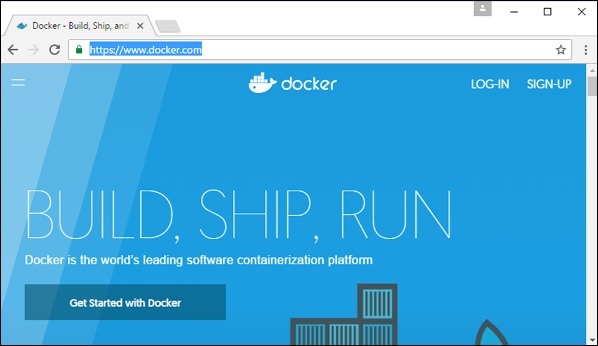
Docker - Overview

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

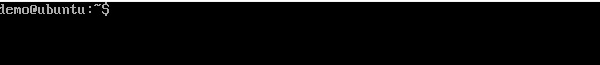
* **Docker for Mac** − It allows one to run Docker containers on the Mac OS.
* **Docker for Linux** − It allows one to run Docker containers on the Linux OS.
* **Docker for Windows** − It allows one to run Docker containers on the Windows OS.
* **Docker Engine** − It is used for building Docker images 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.

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.



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 −



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

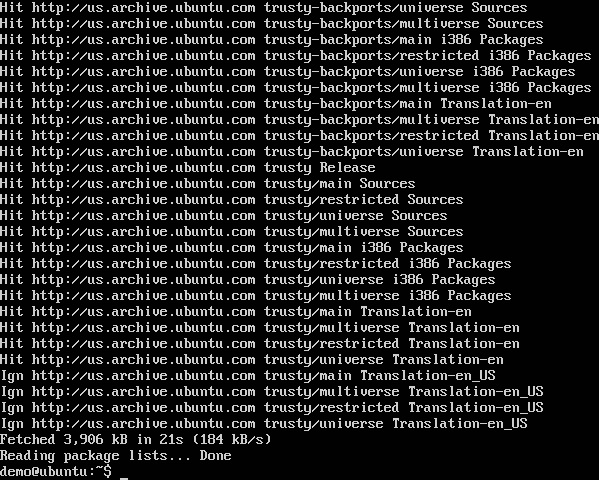
None

### Example

sudo apt-get update

### Output

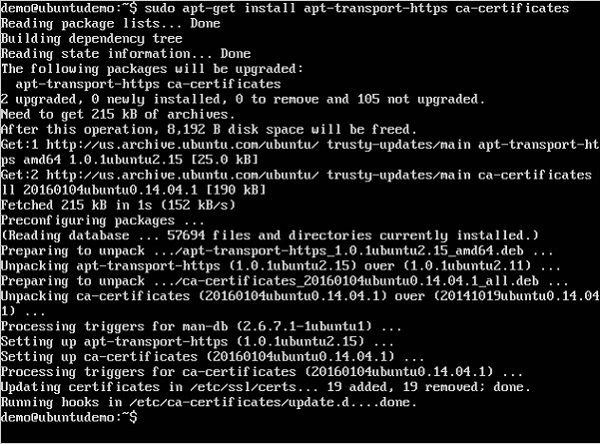
When we run the above command, we will get the following result −



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

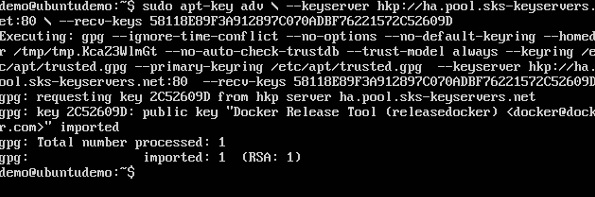
**Step 3** − The next step is to install the necessary certificates that will be required to work with the Docker site later on to download the necessary Docker packages. It can be done with the following command.

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



**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.



**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](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](https://apt.dockerproject.org/repo/) ubuntu-wily main
* Xenial 16.04 (LTS) - [https://apt.dockerproject.org/repo](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](https://apt.dockerproject.org/repo/)ubuntu-trusty main”.

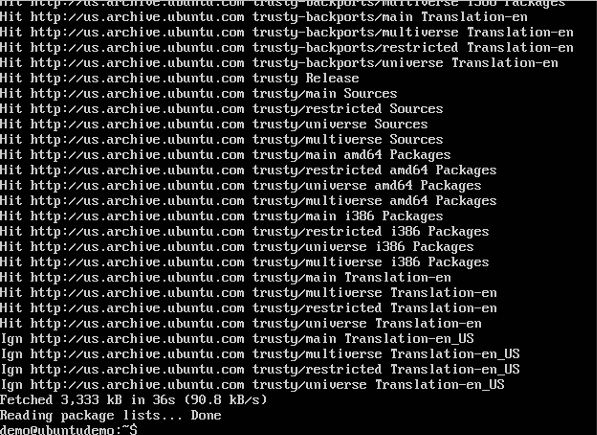
And then, we will need to add this repository to the **docker.list** as mentioned above.

echo "deb https://apt.dockerproject.org/repo ubuntu-trusty main”

| sudo tee /etc/apt/sources.list.d/docker.list



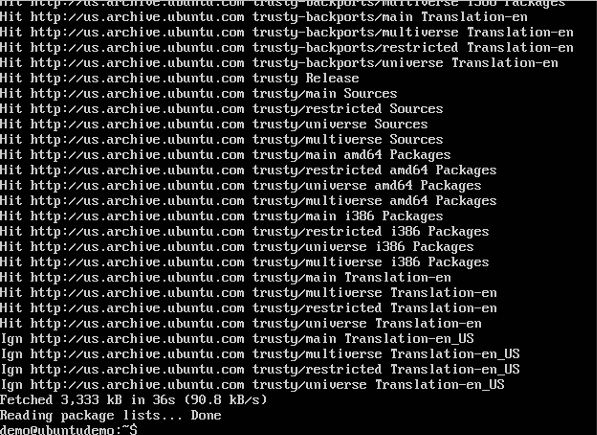
**Step 6** − Next, we issue the **apt-get update command** to update the packages on the Ubuntu system.



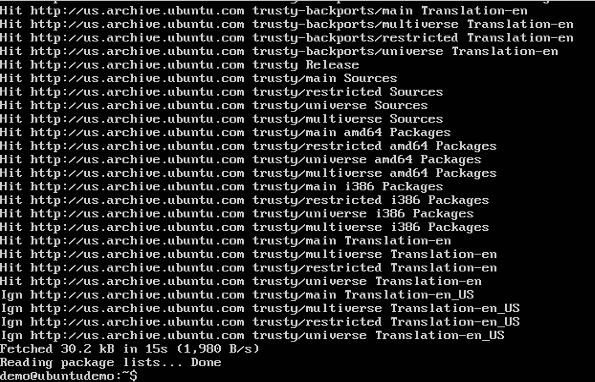
**Step 7** − If you want to verify that the package manager is pointing to the right repository, you can do it by issuing the **apt-cache command**.

apt-cache policy docker-engine

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



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

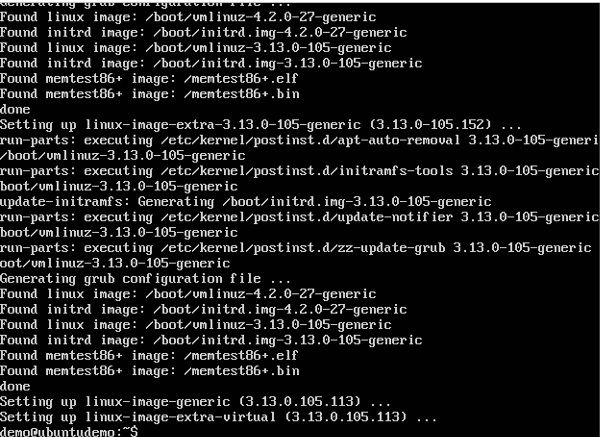


**Step 9** − For Ubuntu Trusty, Wily, and Xenial, we have to install the linux-image-extra-\* kernel packages, which allows one to use the **aufs storage driver**. This driver is used by the newer versions of Docker.

It can be done by using the following command.

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

linux-image-extra-virtual

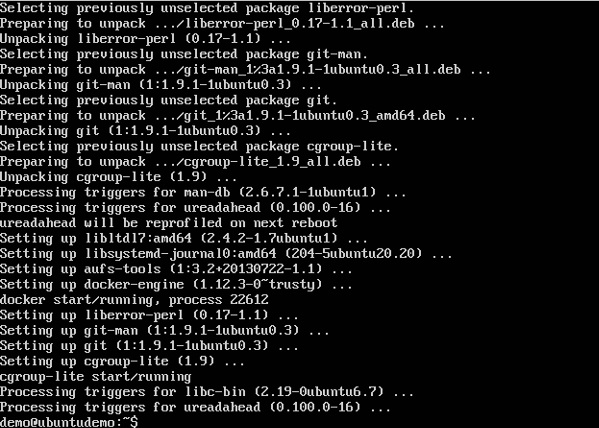


**Step 10** − The final step is to install Docker and we can do this with the following command −

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.



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

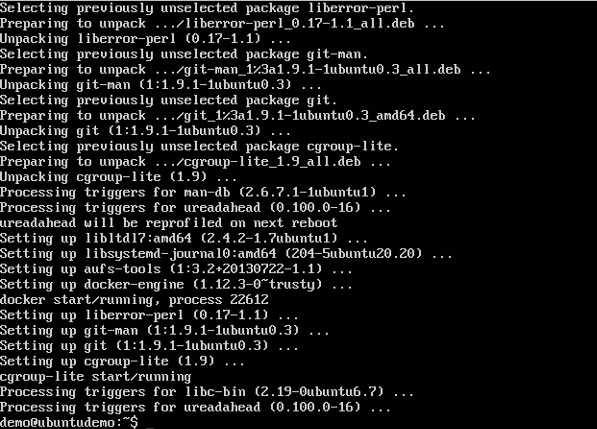
The output will provide the various details of the Docker version installed on the system.

### Example

sudo docker version

### Output

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



## 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 −

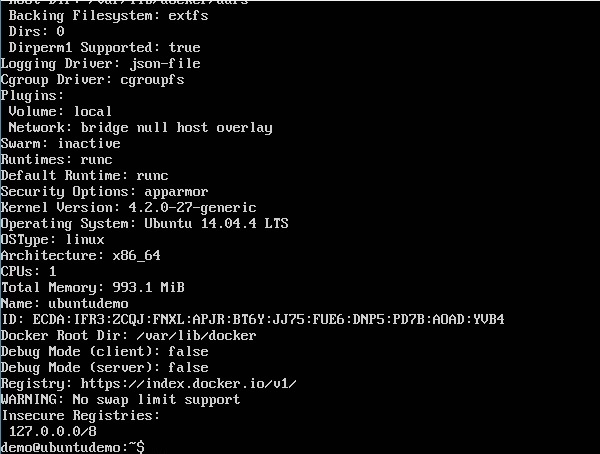
* Number of containers
* Number of images
* The storage driver used by Docker
* The root directory used by Docker
* The execution driver used by Docker

### Example

sudo docker info

### Output

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



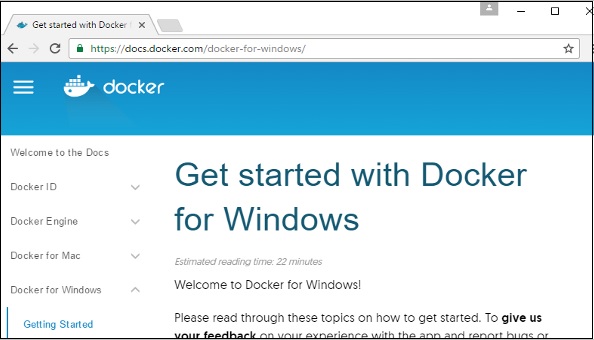
## Docker for Windows

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/>



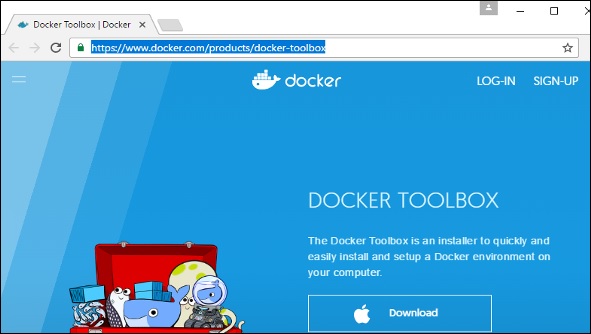
## Docker ToolBox

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>

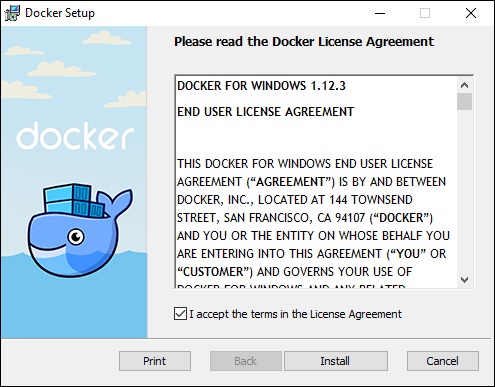


# Docker - Installation

## 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.



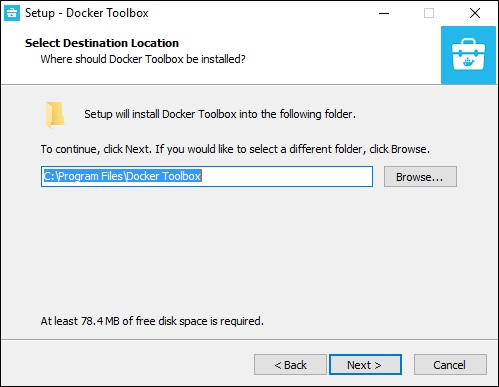
## Docker ToolBox

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

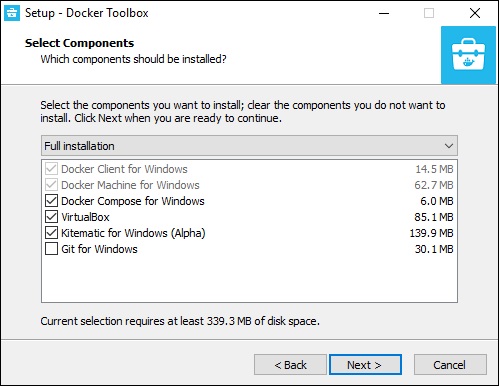
**Step 1** − Click the Next button on the start screen.



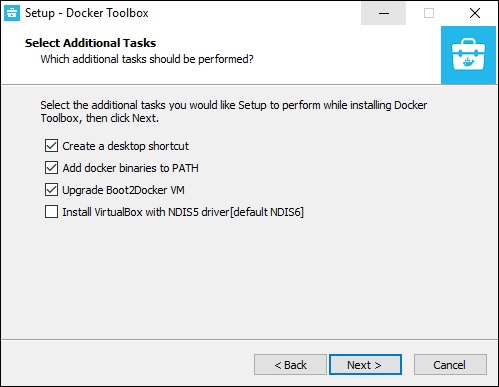
**Step 2** − Keep the default location on the next screen and click the Next button.



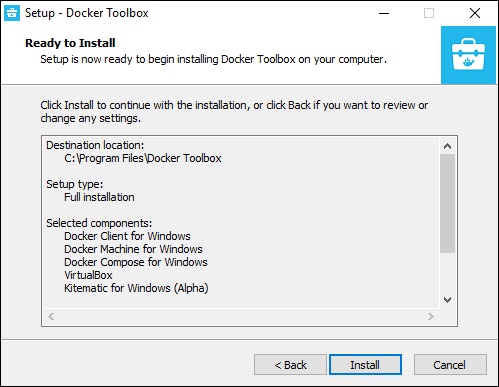
**Step 3** − Keep the default components and click the Next button to proceed.



**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.

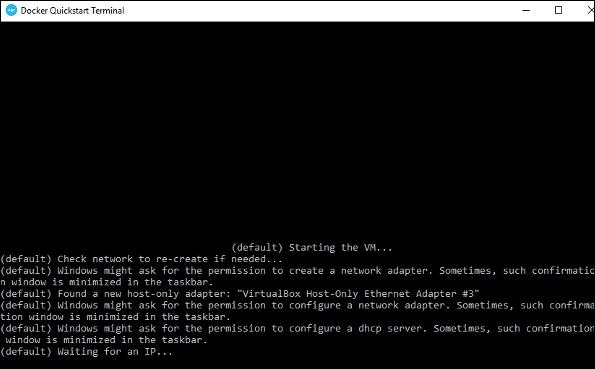


## Working with Docker Toolbox

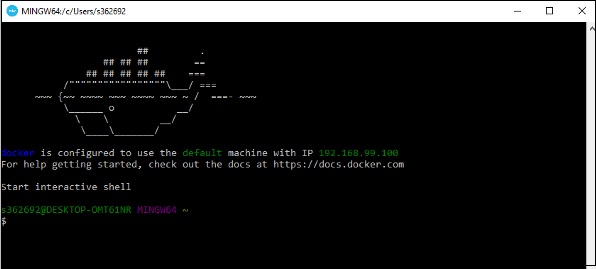
Let’s now look at how Docker Toolbox can be used to work with Docker containers on Windows. The first step is to launch the Docker Toolbox application for which the shortcut is created on the desktop when the installation of Docker toolbox is carried out.



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



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.

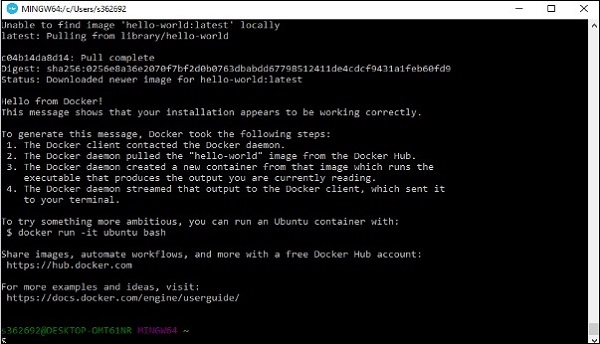
### Example

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

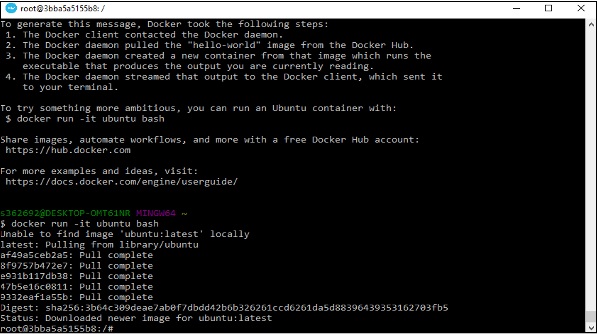
When we run the above command, we will get the following result −



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.



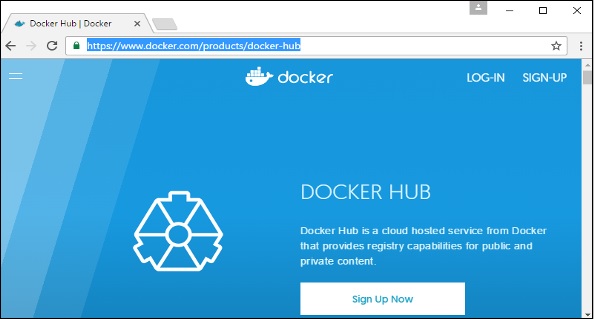
In the output you can see that the Ubuntu image is downloaded and run and then you will be logged in as a root user in the Ubuntu container.

# Docker - Hub

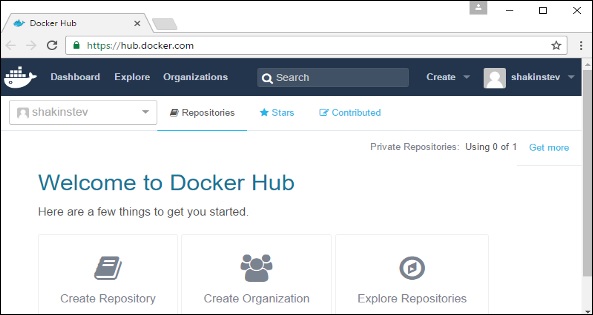
Docker Hub is a registry service on the cloud that allows you to download Docker images that are built by other communities. You can also upload your own Docker built images to Docker hub. In this chapter, we will see how to download and the use the Jenkins Docker image from Docker hub.

The official site for Docker hub is − <https://www.docker.com/community-edition#/add_ons>

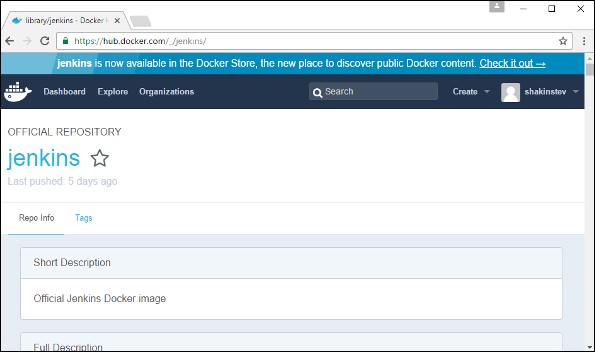
**Step 1** − First you need to do a simple sign-up on Docker hub.



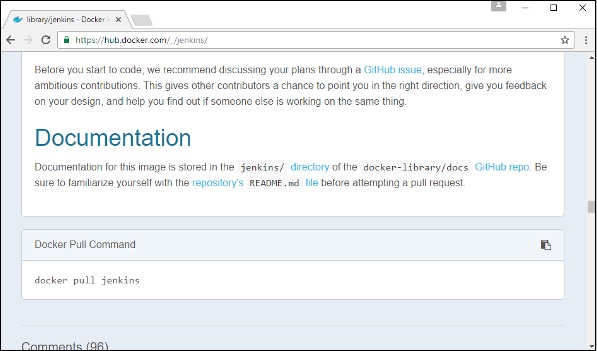
**Step 2** − Once you have signed up, you will be logged into Docker Hub.



**Step 3** − Next, let’s browse and find the Jenkins image.

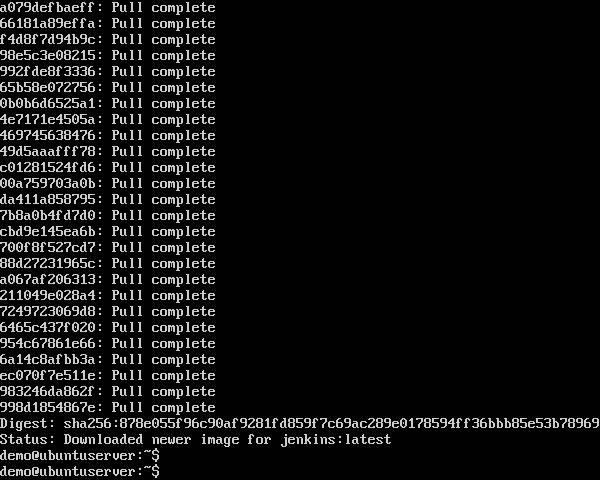


**Step 4** − If you scroll down on the same page, you can see the Docker **pull** command. This will be used to download the Jenkins image onto the local Ubuntu server.



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

sudo docker pull jenkins

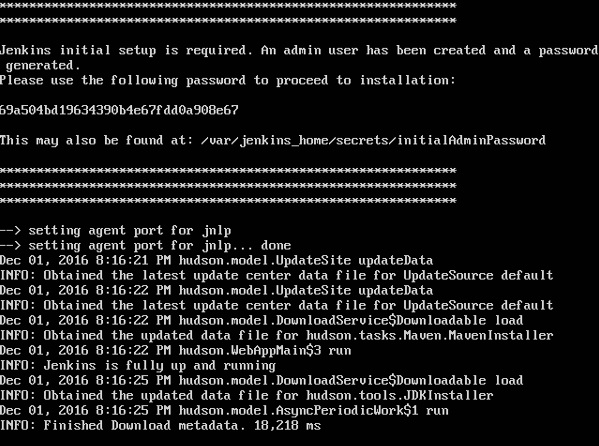


To run Jenkins, you need to run the following command −

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.

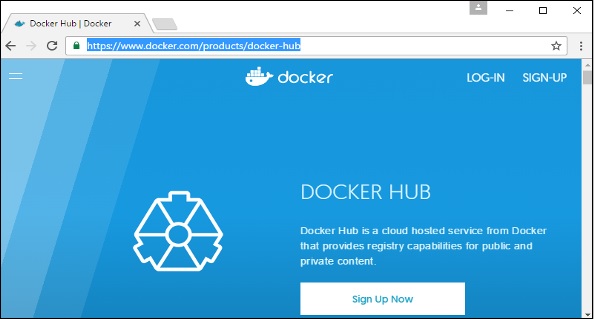


You will then have Jenkins successfully running as a container on the Ubuntu machine.

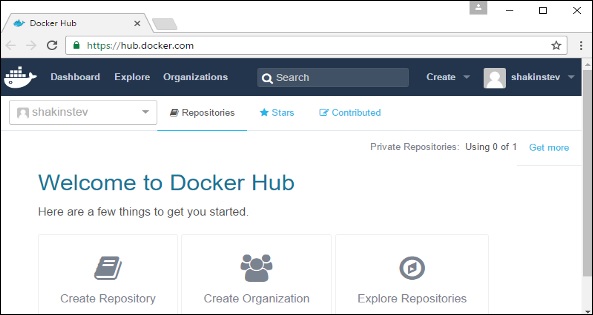
Docker Hub is a registry service on the cloud that allows you to download Docker images that are built by other communities. You can also upload your own Docker built images to Docker hub. In this chapter, we will see how to download and the use the Jenkins Docker image from Docker hub.

The official site for Docker hub is − <https://www.docker.com/community-edition#/add_ons>

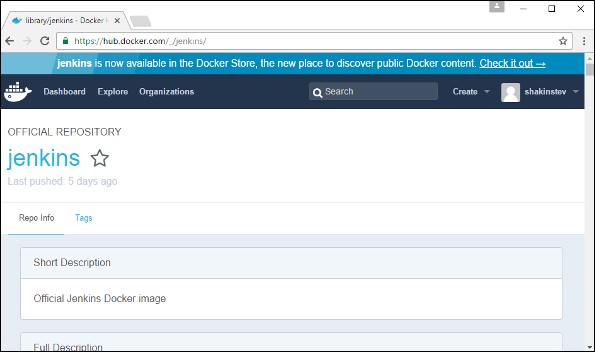
**Step 1** − First you need to do a simple sign-up on Docker hub.



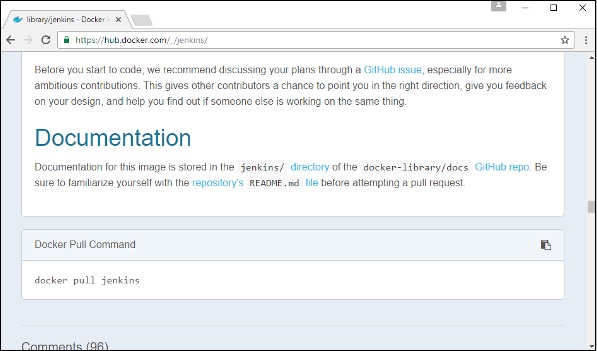
**Step 2** − Once you have signed up, you will be logged into Docker Hub.



**Step 3** − Next, let’s browse and find the Jenkins image.

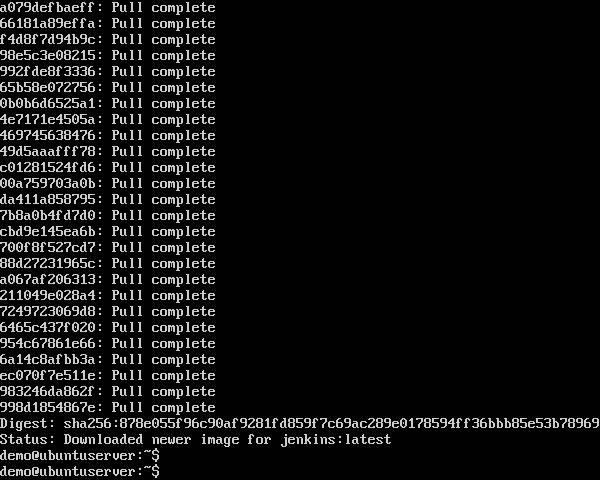


**Step 4** − If you scroll down on the same page, you can see the Docker **pull** command. This will be used to download the Jenkins image onto the local Ubuntu server.



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

sudo docker pull jenkins

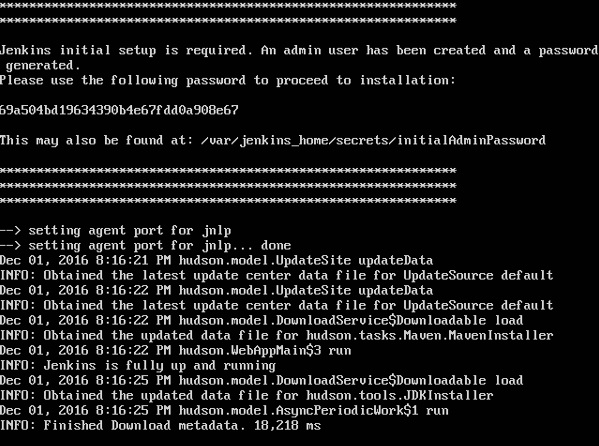


To run Jenkins, you need to run the following command −

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.



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

# 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 -it centos /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 −



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.

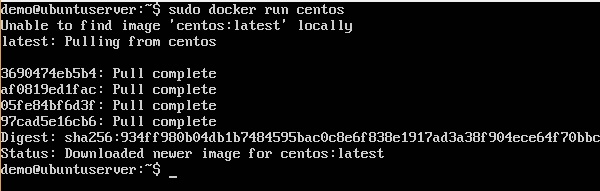
### Example

sudo docker run centos

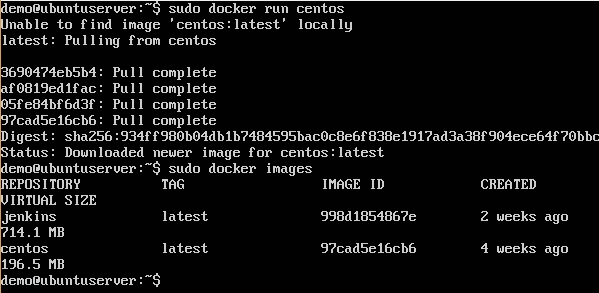
This command will download the **centos** image, if it is not already present, and run the OS as a container.

### Output

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



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.



## 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.

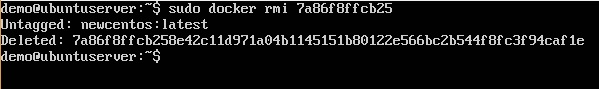
### Example

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 −



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

### 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 −



## 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 −

# Docker Inspect

# Docker – Containers

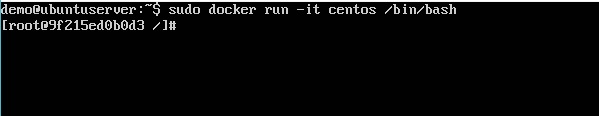
Containers are instances of Docker images that can be run using the Docker run command. The basic purpose of Docker is to run containers. Let’s discuss how to work with containers.

## 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 Crtl+p and you will return to your OS shell.



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

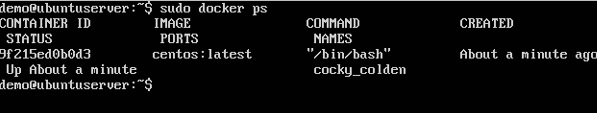
The output will show the currently running containers.

### Example

sudo docker ps

### Output

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



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 −



## 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 −

# Docker History

# 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 −



## 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.

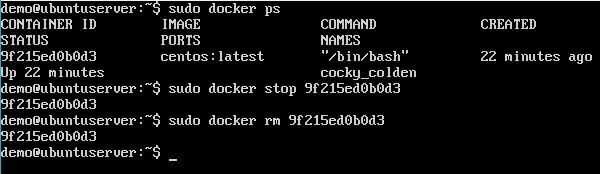
### Example

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 −



## 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.

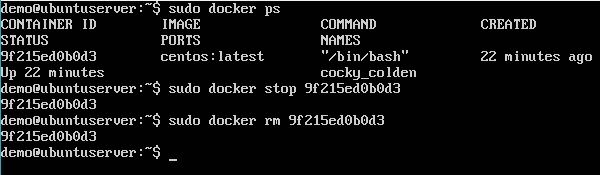
### Example

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 −



## 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.

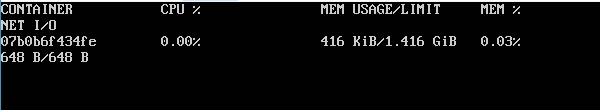
### Example

sudo docker stats 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 −



## 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

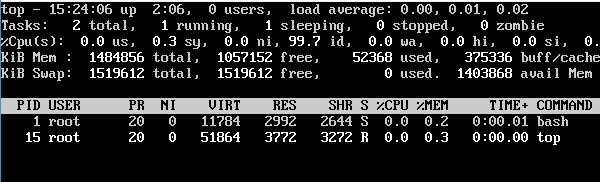
The above command will attach to the Docker container **07b0b6f434fe**.

### Output

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



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



## 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 −



## 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.

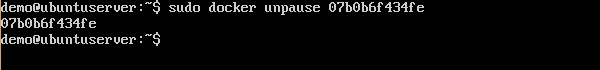
### Example

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 −



## 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.

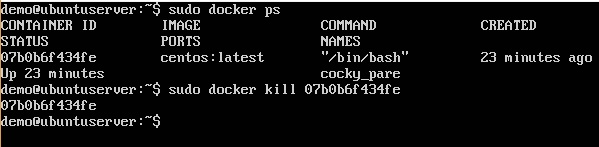
### Example

sudo docker kill 07b0b6f434fe

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

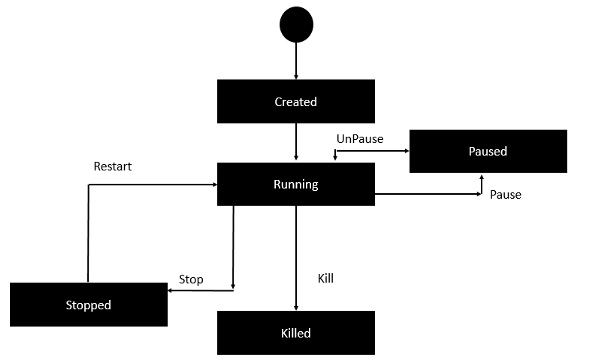
### Output

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



## Docker – Container Lifecycle

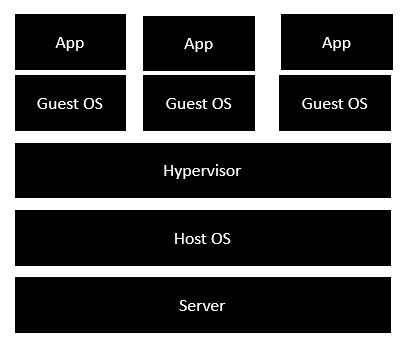
The following illustration explains the entire lifecycle of a Docker container.



* 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.

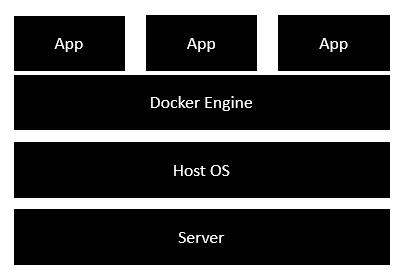
# Docker - Architecture

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.

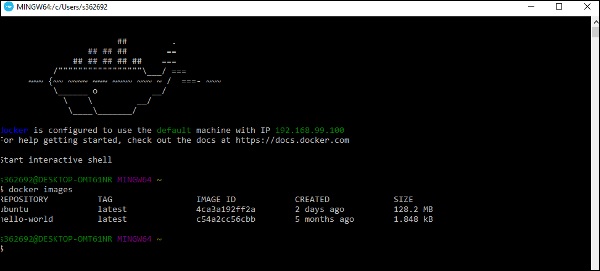
The clear advantage in this architecture is that you don’t need to have extra hardware for Guest OS. Everything works as Docker containers.

# 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

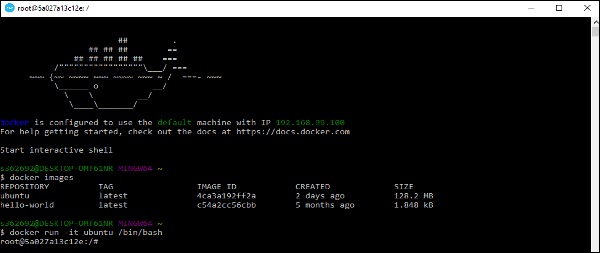
Let’s run the Docker **images** command on the Windows host.



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

## Running a Container

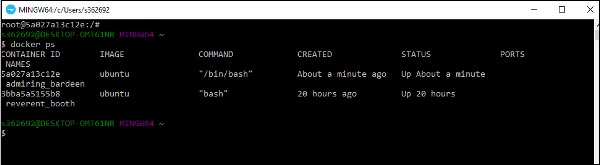
Now let’s run a container in the Windows Docker host.



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

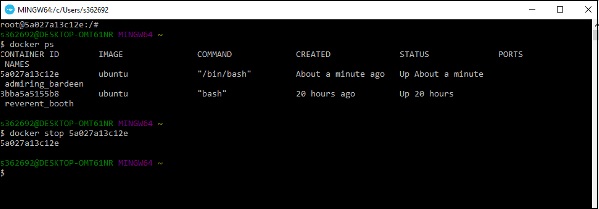
## Listing All Containers

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



## Stopping a Container

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



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.

# Docker - Configuring

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

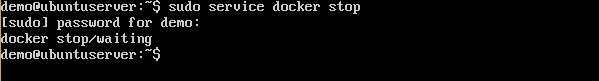
A message showing that the Docker process has stopped.

### Example

sudo service docker stop

### Output

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



## 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

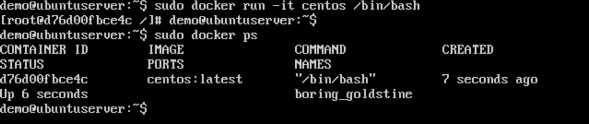
### Output

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

# Service Docker Start

# 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.



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

sudo docker run –it centos /bin/bash

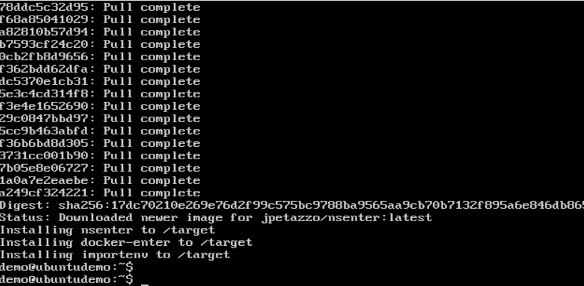
We used this command to create a new container and then used the Ctrl+P+Q command to exit out of the container. It ensures that the container still exists even after we exit from the container.

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.

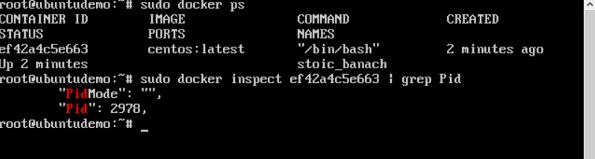
Now there is an easier way to attach to containers and exit them cleanly without the need of destroying them. One way of achieving this is by using the **nsenter** command.

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



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**.



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** s 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.

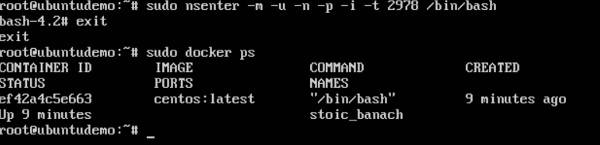
### Return Value

None

### Example

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

### Output



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.

# 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.



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.



**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.

# Build the Image

# Docker - Building 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

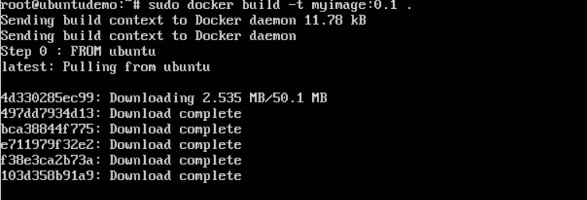
sudo docker build –t myimage:0.1.

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

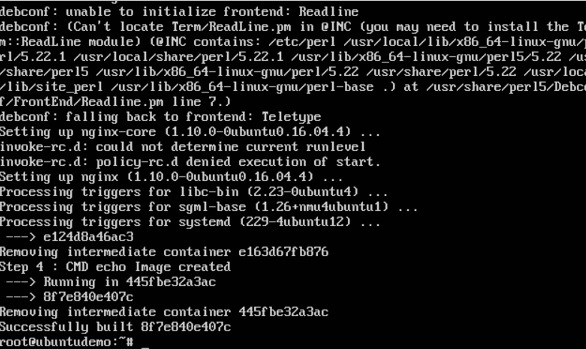
Since the Docker File is in the present working directory, we used "." at the end of the command to signify the present working directory.

### Output

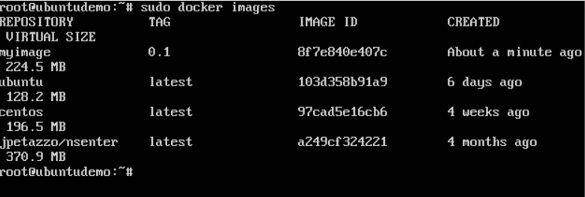
From the output, you will first see that the Ubuntu Image will be downloaded from Docker Hub, because there is no image available locally on the machine.



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



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.



You can now build containers from your new Image.

# 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.



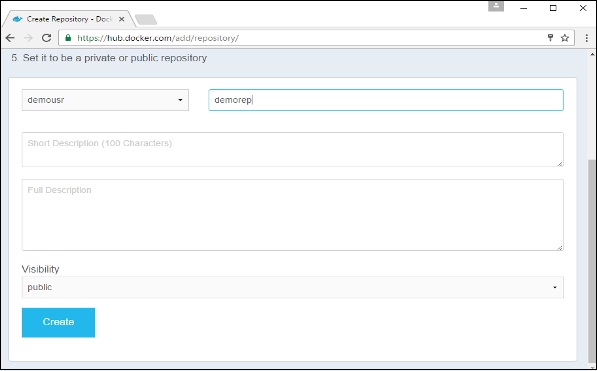
Here, we have our **myimage:0.1** image which was created as a part of the “Building Docker Files” chapter. Let’s use this to upload to the Docker public repository.

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

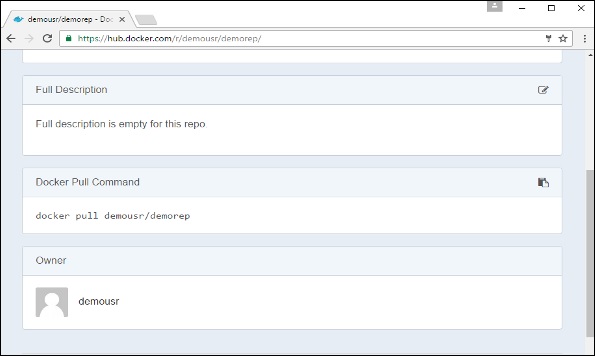
**Step 1** − Log into Docker Hub and create your repository. This is the repository where your image will be stored. Go to <https://hub.docker.com/> and log in with your credentials.



**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.



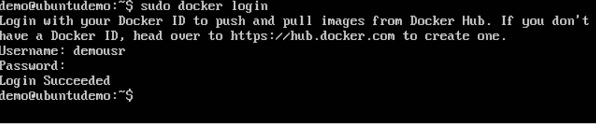
The **pull** command which will be used in our repository is as follows −

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.



**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

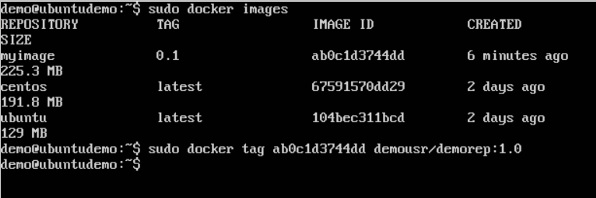
None

### Example

sudo docker tag ab0c1d3744dd demousr/demorep:1.0

### Output

A sample output of the above example is given below.



## 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.

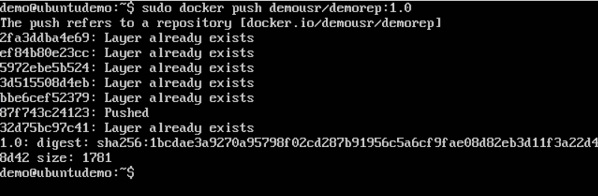
### Return Value

The long ID of the repository pushed to Docker Hub.

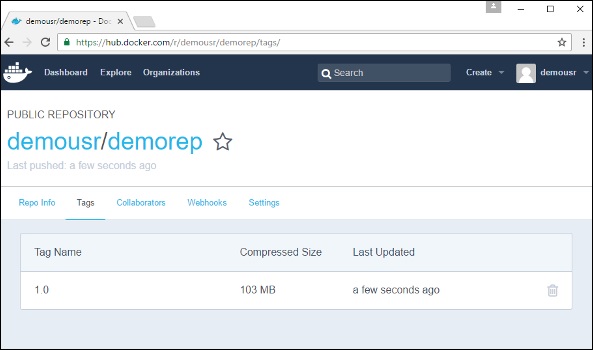
### Example

sudo docker push demousr/demorep:1.0

### Output



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



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/demorep:1.0**, from the local Docker host. Let’s use the Docker **pull command** to pull the repository from the Docker Hub.



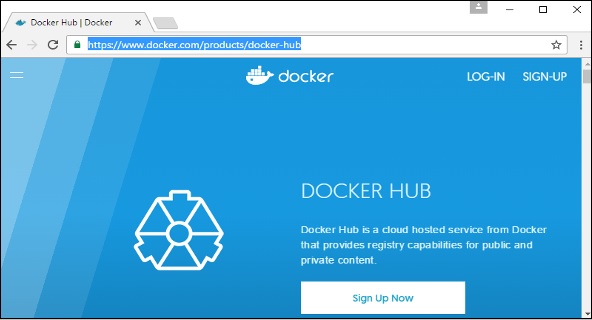
From the above screenshot, you can see that the Docker **pull** command has taken our new repository from the Docker Hub and placed it on our machine.

# Docker - Managing Ports

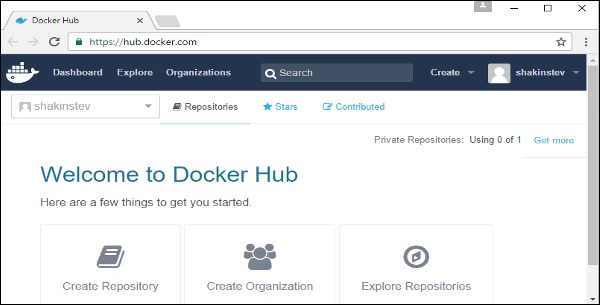
In Docker, the containers themselves can have applications running on ports. When you run a container, if you want to access the application in the container via a port number, you need to map the port number of the container to the port number of the Docker host. Let’s look at an example of how this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

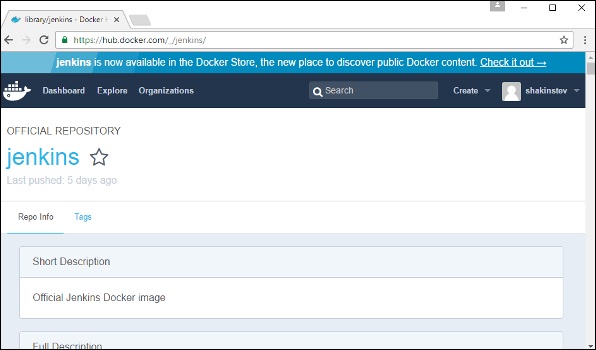
**Step 1** − First, you need to do a simple sign-up on Docker Hub.



**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.

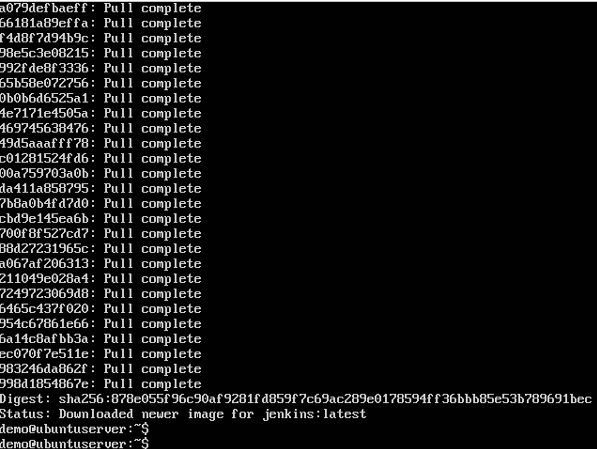


**Step 4** − If you scroll down on the same page, you can see the Docker **pull** command. This will be used to download the Jenkins Image onto the local Ubuntu server.



**Step 5** − Now go to the Ubuntu server and run the command −

sudo docker pull jenkins



**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

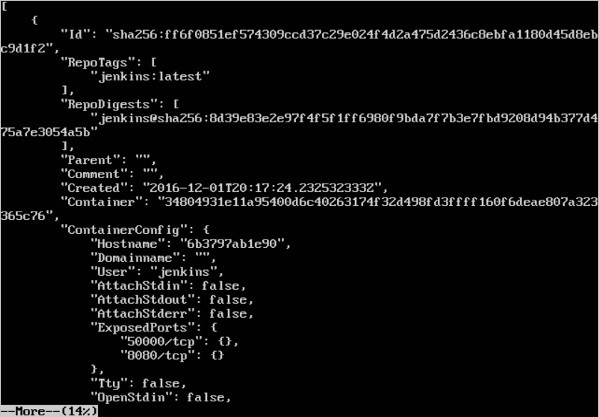
### Return Value

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

### Example

sudo docker inspect jenkins

### Output



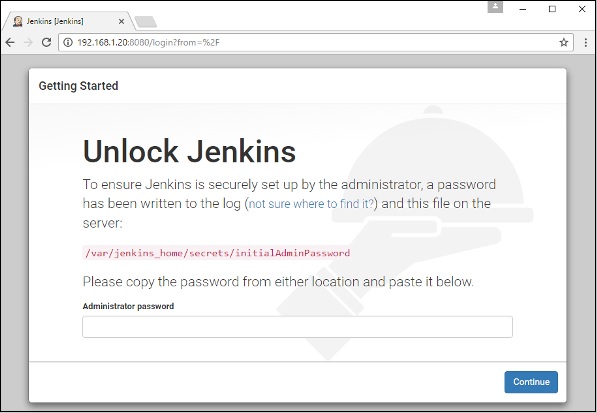
The output of the **inspect** command gives a JSON output. If we observe the output, we can see that there is a section of "ExposedPorts" and see that there are two ports mentioned. One is the **data port** of 8080 and the other is the **control port** of 50000.

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.



# Docker - Private Registries

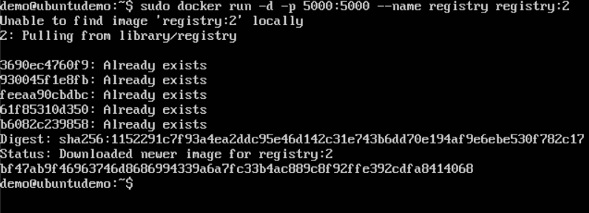
You might have the need to have your own private repositories. You may not want to host the repositories on Docker Hub. For this, there is a repository container itself from Docker. Let’s see how we can download and use the container for registry.

**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



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



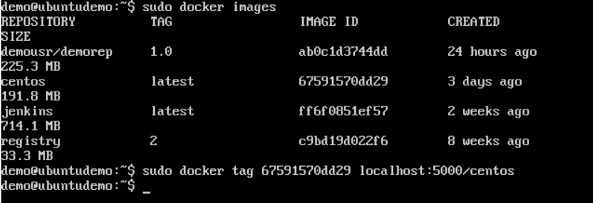
We have now confirmed that the registry container is indeed running.

**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.



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

sudo docker push localhost:5000/centos

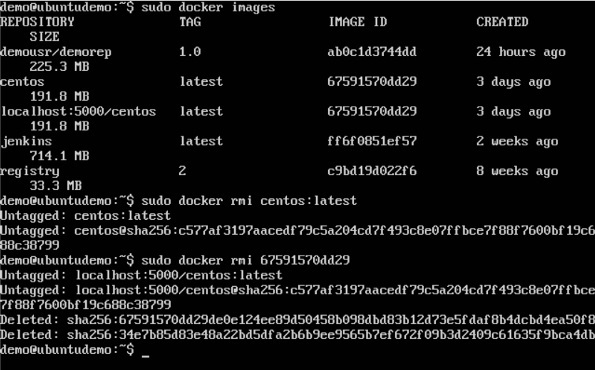
Here, we are pushing the **centos** image to the private repository hosted at **localhost:5000**.



**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

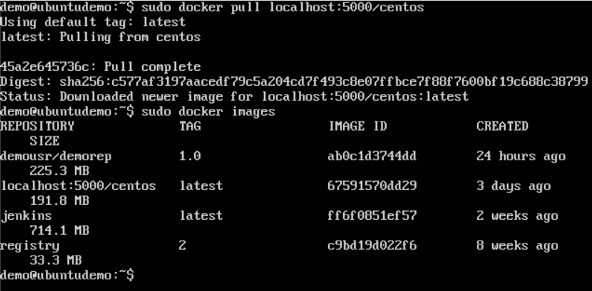
sudo docker rmi 67591570dd29



**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**.



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

# 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.

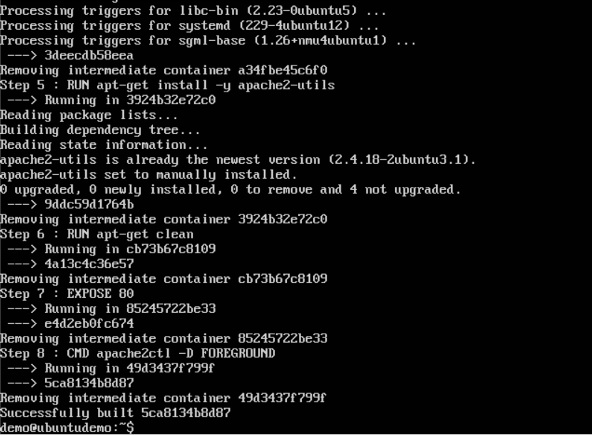


Now that the file details have been entered, just save the file.

**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.



**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.

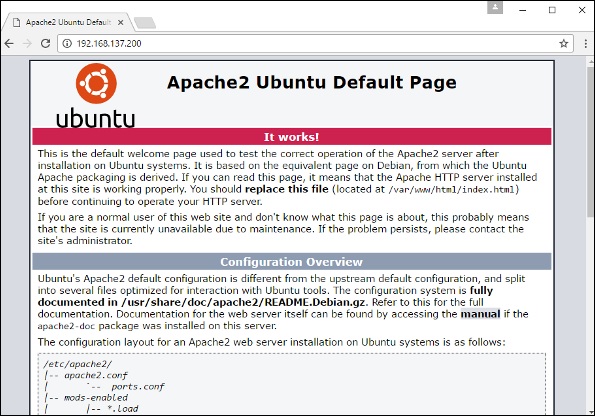
sudo docker run –d –p 80:80 mywebserver



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.



# 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.

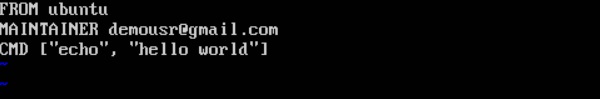
**Step 1** − Build the Docker File with the following commands −

FROM ubuntu

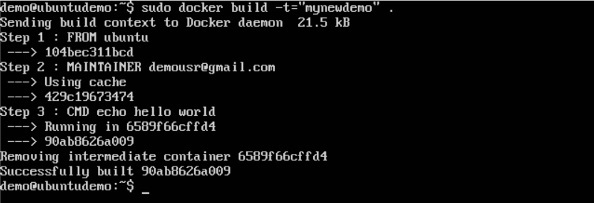
MAINTAINER demousr@gmail.com

CMD [“echo” , “hello world”]

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



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



**Step 3** − Run a container from the image.



## 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

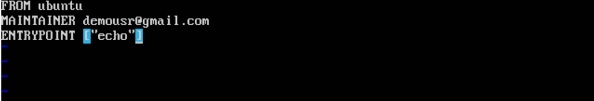
Let’s take a look at an example to understand more about ENTRYPOINT. In our example, we will enter a simple **echo** command in our Docker File and create an image and launch a container from it.

**Step 1** − Build the Docker File with the following commands −

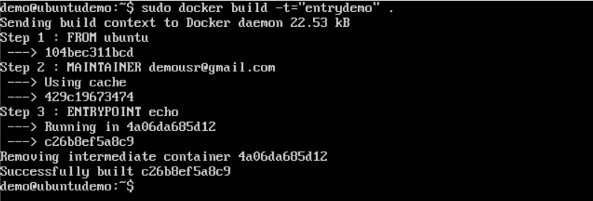
FROM ubuntu

MAINTAINER demousr@gmail.com

ENTRYPOINT [“echo”]



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



**Step 3** − Run a container from the image.



## 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

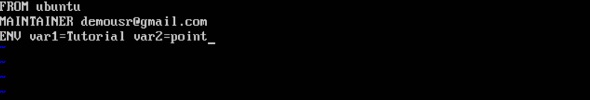
In our example, we will enter a simple **echo** command in our Docker File and create an image and launch a container from it.

**Step 1** − Build the Docker File with the following commands −

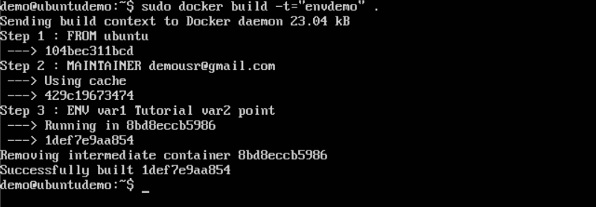
FROM ubuntu

MAINTAINER demousr@gmail.com

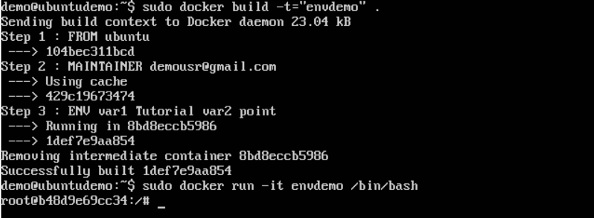
ENV var1=Tutorial var2=point



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



**Step 3** − Run a container from the image.



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



## 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.

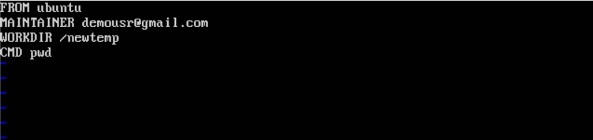
**Step 1** − Build the Docker File with the following commands −

FROM ubuntu

MAINTAINER demousr@gmail.com

WORKDIR /newtemp

CMD pwd



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



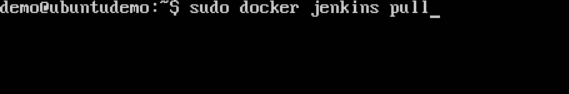
**Step 3** − Run a container from the image.



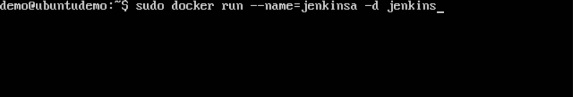
# Docker - Container Linking

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.

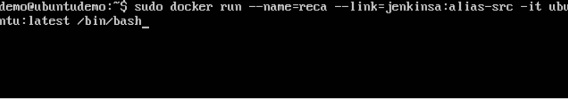
**Step 1** − Download the Jenkins image, if it is not already present, using the Jenkins **pull** command.



**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**.



**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.

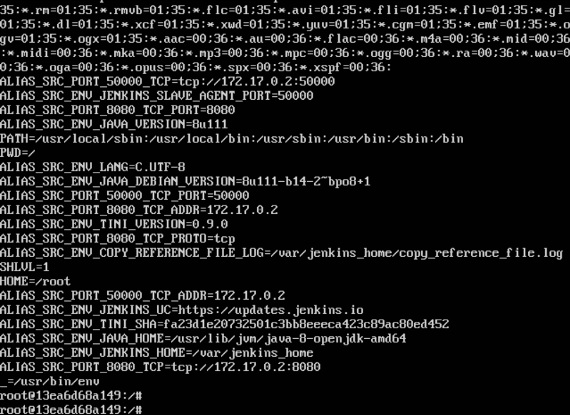


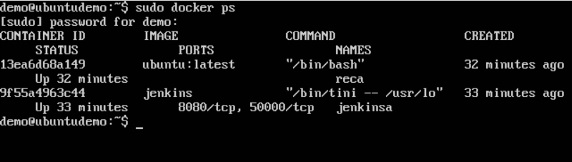
When you do a **docker ps**, you will see both the containers running.

**Step 4** − Now, attach to the receiving container.



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





# 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

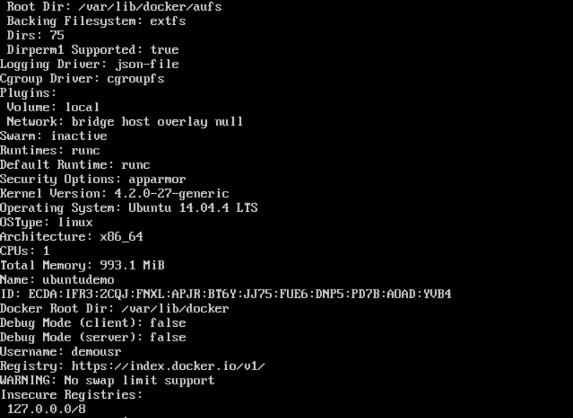
The command will provide all relative information on the Docker component installed on the Docker Host.

### 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**.



## Data Volumes

In Docker, you have a separate volume that can 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.



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**.

V Option

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.



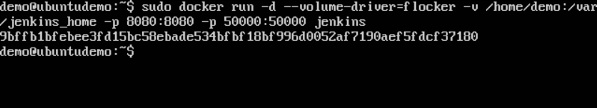
## Changing the Storage Driver for a Container

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.

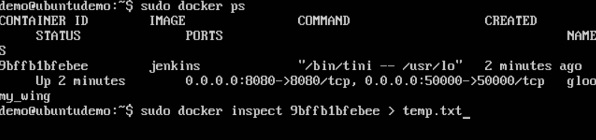


To confirm that the driver has been changed, first let’s use the **docker ps** command to see the running containers and get the container ID. So, issue the following command first −

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



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.



## 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.

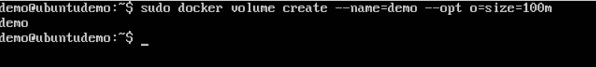
### Example

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 −



## 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

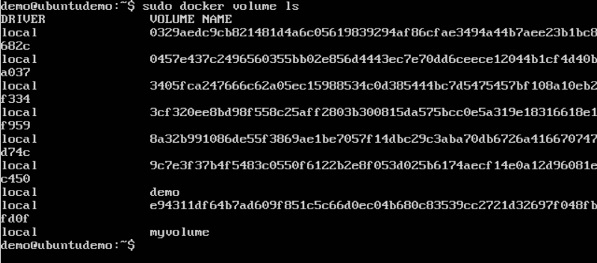
The command will output all the volumes on the **docker host**.

### Example

sudo docker volume ls

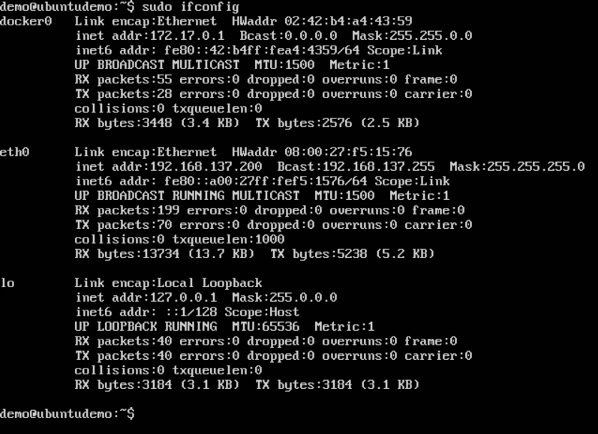
### Output

The output of the above command is shown below −



# 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.



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

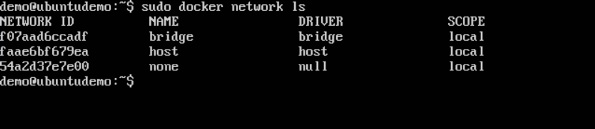
The command will output all the networks on the Docker Host.

### Example

sudo docker network ls

### Output

The output of the above command is shown below



## 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 −



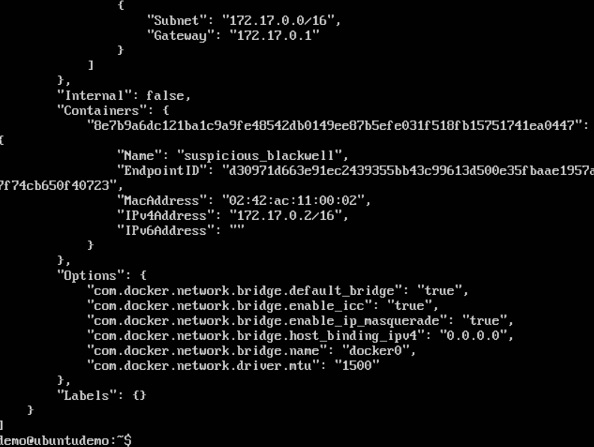
Now let’s run a container and see what happens when we inspect the network again. Let’s spin up an Ubuntu container with the following command −

sudo docker run –it ubuntu:latest /bin/bash



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



## 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

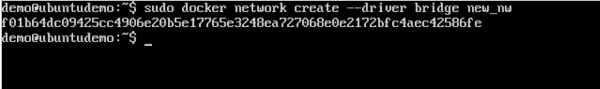
The command will output the long ID for the new network.

### Example

sudo docker network create –-driver bridge new\_nw

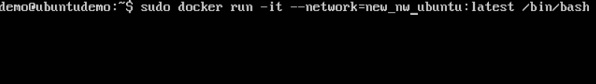
### Output

The output of the above command is shown below −



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



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



# 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.

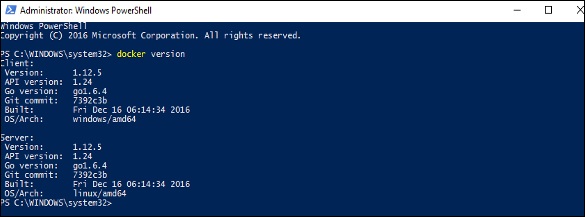
The toolbox consists of the following components −

* **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

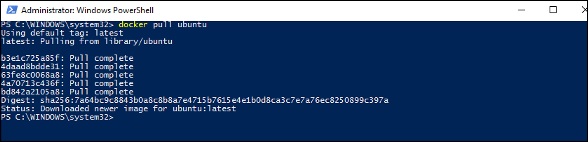
With Docker toolbox on Windows 10, you can now run Docker commands off **powershell**. If you open powershell on Windows and type in the command of Docker version, you will get all the required details about the Docker version installed.



## 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.

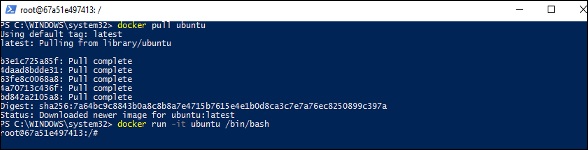
The first step is to use the Docker **pull** command to pull the Ubuntu image from Docker Hub.



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

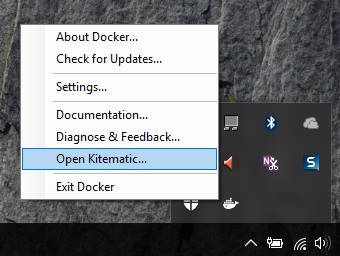
docker run –it ubuntu /bin/bash

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



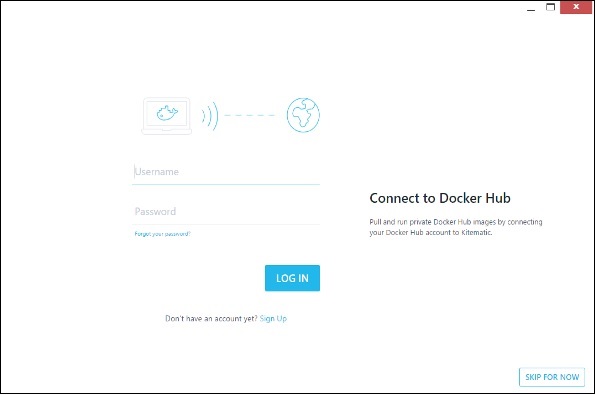
## Kitematic

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.

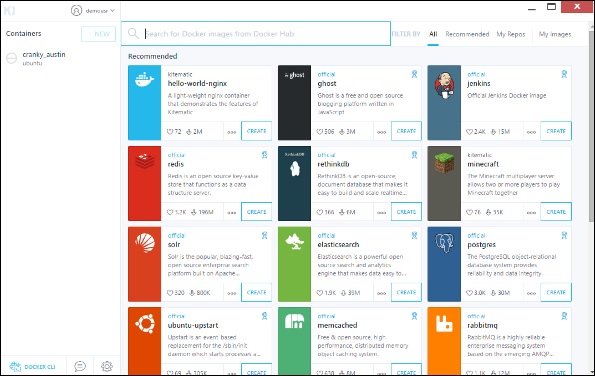


It will prompt you to download Kitematic GUI. Once downloaded, just unzip the contents. There will be a file called **Kitematic.exe**. Double-click this exe file to open the GUI interface.

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.



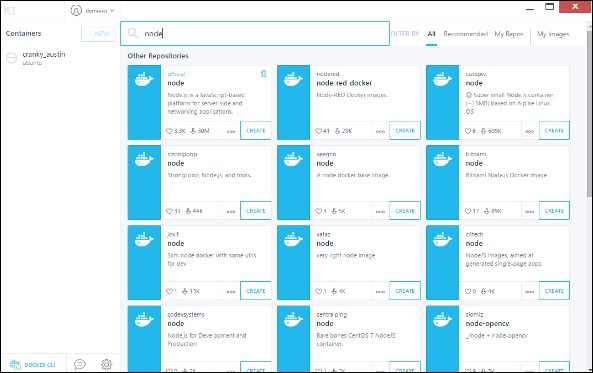
Once logged in, you will be able to see all the images downloaded on the system on the left-hand side of the interface.



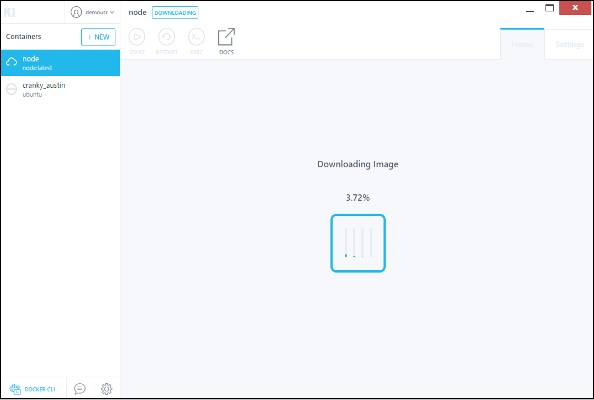
On the right-hand side, you will find all the images available on Docker Hub.

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

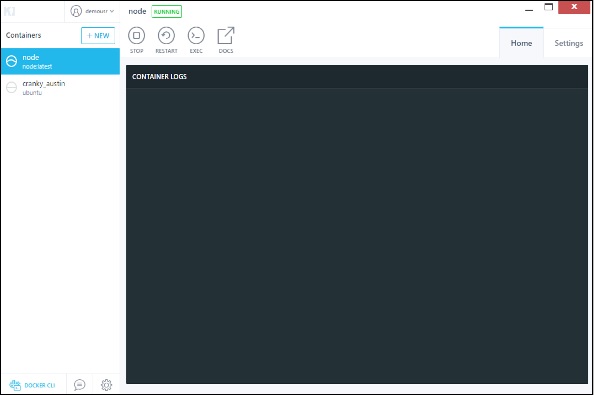
**Step 1** − Enter the keyword of node in the search criteria.



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

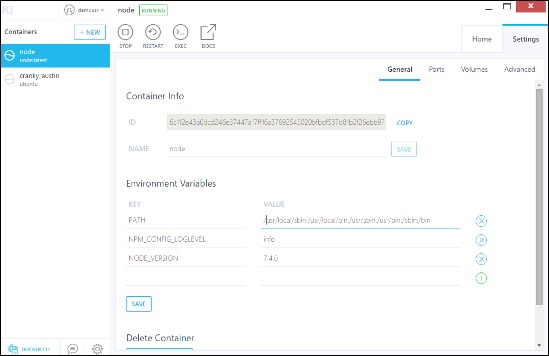


Once the image has been downloaded, it will then start running the Node container.

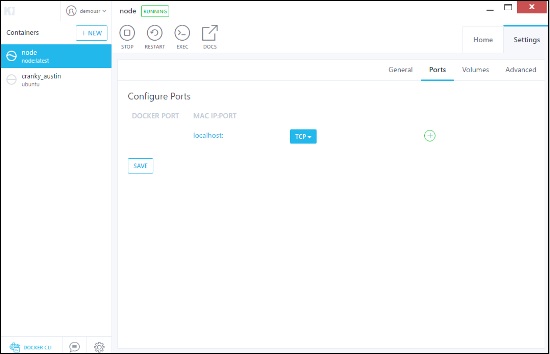


**Step 3** − If you go to the **settings** tab, you can drill-down to further settings options, as shown below.

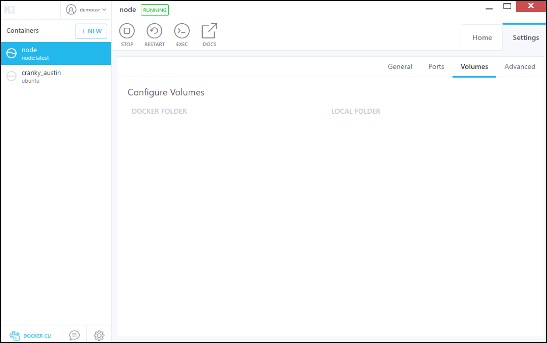
* **General settings** − In this tab, you can name the container, change the path settings, and delete the container.



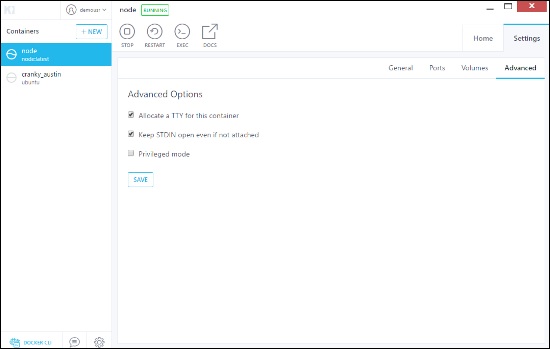
* **Ports** − Here you can see the different port mappings. If you want, you can create your own port mappings.



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



* **Advanced** − It contains the advanced settings for the container.



<https://www.tutorialspoint.com/docker/index.htm>