 10:39...
<input type="checkbox"/>	AmazonAPIGateway...	0	2015-07-09 10:34 PST	2015-07-09 10:34...
<input type="checkbox"/>	AmazonAPIGateway...	0	2015-07-09 10:36 PST	2015-07-09 10:36...
<input type="checkbox"/>	AmazonAPIGateway...	0	2015-11-11 15:41 PST	2015-11-11 15:41...
<input type="checkbox"/>	AmazonAppStreamF...	0	2015-02-06 10:40 PST	2015-02-06 10:40...
<input type="checkbox"/>	AmazonAppStreamR...	0	2015-02-06 10:40 PST	2016-12-07 13:00...
<input type="checkbox"/>	AmazonAppStreamS...	0	2016-11-18 20:17 PST	2016-11-18 20:17...
<input type="checkbox"/>	AmazonAthenaFullA...	0	2016-11-30 08:46 PST	2016-11-30 08:46...
<input type="checkbox"/>	AmazonCognitoDev	0	2015-03-24 10:22 PST	2015-03-24 10:22...

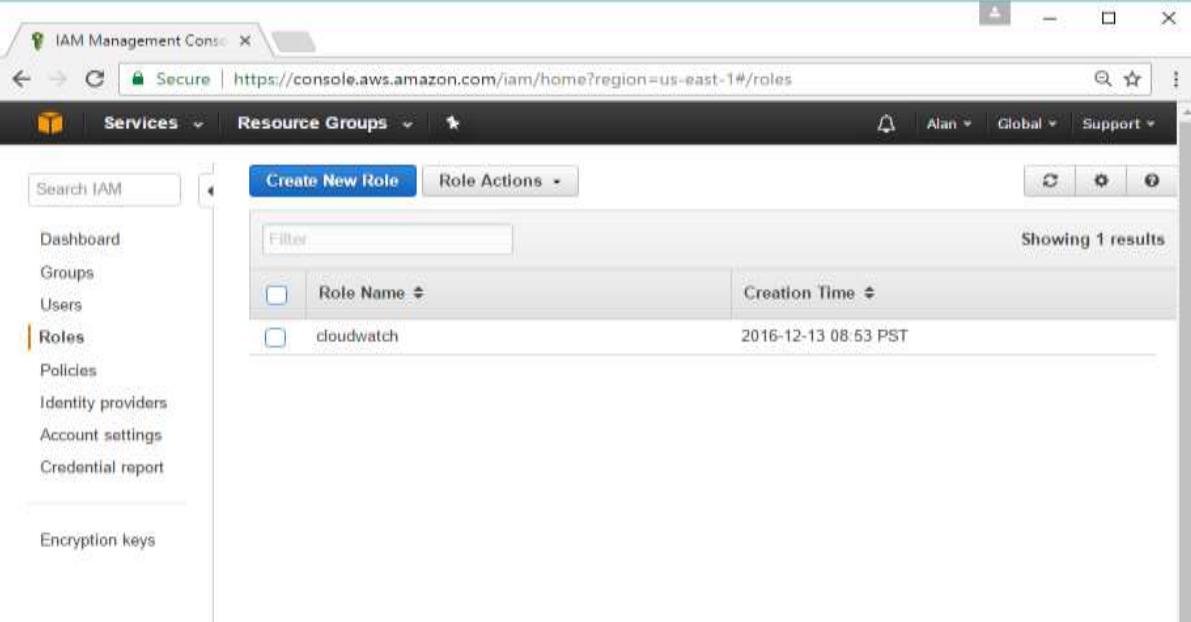
Step 4: Click on 'Create Your Own Policy' and give the policy name as **dockercloud-policy** and the policy definition as shown below.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "ec2:*",
        "iam>ListInstanceProfiles"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

The screenshot shows the AWS IAM Management Console with the URL <https://console.aws.amazon.com/iam/home?region=us-east-1#/policies>. The interface is in dark mode. On the left, there's a sidebar with 'Create Policy' options: 'Step 1 : Create Policy', 'Step 2 : Set Permissions', and 'Step 3 : Review Policy'. The 'Review Policy' section is active. It contains a 'Policy Name' input field with 'dockercloud-policy', a 'Description' text area, and a 'Policy Document' code editor. The code in the editor is identical to the one provided at the top of the page. At the bottom right of the 'Review Policy' section, there are buttons for 'Cancel', 'Validate Policy', 'Previous', and a prominent blue 'Create Policy' button.

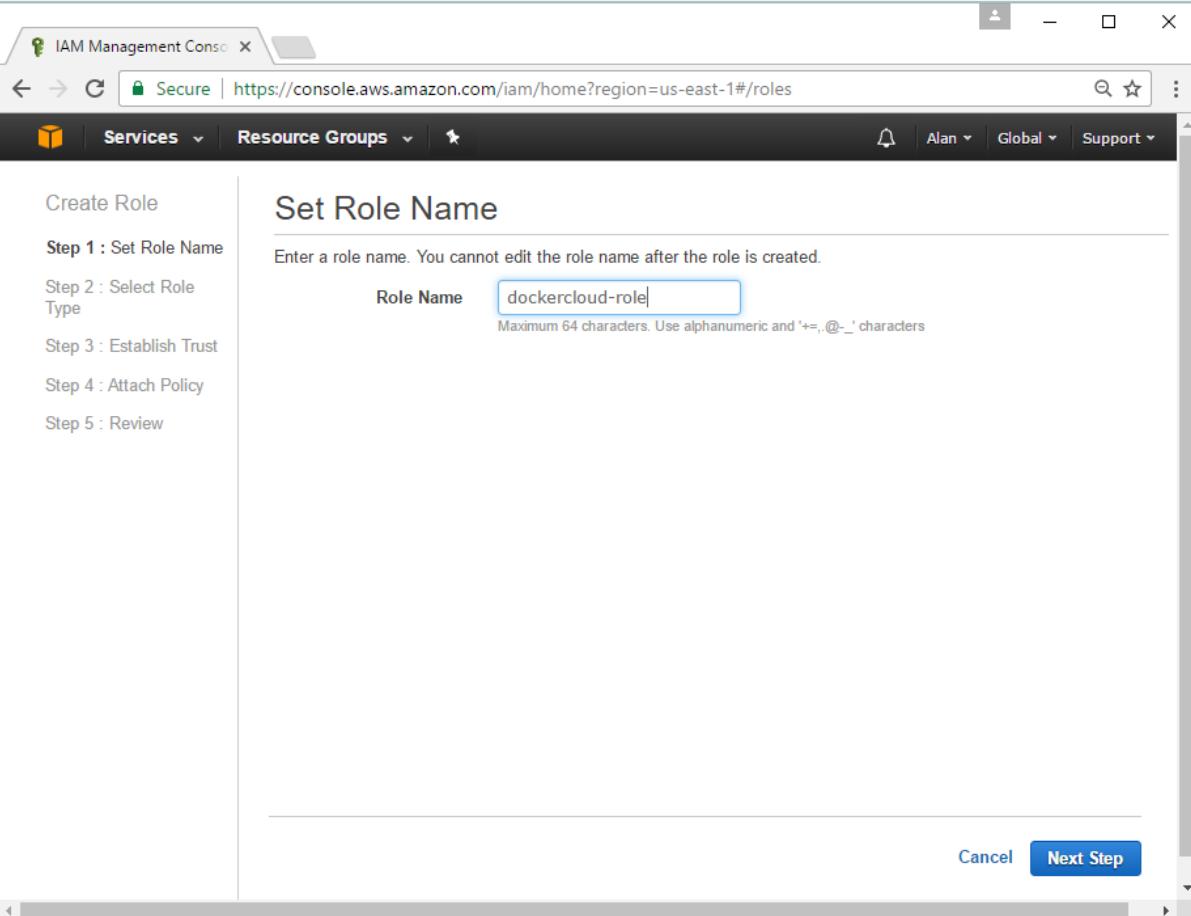
Next, click the **Create Policy** button.

Step 5: Next, you need to create a **role** which will be used by Docker to spin up nodes on AWS. For this, go to the **Roles** section in AWS and click the **Create New Role** option.



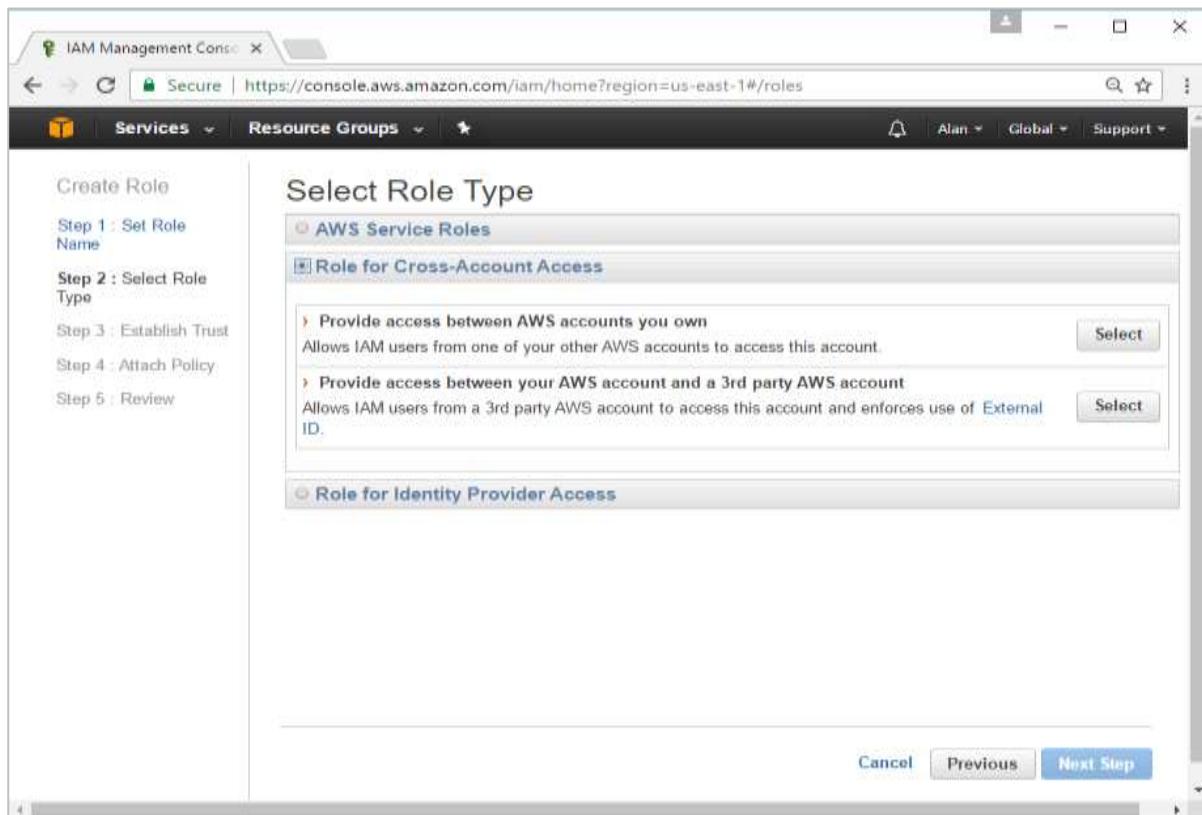
The screenshot shows the AWS IAM Management Console. The left sidebar has 'Roles' selected. The main area displays a table with one result: 'cloudwatch' created on '2016-12-13 08:53 PST'. There are 'Create New Role' and 'Role Actions' buttons at the top.

Step 6: Give the name for the role as **dockercloud-role**.



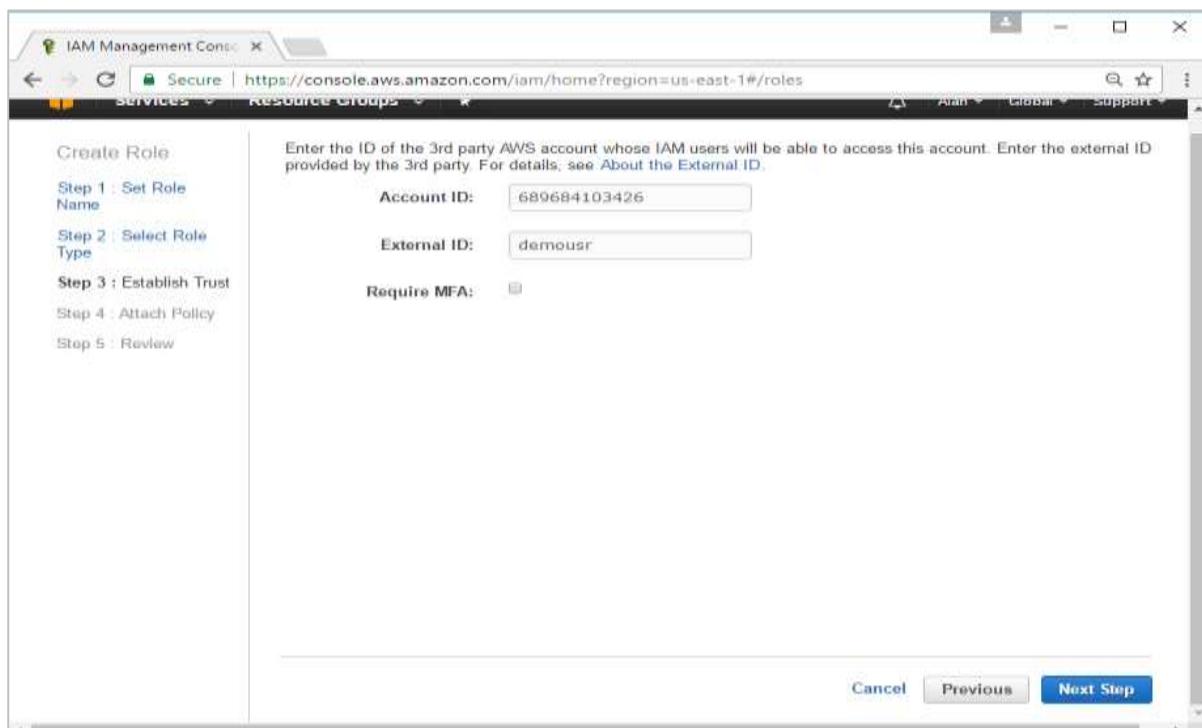
The screenshot shows the 'Set Role Name' step of the 'Create Role' wizard. On the left, a sidebar lists steps: Step 1: Set Role Name (highlighted), Step 2: Select Role Type, Step 3: Establish Trust, Step 4: Attach Policy, Step 5: Review. The main area has a 'Role Name' input field containing 'dockercloud-role'. A note says 'Enter a role name. You cannot edit the role name after the role is created.' Below it, a note says 'Maximum 64 characters. Use alphanumeric and '+-=,@-_ characters'. At the bottom right are 'Cancel' and 'Next Step' buttons.

Step 7: On the next screen, go to 'Role for Cross Account Access' and select "Provide access between your account and a 3rd party AWS account"



Step 8: On the next screen, enter the following details:

- In the Account ID field, enter the ID for the Docker Cloud service: 689684103426.
- In the External ID field, enter your Docker Cloud username.



Step 9: Then, click the **Next Step** button and on the next screen, attach the policy which was created in the earlier step.

The screenshot shows the 'Attach Policy' step of the IAM role creation wizard. On the left, a sidebar lists steps from 'Step 1: Set Role Name' to 'Step 5: Review'. The main area is titled 'Attach Policy' with the instruction 'Select one or more policies to attach. Each role can have up to 10 policies attached.' A table titled 'Showing 2 results' displays two policies. The first, 'AWSElasticBeansta...', has an unchecked checkbox. The second, 'dockercloud-policy', has a checked checkbox. The table columns are 'Policy Name', 'Attached Entities', 'Creation Time', and 'Edited Time'. At the bottom right are 'Cancel', 'Previous', and 'Next Step' buttons, with 'Next Step' being highlighted in blue.

Step 10: Finally, on the last screen when the role is created, make sure to copy the **arn** role which is created.

arn:aws:iam::085363624145:role/dockercloud-role

The screenshot shows the 'Review' step of the IAM role creation wizard. The left sidebar shows steps from 'Step 1: Set Role Name' to 'Step 5: Review'. The main area is titled 'Review' and contains the following information:

- Role Name:** dockercloud-role (with a 'Edit Role Name' link)
- Role ARN:** arn:aws:iam::085363624145:role/dockercloud-role
- Trusted Entities:** The account 689684103426
- Policies:** arn:aws:iam::085363624145:policy/dockercloud-policy (with a 'Change Policies' link)
- Give this link to users who can switch roles in the console:** https://signin.aws.amazon.com/switchrole?account=085363624145&roleName=dockercloud-role (with a 'Copy Link' button)

At the bottom right are 'Cancel', 'Previous', and 'Create Role' buttons, with 'Create Role' being highlighted in blue.

Step 11: Now go back to **Docker Cloud**, select **Cloud Providers**, and click the **plug symbol** next to Amazon Web Services.

The screenshot shows the Docker Cloud interface with the title "Cloud Settings". On the left, there's a sidebar with icons for General, Cloud providers, Source providers, Notifications, Default Privacy, Billing, Plan, and Quotas. The "Cloud providers" option is selected. In the main area, there's a profile picture for "demouser" (Member since Dec 17, 2016). Below it is a table titled "Cloud providers" listing five providers: Amazon Web Services, Digital Ocean, Microsoft Azure, SoftLayer, and Packet. Each provider has a "Add new credentials" button and a status indicator (e.g., "Free Tier", "\$20 Code", "Free trial").

Enter the **arn** role and click the **Save** button.

The screenshot shows the Docker Cloud interface with the title "Cloud Settings". A modal dialog box is open, titled "Add AWS Credentials". It contains a message: "Don't know where or how to get this? Click [here](#) to learn more." Below this is a text input field containing the ARN: "arn:aws:iam::085363624145:role/dockercloud-role". At the bottom of the dialog are two buttons: "Cancel" (red) and "Save" (blue). The background shows the "Cloud providers" table from the previous screenshot.

Once saved, the integration with AWS would be complete.

The screenshot shows the Docker Cloud interface under the 'Cloud Settings' section. On the left, there's a sidebar with icons for General, Cloud providers, Source providers, Notifications, Default Privacy, Billing, Plan, and Quotas. The main area displays a user profile for 'demousr' (Member since Dec 17, 2016) and a list of 'Cloud providers'. The providers listed are Amazon Web Services, Digital Ocean, Microsoft Azure, SoftLayer, and Packet. Each provider entry includes a role ARN, a status icon, and a 'Free Tier' badge.

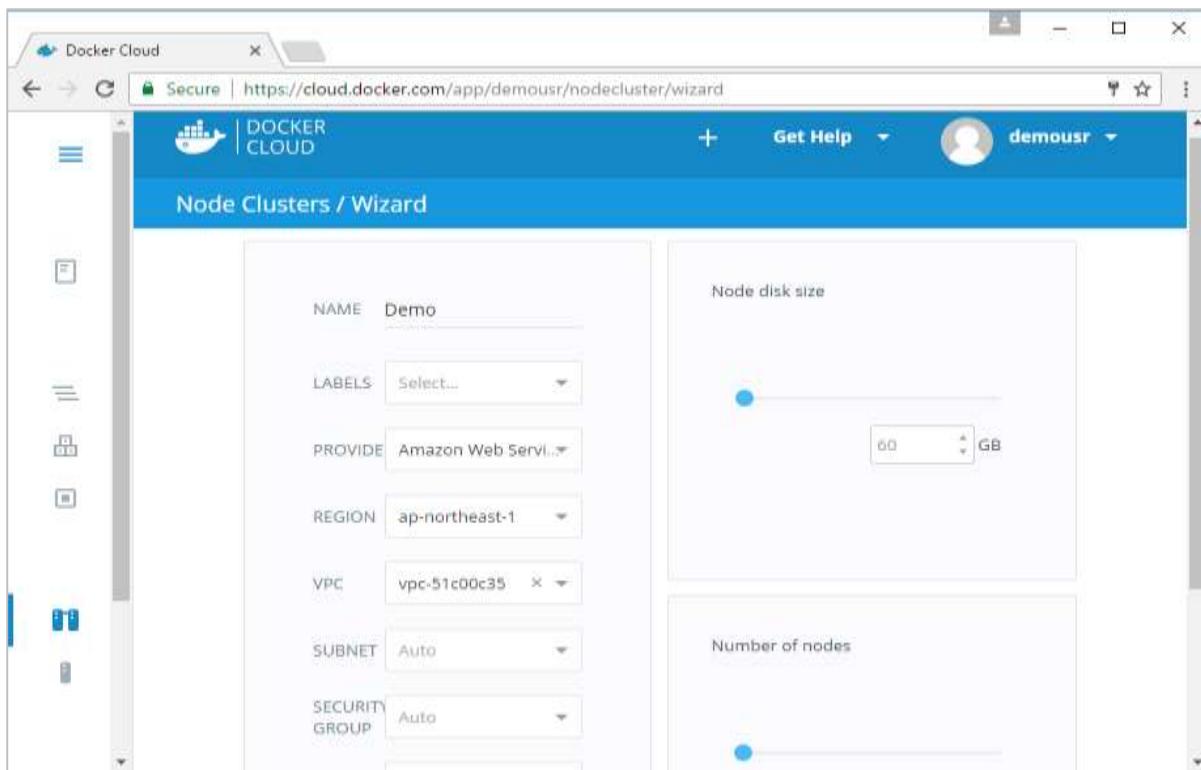
Setting Up Nodes

Once the integration with AWS is complete, the next step is to setup a node. Go to the Nodes section in Docker Cloud. Note that the setting up of nodes will automatically setup a node cluster first.

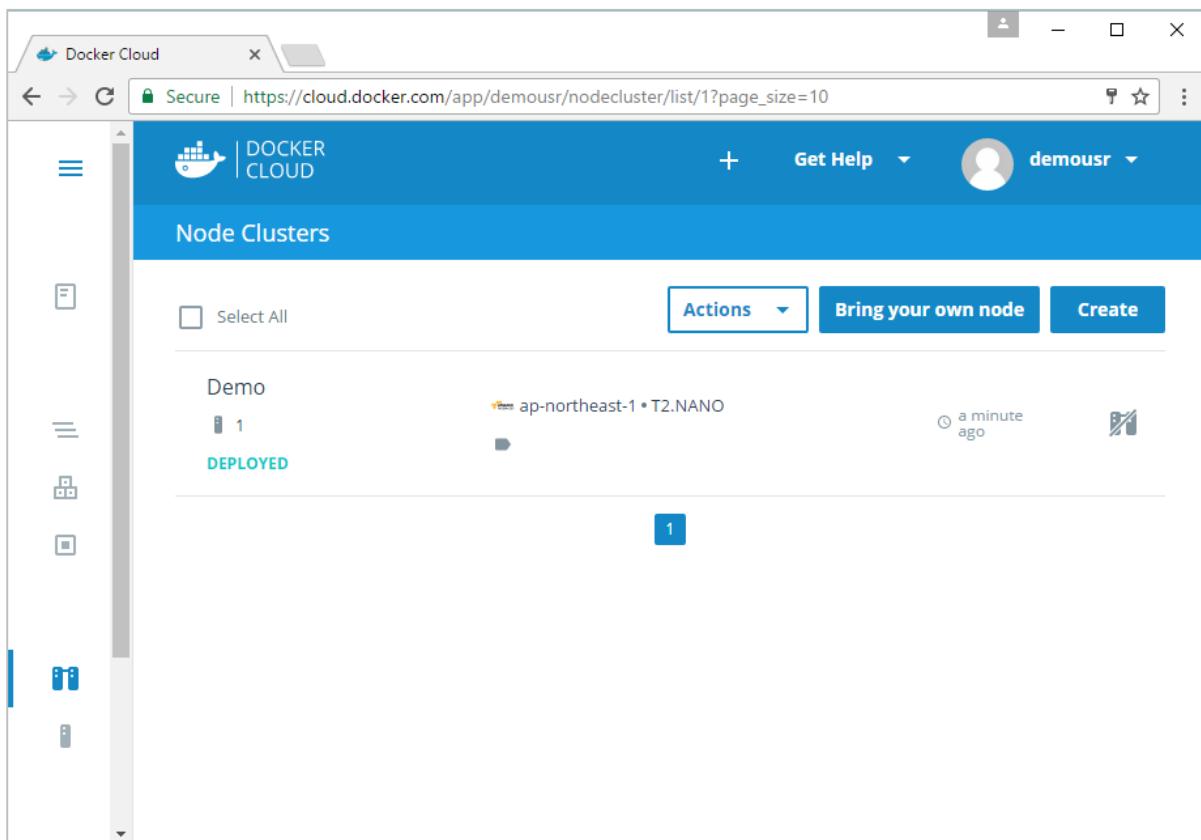
Step 1: Go to the Nodes section in Docker Cloud.

The screenshot shows the Docker Cloud interface with the 'Node Clusters' section selected in the sidebar. The main area is titled 'wizard' and contains fields for 'Node cluster name' (set to 'DE'), 'Node disk size' (set to 60 GB), and 'Number of nodes' (set to 1). There are dropdown menus for 'Select...' and 'Amazon Web Serv...'. The sidebar also lists Stacks, Services, Containers, and Cloud Settings.

Step 2: Next, you can give the details of the nodes which will be setup in AWS.



You can then click the Launch Node cluster which will be present at the bottom of the screen. Once the node is deployed, you will get the notification in the Node Cluster screen.



Deploying a Service

The next step after deploying a node is to deploy a service. To do this, we need to perform the following steps.

Step 1: Go to the **Services Section** in Docker Cloud. Click the **Create** button.

Docker Cloud

Secure | https://cloud.docker.com/app/demousr/service/list/1?page_size=10

BUILD

APPLICATIONS

- Repositories
- Stacks
- Services**
- Containers

INFRASTRUCTURE

- Node Clusters
- Nodes

SETTINGS

Actions Create

Step 2: Choose the Service which is required. In our case, let's choose **mongo**.

CACHE SERVERS

- redis
- memcached

ANALYTICS

- elasticsearch

MISCELLANEOUS

- dockercloud/hello-world
- dockercloud/authorizedkeys

APPLICATION SERVERS

- tomcat
- glassfish

DATABASE SERVERS

- mongo
- postgres
- mariadb

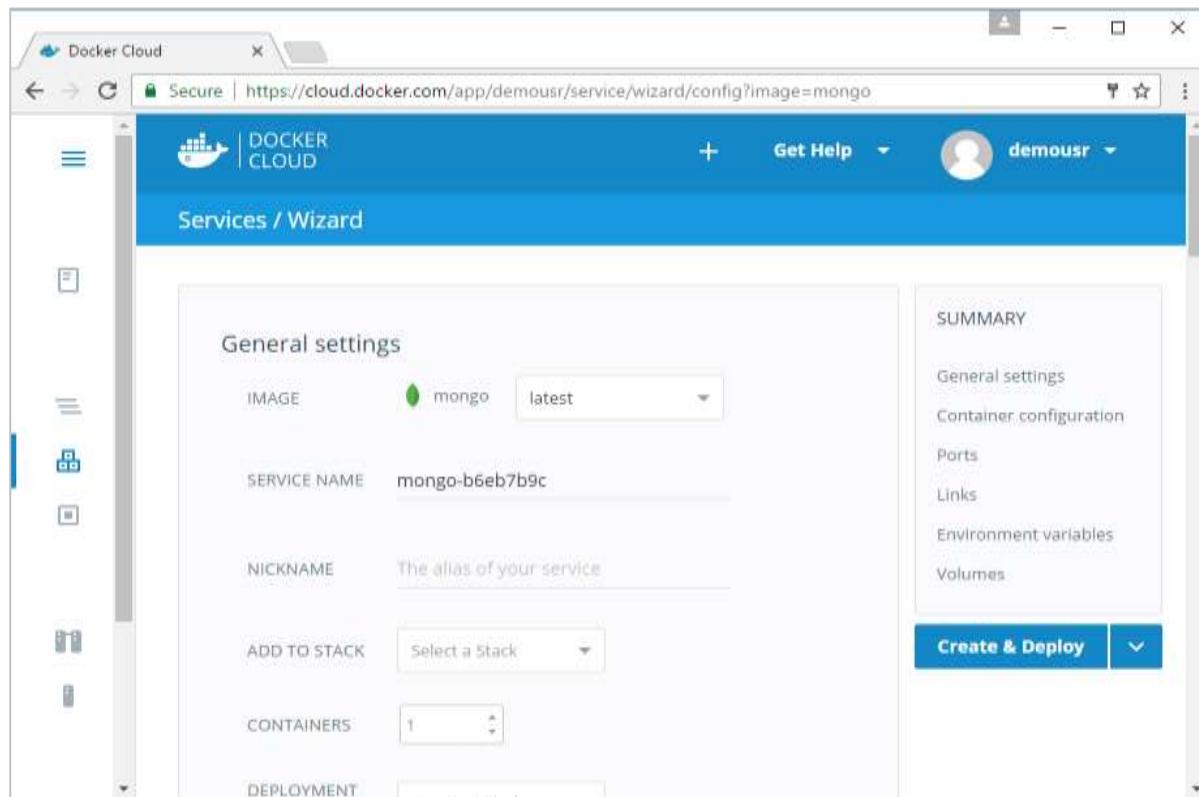
CMS

- wordpress

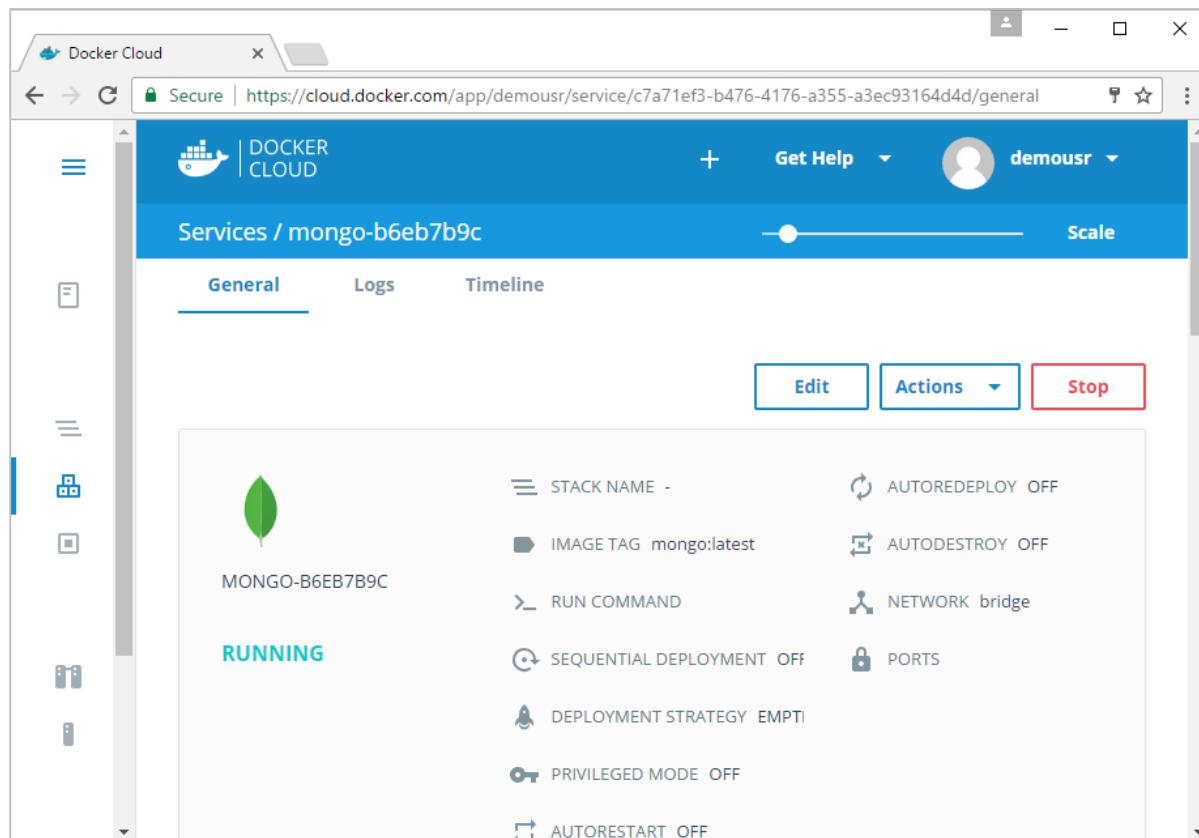
MONITORING

- datadog/docker-dd-agent

Step 3: On the next screen, choose the **Create & Deploy** option. This will start deploying the **Mongo** container on your node cluster.



Once deployed, you will be able to see the container in a running state.



28. Docker – Logging

Docker has logging mechanisms in place which can be used to debug issues as and when they occur. There is logging at the **daemon level** and at the **container level**. Let's look at the different levels of logging.

Daemon Logging

At the daemon logging level, there are four levels of logging available:

- **Debug** — It details all the possible information handled by the daemon process.
- **Info** — It details all the errors + Information handled by the daemon process.
- **Errors** — It details all the errors handled by the daemon process.
- **Fatal** — It only details all the fatal errors handled by the daemon process.

Go through the following steps to learn how to enable logging.

Step 1: First, we need to stop the **docker daemon process**, if it is already running. It can be done using the following command:

```
sudo service docker stop
```

```
demo@ubuntudemo:~$ sudo service docker stop
```

Step 2: Now we need to start the **docker daemon process**. But this time, we need to append the **-l** parameter to specify the logging option. So let's issue the following command when starting the **docker daemon process**.

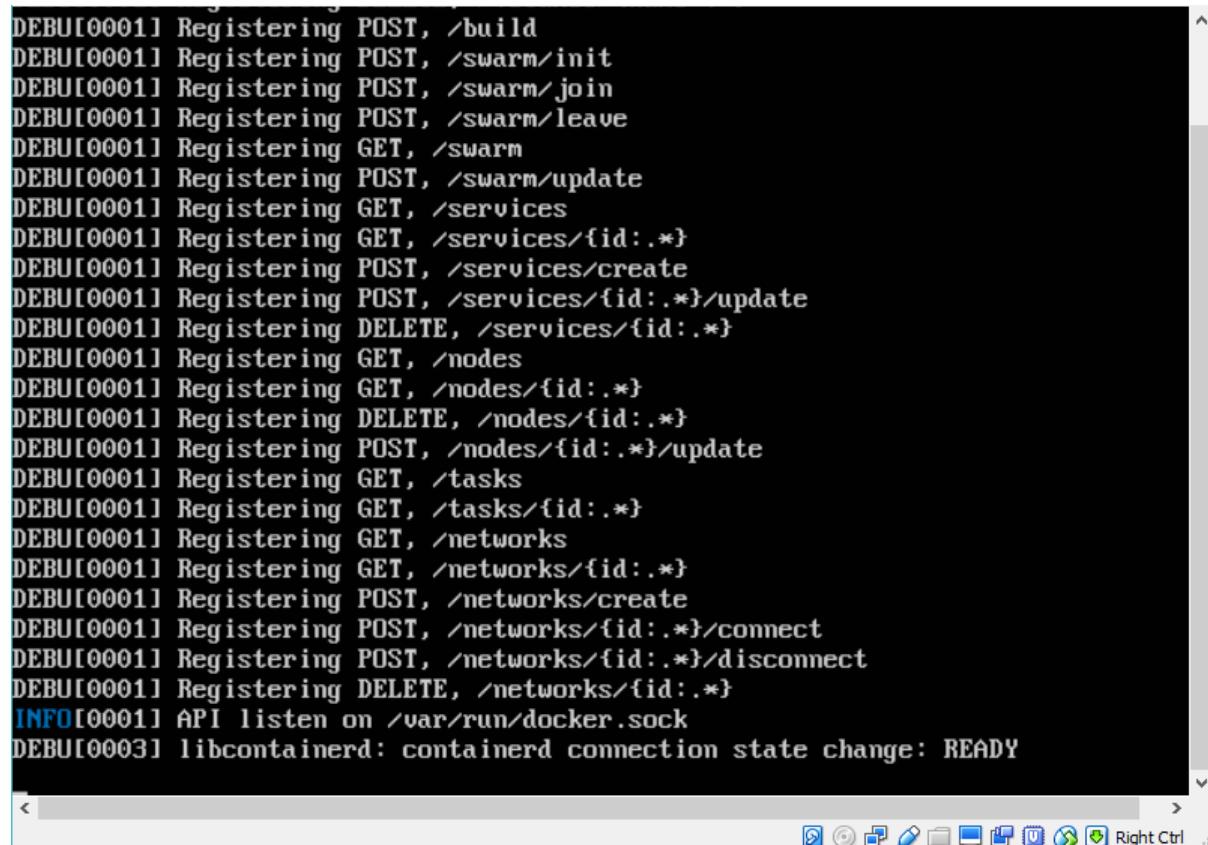
```
sudo dockerd -l debug &
```

The following points need to be noted about the above command:

- **dockerd** is the executable for the **docker daemon process**.
- The **-l** option is used to specify the logging level. In our case, we are putting this as debug.
- **&** is used to come back to the command prompt after the logging has been enabled.

```
demo@ubuntudemo:~$ sudo dockerd -l debug &
```

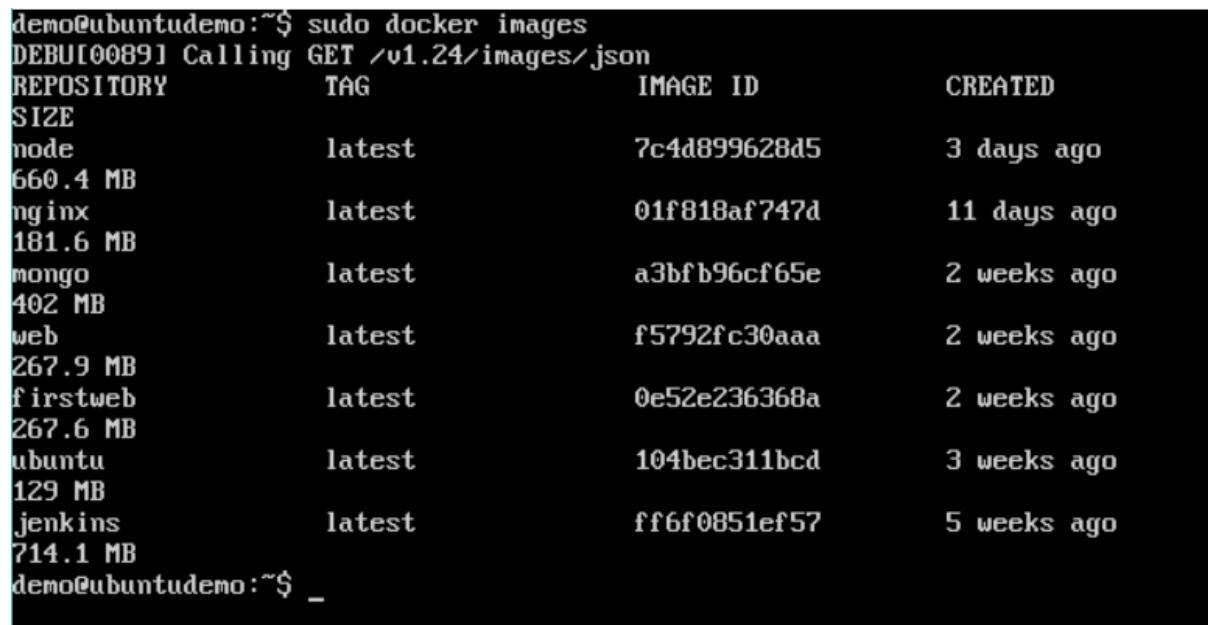
Once you start the Docker process with logging, you will also now see the **Debug Logs** being sent to the console.



```

DEBU[0001] Registering POST, /build
DEBU[0001] Registering POST, /swarm/init
DEBU[0001] Registering POST, /swarm/join
DEBU[0001] Registering POST, /swarm/leave
DEBU[0001] Registering GET, /swarm
DEBU[0001] Registering POST, /swarm/update
DEBU[0001] Registering GET, /services
DEBU[0001] Registering GET, /services/{id:.*}
DEBU[0001] Registering POST, /services/create
DEBU[0001] Registering POST, /services/{id:.*}/update
DEBU[0001] Registering DELETE, /services/{id:.*}
DEBU[0001] Registering GET, /nodes
DEBU[0001] Registering GET, /nodes/{id:.*}
DEBU[0001] Registering DELETE, /nodes/{id:.*}
DEBU[0001] Registering POST, /nodes/{id:.*}/update
DEBU[0001] Registering GET, /tasks
DEBU[0001] Registering GET, /tasks/{id:.*}
DEBU[0001] Registering GET, /networks
DEBU[0001] Registering GET, /networks/{id:.*}
DEBU[0001] Registering POST, /networks/create
DEBU[0001] Registering POST, /networks/{id:.*}/connect
DEBU[0001] Registering POST, /networks/{id:.*}/disconnect
DEBU[0001] Registering DELETE, /networks/{id:.*}
INFO[0001] API listen on /var/run/docker.sock
DEBU[0003] libcontainerd: containerd connection state change: READY
  
```

Now, if you execute any Docker command such as **docker images**, the Debug information will also be sent to the console.



REPOSITORY	TAG	IMAGE ID	CREATED
node	latest	7c4d899628d5	3 days ago
nginx	latest	01f818af747d	11 days ago
mongo	latest	a3bfb96cf65e	2 weeks ago
web	latest	f5792fc30aaa	2 weeks ago
firstweb	latest	0e52e236368a	2 weeks ago
ubuntu	latest	104bec311bcd	3 weeks ago
jenkins	latest	ff6f0851ef57	5 weeks ago

Container Logging

Logging is also available at the container level. So in our example, let's spin up an Ubuntu container first. We can do it by using the following command.

```
sudo docker run -it ubuntu /bin/bash
```

```
demo@ubuntudemo:~$ sudo docker run -it ubuntu /bin/bash
root@6bf1271fcdd:/# demo@ubuntudemo:~$
```

Now, we can use the **docker log command** to see the logs of the container.

Syntax

```
Docker logs containerID
```

Parameters

- **containerID** – This is the ID of the container for which you need to see the logs.

Example

On our Docker Host, let's issue the following command. Before that, you can issue some commands whilst in the container.

```
sudo docker logs 6bf1271fcdd
```

Output

```
demo@ubuntudemo:~$ sudo docker logs 6bf1271fcdd
root@6bf1271fcdd:/#
root@6bf1271fcdd:/# ifconfig
bash: ifconfig: command not found
root@6bf1271fcdd:/# ls
bin  dev  home  lib64  mnt  proc  run  srv  tmp  var
boot  etc  lib  media  opt  root  sbin  sys  usr
demo@ubuntudemo:~$
```

From the output, you can see that the commands executed in the container are shown in the logs.

29. Docker – Docker Compose

Docker Compose is used to run multiple containers as a single service. For example, suppose you had an application which required NGINX and MySQL, you could create one file which would start both the containers as a service without the need to start each one separately.

In this chapter, we will see how to get started with Docker Compose. Then, we will look at how to get a simple service with MySQL and NGINX up and running using Docker Compose.

Docker Compose – Installation

The following steps need to be followed to get Docker Compose up and running.

Step 1: Download the necessary files from **github** using the following command:

```
curl -L "https://github.com/docker/compose/releases/download/1.10.0-rc2/docker-compose-$(uname -s)-$(uname -m)" -o /home/demo/docker-compose
```

The above command will download the latest version of Docker Compose which at the time of writing this article is **1.10.0-rc2**. It will then store it in the directory **/home/demo/**.

```
demo@ubuntudemo:~$ curl -L "https://github.com/docker/compose/releases/download/1.10.0-rc2/docker-compose-$(uname -s)-$(uname -m)" -o /home/demo/docker-compose
```

Step 2: Next, we need to provide **execute privileges** to the downloaded Docker Compose file, using the following command:

```
chmod +x /home/demo/docker-compose
```

```
demo@ubuntudemo:~$ curl -L "https://github.com/docker/compose/releases/download/1.10.0-rc2/docker-compose-$(uname -s)-$(uname -m)" -o /home/demo/docker-compose
% Total    % Received % Xferd  Average Speed   Time     Time      Current
          Dload  Upload Total   Spent    Left Speed
100  600    0  600    0     0   403      0 --:--:--  0:00:01 --:--:--  403
100 7929k  100 7929k    0     0   215k      0  0:00:36  0:00:36 --:--:--  213k
demo@ubuntudemo:~$ chmod +x /home/demo/docker-compose
```

We can then use the following command to see the **compose** version.

Syntax

```
docker-compose version
```

Parameters

- **version** – This is used to specify that we want the details of the version of **Docker Compose**.

Output

The version details of Docker Compose will be displayed.

Example

The following example shows how to get the **docker-compose** version.

```
sudo ./docker-compose -version
```

Output

You will then get the following output:

```
demo@ubuntudemo:~$ curl -L "https://github.com/docker/compose/releases/download/1.10.0-rc2/docker-compose-$(uname -s)-$(uname -m)" -o /home/demo/docker-compose
  % Total    % Received % Xferd  Average Speed   Time     Time     Time  Current
                                           Dload  Upload   Total   Spent    Left  Speed
100  600    0  600    0      0   403      0 --:--:--  0:00:01 --:--:--  403
100 7929k  100 7929k    0      0  215k      0  0:00:36  0:00:36 --:--:-- 213k
demo@ubuntudemo:~$ chmod +x /home/demo/docker-compose
demo@ubuntudemo:~$ sudo ./docker-compose version
docker-compose version 1.10.0-rc2, build fb241d0
docker-py version: 2.0.1
CPython version: 2.7.9
OpenSSL version: OpenSSL 1.0.1t  3 May 2016
demo@ubuntudemo:~$
```

Creating Your First Docker-Compose File

Now let's go ahead and create our first Docker Compose file. All Docker Compose files are YAML files. You can create one using the **vim** editor. So execute the following command to create the **compose** file:

```
sudo vim docker-compose.yml
```

```

version: '2'
services:
  databases:
    image: mysql
    ports:
    - "3306:3306"
    environment:
    - MYSQL_ROOT_PASSWORD=password
    - MYSQL_USER=user
    - MYSQL_PASSWORD=password
    - MYSQL_DATABASE=demodb
  web:
    image: nginx
  
```

Let's take a close look at the various details of this file:

- The **database** and **web** keyword are used to define two separate services. One will be running our **mysql** database and the other will be our **nginx** web server.
- The **image** keyword is used to specify the image from **dockerhub** for our **mysql** and **nginx** containers.
- For the database, we are using the **ports** keyword to mention the ports that need to be exposed for **mysql**.
- And then, we also specify the environment variables for **mysql** which are required to run **mysql**.

Now let's run our Docker Compose file using the following command:

```
sudo ./docker-compose up
```

This command will take the **docker-compose.yml** file in your local directory and start building the containers.

```
demo@ubuntudemo:~/ $ sudo ./docker-compose up
```

Once executed, all the images will start downloading and the containers will start automatically.

```
databases_1 | 2017-01-13T09:47:56.645996Z 0 [Note] Server socket created on T
'::'.
databases_1 | 2017-01-13T09:47:56.645558Z 0 [Note] InnoDB: Buffer pool(s) load
completed at 170113  9:47:56
databases_1 | 2017-01-13T09:47:56.684579Z 0 [Warning] 'user' entry 'root@loca
lost' ignored in --skip-name-resolve mode.
databases_1 | 2017-01-13T09:47:56.684730Z 0 [Warning] 'user' entry 'mysql.sys@l
ocalhost' ignored in --skip-name-resolve mode.
databases_1 | 2017-01-13T09:47:56.684809Z 0 [Warning] 'db' entry 'sys mysql.s
@localhost' ignored in --skip-name-resolve mode.
databases_1 | 2017-01-13T09:47:56.684849Z 0 [Warning] 'proxies_priv' entry 'r
oot@localhost' ignored in --skip-name-resolve mode.
databases_1 | 2017-01-13T09:47:56.688368Z 0 [Warning] 'tables_priv' entry 'sy
config mysql.sys@localhost' ignored in --skip-name-resolve mode.
databases_1 | 2017-01-13T09:47:56.703852Z 0 [Note] Event Scheduler: Loaded 0
events
databases_1 | 2017-01-13T09:47:56.704110Z 0 [Note] Executing 'SELECT * FROM I
FORMATION_SCHEMA.TABLES;' to get a list of tables using the deprecated partition
engine. You may use the startup option '--disable-partition-engine-check' to s
kip this check.
databases_1 | 2017-01-13T09:47:56.704249Z 0 [Note] Beginning of list of non-n
atively partitioned tables
databases_1 | 2017-01-13T09:47:56.729027Z 0 [Note] End of list of non-native
partitioned tables
databases_1 | 2017-01-13T09:47:56.729233Z 0 [Note] mysqld: ready for connectio
ns.
databases_1 | Version: '5.7.17'  socket: '/var/run/mysqld/mysqld.sock'  port:
3306  MySQL Community Server (GPL)
```

And when you do a **docker ps**, you can see that the containers are indeed up and running.

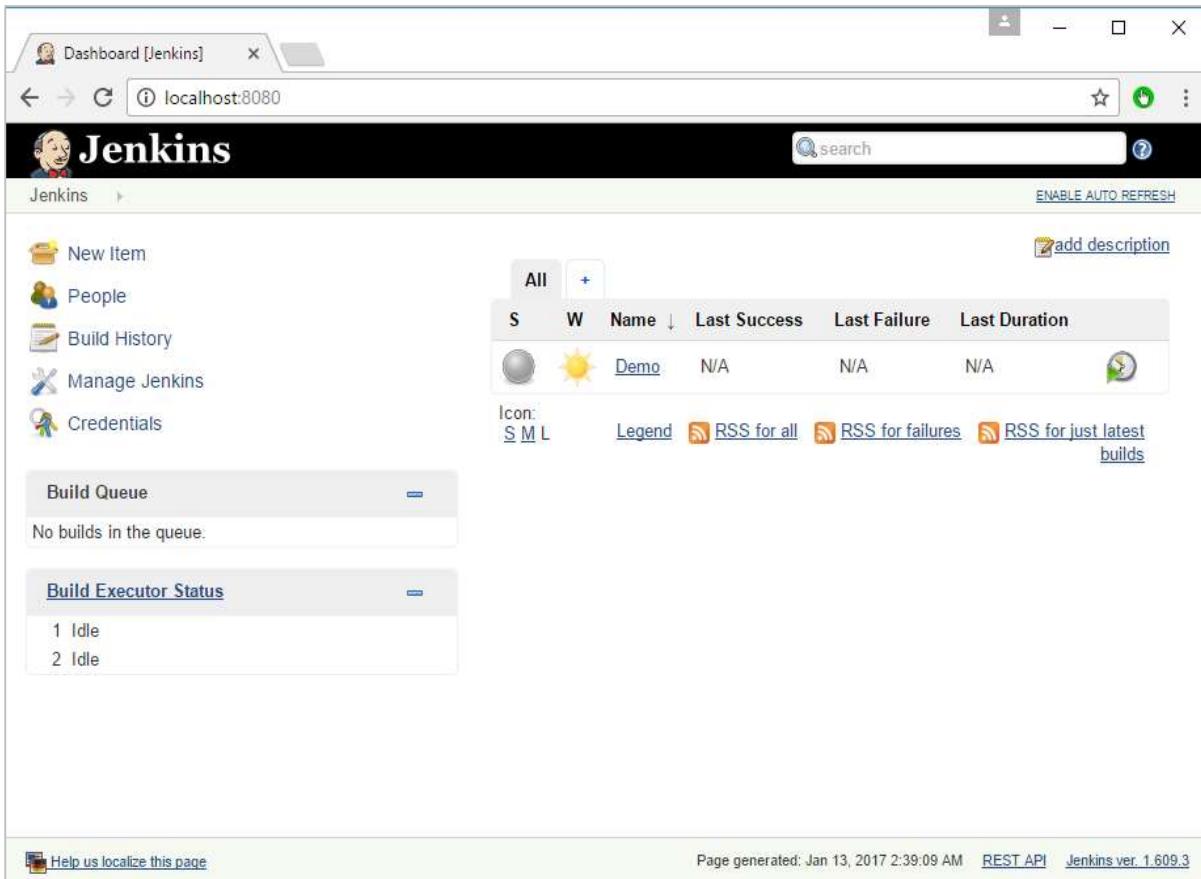
```
[localhost] ignored in --skip-name-resolve mode.
databases_1 | 2017-01-13T09:47:56.684849Z 0 [Warning] 'proxies_priv' entry 'r
oot@localhost' ignored in --skip-name-resolve mode.
databases_1 | 2017-01-13T09:47:56.688368Z 0 [Warning] 'tables_priv' entry 'sy
config mysql.sys@localhost' ignored in --skip-name-resolve mode.
databases_1 | 2017-01-13T09:47:56.703852Z 0 [Note] Event Scheduler: Loaded 0
events
databases_1 | 2017-01-13T09:47:56.704110Z 0 [Note] Executing 'SELECT * FROM I
FORMATION_SCHEMA.TABLES;' to get a list of tables using the deprecated partition
engine. You may use the startup option '--disable-partition-engine-check' to s
kip this check.
databases_1 | 2017-01-13T09:47:56.704249Z 0 [Note] Beginning of list of non-n
atively partitioned tables
databases_1 | 2017-01-13T09:47:56.729027Z 0 [Note] End of list of non-native
partitioned tables
databases_1 | 2017-01-13T09:47:56.729233Z 0 [Note] mysqld: ready for connectio
ns.
databases_1 | Version: '5.7.17'  socket: '/var/run/mysqld/mysqld.sock'  port:
3306  MySQL Community Server (GPL)
^Z
[5]+  Stopped                  sudo ./docker-compose up
demo@ubuntudemo:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            NAMES
d3501a8bf6c4        mysql              "docker-entrypoint.sh"   57 minutes ago   demo_databases_1
b45c8a2a1b4f        nginx              "nginx -g 'daemon off'"  About an hour ago  demo_web_1
demo@ubuntudemo:~$
```

30. Docker – Continuous Integration

Docker has integrations with many Continuous Integrations tools, which also includes the popular CI tool known as **Jenkins**. Within Jenkins, you have plugins available which can be used to work with containers. So let's quickly look at a Docker plugin available for the Jenkins tool.

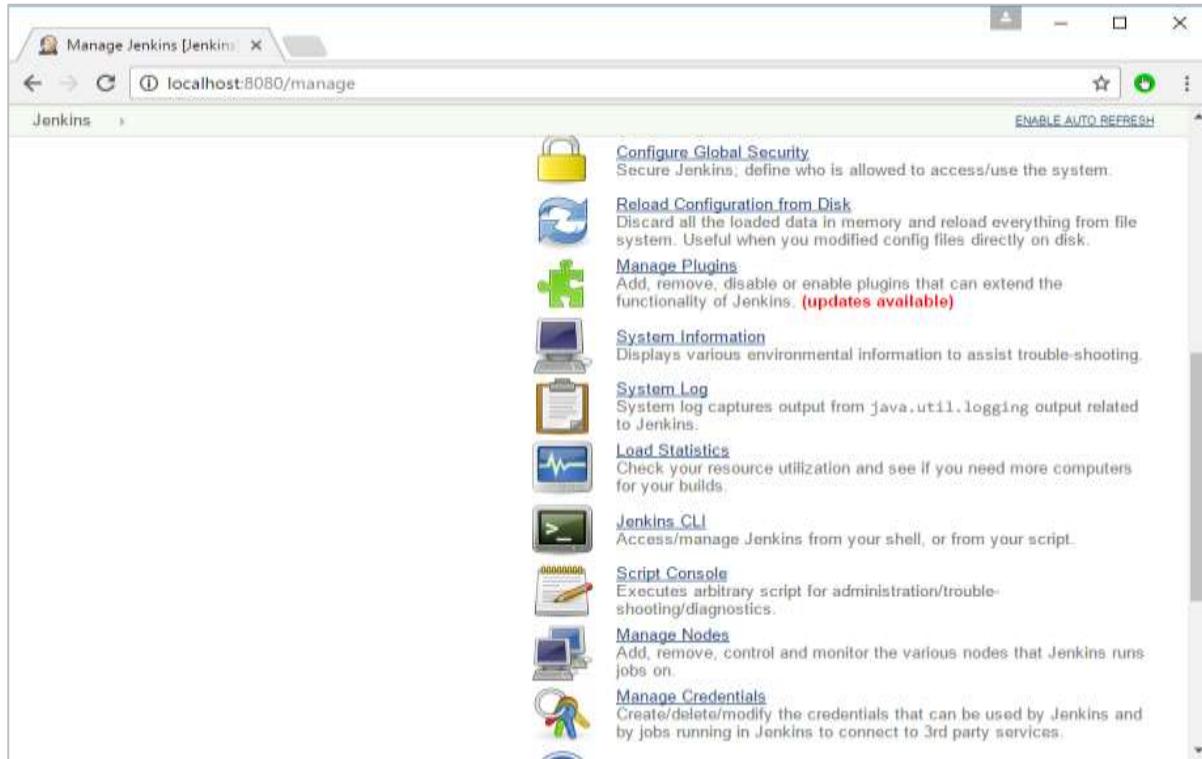
Let's go step by step and see what's available in Jenkins for Docker containers.

Step 1: Go to your Jenkins dashboard and click **Manage Jenkins**.



The screenshot shows the Jenkins dashboard at localhost:8080. On the left sidebar, there are links for New Item, People, Build History, Manage Jenkins, and Credentials. The main area displays a table of builds with columns: S (Status), W (Last Result), Name (Demo), Last Success (N/A), Last Failure (N/A), and Last Duration. Below the table, there are icons for S, M, and L, and links for Legend, RSS for all, RSS for failures, and RSS for just latest builds. At the bottom, it says "No builds in the queue." and "Build Executor Status" with two entries: "1 Idle" and "2 Idle". The footer includes links for Help us localize this page, REST API, and Jenkins ver. 1.609.3.

Step 2: Go to Manage Plugins.



Step 3: Search for Docker plugins. Choose the Docker plugin and click the **Install without restart** button.

Install ↓	Name	Version
<input type="checkbox"/>	CloudBees Docker Build and Publish plugin	
<input type="checkbox"/>	This plugin provides the ability to build projects with a Dockerfile, and publish the resultant tagged image (repo) to a Docker registry.	1.3.2
<input type="checkbox"/>	Docker build step plugin	1.37
<input type="checkbox"/>	This plugin allows to add various Docker commands into your job as a build step	
<input type="checkbox"/>	Quay.io trigger plugin	0.1
<input checked="" type="checkbox"/>	This plugin provides integration between Jenkins and Quay.io docker registry.	
<input checked="" type="checkbox"/>	Docker plugin	0.16.1
<input type="checkbox"/>	This plugin allows slaves to be dynamically provisioned using Docker.	
<input type="checkbox"/>	Amazon EC2 Container Service plugin with autoscaling capabilities	1.0
<input type="checkbox"/>	Use Amazon ECS Containers to setup (docker-based) elastic build executors	

Buttons at the bottom: **Install without restart**, **Download now and install after restart**, and **Update**.

Step 4: Once the installation is completed, go to your job in the Jenkins dashboard. In our example, we have a job called **Demo**.

The screenshot shows the Jenkins dashboard at localhost:8080. On the left, there's a sidebar with links like 'New Item', 'People', 'Build History', 'Manage Jenkins', and 'Credentials'. The main area displays a table of jobs. The 'Demo' job is listed with the following details: Name: Demo, Last Success: 23 min - #2, Last Failure: N/A, Last Duration: 0.53 sec. Below the table, there are sections for 'Build Queue' (No builds in the queue) and 'Build Executor Status' (1 Idle, 2 Idle). At the bottom, there are links for 'Help us localize this page', 'Page generated: Jan 13, 2017 3:11:24 AM', 'REST API', and 'Jenkins ver. 1.609.3'.

Step 5: In the job, when you go to the Build step, you can now see the option to start and stop containers.

The screenshot shows the configuration page for the 'Demo' job at localhost:8080/job/Demo/configure. It includes sections for 'Source Code Management' (set to 'None'), 'Build Triggers' (with options for 'Build after other projects are built', 'Build periodically', and 'Poll SCM'), and 'Build' (with a dropdown menu titled 'Add build step'). The 'Add build step' menu lists several options: 'Add a new template to all docker clouds', 'Execute Windows batch command', 'Execute shell', 'Invoke Ant', 'Invoke top-level Maven targets', and 'Start/Stop Docker Containers'. The 'Start/Stop Docker Containers' option is highlighted with a blue border. At the bottom, there are links for 'Help us localize this page', 'Page generated: Jan 13, 2017 2:45:37 AM', 'REST API', and 'Jenkins ver. 1.609.3'.

Step 6: As a simple example, you can choose the further option to stop containers when the build is completed. Then, click the **Save** button.

The screenshot shows the Jenkins configuration page for the 'Demo' job. In the 'Build' section, there is a 'Start/Stop Docker Containers' panel. Under 'Action to choose', the dropdown is set to 'Stop All Containers'. Below this is a 'Delete' button. At the bottom of the panel are 'Add build step' and 'Post-build Actions' buttons. At the very bottom are 'Save' and 'Apply' buttons.

Now, just run your job in Jenkins. In the Console output, you will now be able to see that the command to Stop All containers has run.

The screenshot shows the Jenkins Console Output for build #2. The main title is 'Console Output'. The log output shows the following commands were run:

```

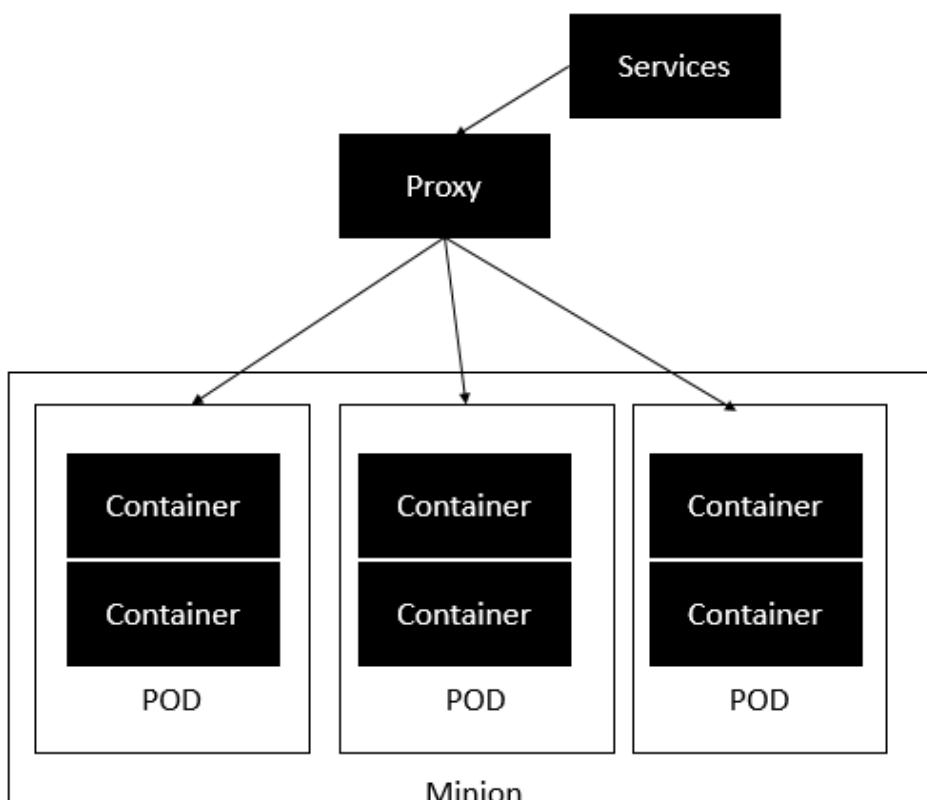
Started by user anonymous
Building in workspace C:\Users\s362692\.hudson\jobs\Demo\workspace
Stopping all containers
Finished: SUCCESS

```

31. Docker – Kubernetes Architecture

Kubernetes is an orchestration framework for Docker containers which helps expose containers as services to the outside world. For example, you can have two services: One service would contain **nginx** and **mongoDB**, and another service would contain **nginx** and **redis**. Each service can have an IP or service point which can be connected by other applications. Kubernetes is then used to manage these services.

The following diagram shows in a simplistic format how Kubernetes works from an architecture point of view.



The **minion** is the node on which all the services run. You can have many minions running at one point in time. Each minion will host one or more **POD**. Each POD is like hosting a service. Each POD then contains the Docker containers. Each POD can host a different set of Docker containers. The proxy is then used to control the exposing of these services to the outside world.

Kubernetes has several components in its architecture. The role of each component is explained below:

- **etcd** – This component is a highly available **key-value** store that is used for storing **shared configuration** and **service discovery**. Here the various applications will be able to connect to the services via the **discovery service**.
- **Flannel** – This is a backend network which is required for the containers.
- **kube-apiserver** – This is an API which can be used to orchestrate the Docker containers.
- **kube-controller-manager** – This is used to control the **Kubernetes services**.
- **kube-scheduler** – This is used to schedule the containers on hosts.
- **Kubelet** – This is used to control the launching of containers via **manifest files**.
- **kube-proxy** – This is used to provide network proxy services to the outside world.

32. Docker – Working of Kubernetes

In this chapter, we will see how to install **Kubernetes** via **kubeadm**. This is a tool which helps in the installation of Kubernetes. Let's go step by step and learn how to install Kubernetes:

Step 1: Ensure that the **Ubuntu server version** you are working on is **16.04**.

Step 2: Ensure that you generate a **ssh** key which can be used for **ssh** login. You can do this using the following command:

```
ssh-keygen
```

This will generate a key in your **home folder** as shown below.

```
master@master:~$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/master/.ssh/id_rsa):
Created directory '/home/master/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/master/.ssh/id_rsa.
Your public key has been saved in /home/master/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:a30Av0xfNvitwylJsT2Rt71keq90hrp7qUSjqSo5w84 master@master
The key's randomart image is:
+---[RSA 2048]----+
|                               |
|                               |
|                               |
|                               . |
|     . . o . |
|     o S+oo o |
|     . .oBo...+ |
| . . . o.Box*o.=o. |
| . * . ooB++oo |
| .E+...o...BB.ooo. |
+---[SHA256]----+
master@master:~$ _
```

Step 3: Next, depending on the version of Ubuntu you have, you will need to add the relevant site to the **docker.list** for the **apt package manager**, so that it will be able to detect the **Kubernetes packages** from the **kubernetes** site and download them accordingly.

We can do it using the following commands:

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
echo "deb http://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee
/etc/apt/sources.list.d/docker.list
```

Step 4: We then issue an apt-get update to ensure all packages are downloaded on the Ubuntu server.

```
Get:25 http://us.archive.ubuntu.com/ubuntu xenial/universe amd64 Packages [7,532 kB]
Get:26 http://us.archive.ubuntu.com/ubuntu xenial/universe i386 Packages [7,512 kB]
Get:27 http://us.archive.ubuntu.com/ubuntu xenial/universe Translation-en [4,354 kB]
Get:28 http://us.archive.ubuntu.com/ubuntu xenial/multiverse amd64 Packages [144 kB]
Get:29 http://us.archive.ubuntu.com/ubuntu xenial/multiverse i386 Packages [140 kB]
Get:30 http://us.archive.ubuntu.com/ubuntu xenial/multiverse Translation-en [106 kB]
Get:31 http://us.archive.ubuntu.com/ubuntu xenial-updates/main amd64 Packages [452 kB]
Get:32 http://us.archive.ubuntu.com/ubuntu xenial-updates/main i386 Packages [444 kB]
Get:33 http://us.archive.ubuntu.com/ubuntu xenial-updates/main Translation-en [178 kB]
Get:34 http://us.archive.ubuntu.com/ubuntu xenial-updates/restricted amd64 Packages [6,576 B]
Get:35 http://us.archive.ubuntu.com/ubuntu xenial-updates/restricted i386 Packages [6,528 B]
Get:36 http://us.archive.ubuntu.com/ubuntu xenial-updates/restricted Translation-en [2,016 B]
Get:37 http://us.archive.ubuntu.com/ubuntu xenial-updates/universe amd64 Packages [378 kB]
Get:38 http://us.archive.ubuntu.com/ubuntu xenial-updates/universe i386 Packages [373 kB]
Get:39 http://us.archive.ubuntu.com/ubuntu xenial-updates/universe Translation-en [140 kB]
Get:40 http://us.archive.ubuntu.com/ubuntu xenial-updates/multiverse amd64 Packages [7,384 B]
Get:41 http://us.archive.ubuntu.com/ubuntu xenial-updates/multiverse i386 Packages [6,180 B]
Get:42 http://us.archive.ubuntu.com/ubuntu xenial-updates/multiverse Translation-en [3,080 B]
Get:43 http://us.archive.ubuntu.com/ubuntu xenial-backports/main amd64 Packages [4,404 B]
Get:44 http://us.archive.ubuntu.com/ubuntu xenial-backports/main i386 Packages [4,404 B]
Get:45 http://us.archive.ubuntu.com/ubuntu xenial-backports/main Translation-en [3,124 B]
Get:46 http://us.archive.ubuntu.com/ubuntu xenial-backports/universe amd64 Packages [2,412 B]
Get:47 http://us.archive.ubuntu.com/ubuntu xenial-backports/universe i386 Packages [2,412 B]
Get:48 http://us.archive.ubuntu.com/ubuntu xenial-backports/universe Translation-en [1,216 B]
Fetched 26.0 MB in 57s (455 kB/s)
Reading package lists... Done
root@slave:~#
```

Step 5: Install the Docker package as detailed in the earlier chapters.

Step 6: Now it's time to install **kubernetes** by installing the following packages:

```
apt-get install -y kubelet kubeadm kubectl kubernetes-cni
```

```
root@slave:~# apt-get install -y kubelet kubeadm kubectl kubernetes-cni_
```

```
Preparing to unpack .../kubernetes-cni_0.3.0.1-07a8a2-00_amd64.deb ...
Unpacking kubernetes-cni (0.3.0.1-07a8a2-00) ...
Selecting previously unselected package socat.
Preparing to unpack .../socat_1.7.3.1-1_amd64.deb ...
Unpacking socat (1.7.3.1-1) ...
Selecting previously unselected package kubelet.
Preparing to unpack .../kubelet_1.5.1-00_amd64.deb ...
Unpacking kubelet (1.5.1-00) ...
Selecting previously unselected package kubectl.
Preparing to unpack .../kubectl_1.5.1-00_amd64.deb ...
Unpacking kubectl (1.5.1-00) ...
Selecting previously unselected package kubeadm.
Preparing to unpack .../kubeadm_1.6.0-alpha.0-2074-a092d8e0f95f52-00_amd64.deb ...
Unpacking kubeadm (1.6.0-alpha.0-2074-a092d8e0f95f52-00) ...
Processing triggers for systemd (229-4ubuntu7) ...
Processing triggers for ureadahead (0.100.0-19) ...
Processing triggers for man-db (2.7.5-1) ...
Setting up ebtables (2.0.10.4-3.4ubuntu1) ...
update-rc.d: warning: start and stop actions are no longer supported; falling back to defaults
Setting up kubernetes-cni (0.3.0.1-07a8a2-00) ...
Setting up socat (1.7.3.1-1) ...
Setting up kubelet (1.5.1-00) ...
Setting up kubectl (1.5.1-00) ...
Setting up kubeadm (1.6.0-alpha.0-2074-a092d8e0f95f52-00) ...
Processing triggers for systemd (229-4ubuntu7) ...
Processing triggers for ureadahead (0.100.0-19) ...
root@slave:~#
```

Step 7: Once all **kubernetes** packages are downloaded, it's time to start the **kubernetes** controller using the following command:

```
kubeadm init
```

```
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/admin.conf"
[apiclient] Created API client, waiting for the control plane to become ready
[apiclient] All control plane components are healthy after 113.293014 seconds
[apiclient] Waiting for at least one node to register and become ready
[apiclient] First node is ready after 6.502838 seconds
[apiclient] Creating a test deployment
[apiclient] Test deployment succeeded
[token-discovery] Created the kube-discovery deployment, waiting for it to become ready
[token-discovery] kube-discovery is ready after 55.503574 seconds
[addons] Created essential addon: kube-proxy
[addons] Created essential addon: kube-dns

Your Kubernetes master has initialized successfully!

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  http://kubernetes.io/docs/admin/addons/

You can now join any number of machines by running the following on each node:
kubeadm join --token=101573.c17ba345fc84fb71 192.168.1.105
root@slave:~# _
```

Once done, you will get a successful message that the master is up and running and nodes can now join the cluster.