|  |  |
| --- | --- |
| Docker  **Docker** is a platform designed to help developers build, share, and run container applications.. |  |

Docker’s popularity has increased rapidly over the past years, and it has evolved traditional software development. Docker’s containers allow for the immense economy of scale and have made development scalable, while at the same time keeping the process user-friendly. Our Docker tutorial will help you understand Docker containers, and its benefits, and will also help learn the ways to build docker environment and docker commands. So start learning now to know everything about docker - from its advantages to how it is different from other virtual machines, know how to install it and master several docker technologies.

|  |  |
| --- | --- |
|  |  |

# An Introduction to Docker

### Docker is an open platform that enables users to develop, ship, and run applications with ease. Docker software is packaged as containers- a docker standardized unit. These containers have all the elements, such as system tools, libraries, runtime, etc required by the software. In this lesson, you will learn all the basics of Docker and how to set it up along with a guide on choosing the right plan as per your needs.

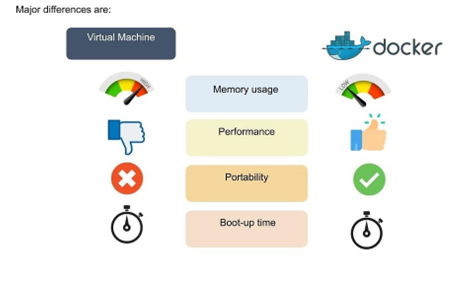
## **What Is Docker?**

When getting started with Docker, we first need to understand Docker. Docker is an OS virtualized software platform that allows IT organizations to quickly create, deploy, and run applications in Docker containers, which have all the dependencies within them. The container itself is a very lightweight package with all the instructions and dependencies—such as frameworks, libraries, and bins—within it.

The Docker container can be moved from environment to environment very easily. In a [DevOps life cycle](https://www.simplilearn.com/steps-to-building-momentum-for-devops-article), Docker really shines when used for deployment. When you deploy your solution, you want to guarantee that the code tested will actually work in the production environment. In addition, when you're building and testing the code, it's beneficial to have a container running the solution at those stages because you can validate your work in the same environment used for production.

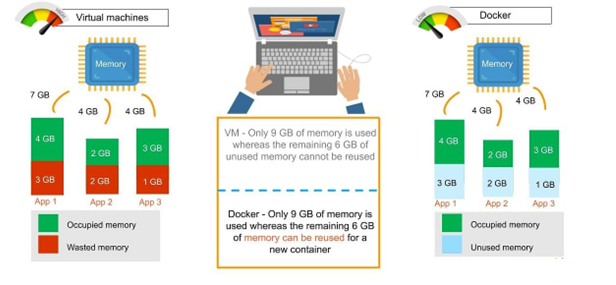
You can use Docker throughout multiple stages of your DevOps cycle, but it is especially valuable in the deployment stage, especially since it allows developers to use rapid deployment. In addition, the environment itself is highly portable and was designed with efficiencies that will enable you to run multiple Docker containers in a single environment, unlike traditional virtual machine environments.

## **Docker vs Virtual Machines**



In the image, you’ll notice some major differences, including:

* The virtual environment has a hypervisor layer, whereas Docker has a Docker engine layer.
* There are additional layers of libraries within the virtual machine, each of which compounds and creates very significant differences between a Docker environment and a virtual machine environment.
* With a virtual machine, the memory usage is very high, whereas, in a Docker environment, memory usage is very low.
* In terms of performance, when you start building out a virtual machine, particularly when you have more than one virtual machine on a server, the performance becomes poorer. With Docker, the performance is always high because of the single Docker engine.
* In terms of portability, virtual machines just are not ideal. They’re still dependent on the host operating system, and a lot of problems can happen when you use virtual machines for portability. In contrast, Docker was designed for portability. You can actually build solutions in a Docker container, and the solution is guaranteed to work as you have built it no matter where it’s hosted.
* The boot-up time for a virtual machine is fairly slow in comparison to the boot-up time for a Docker environment, in which boot-up is almost instantaneous.



* One of the other challenges of using a virtual machine is that if you have unused memory within the environment, you cannot reallocate it. If you set up an environment that has 9 gigabytes of memory, and 6 of those gigabytes are free, you cannot do anything with that unused memory. With Docker, if you have free memory, you can reallocate and reuse it across other containers used within the Docker environment.
* Another challenge of virtual machines is that running multiples of them in a single environment can lead to instability and performance issues. Docker, on the other hand, is designed to run multiple containers in the same environment—it actually gets better with more containers run in that hosted single Docker engine.
* Virtual machines have portability issues; the software can work on one machine, but if you move that virtual machine to another machine, suddenly some of the software won’t work, because some dependencies will not be inherited correctly. Docker is designed to be able to run across multiple environments and to be deployed easily across systems.
* The boot-up time for a virtual machine is about a few minutes, in contrast to the milliseconds it takes for a Docker environment to boot up.

## **Advantages of Docker**

Next in the getting started with docker tutorial we focus on the advantages of Docker. As noted previously, you can do rapid deployment using Docker. The environment itself is highly portable and was designed with efficiencies that allow you to run multiple Docker containers in a single environment, unlike traditional virtual machine environments.

The configuration itself can be scripted through a language called YAML, which allows you to describe the Docker environment you want to create. This, in turn, allows you to scale your environment quickly. But probably the most critical advantage these days is security.

You have to ensure that the environment you’re running is highly secure yet highly scalable, and Docker takes security very seriously. You’ll see it as one of the key components of the agile architecture of the system you’re implementing.

Now that you know the advantages of Docker, the next thing you need to know in this getting started with docker tutorial is how it works and its components.

## **How Does Docker Work?**

Docker works via a Docker engine that is composed of two key elements: a server and a client; and the communication between the two is via REST API. The server communicates the instructions to the client. On older Windows and Mac systems, you can take advantage of the Docker toolbox, which allows you to control the Docker engine using Compose and Kitematic.

Now that we have learned about Docker, it's advantages, and how it works, our next focus in this getting started with docker tutorial is to learn the various components of Docker.

## **Components of Docker**

There are four components that we will discuss in this getting started with docker tutorial:

* Docker client and server
* Docker image
* Docker registry
* Docker container

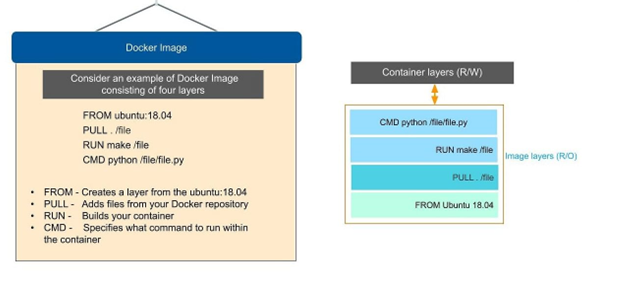
### **Docker Client and Server**

This is a command-line-instructed solution in which you would use the terminal on your Mac or Linux system to issue commands from the Docker client to the Docker daemon. The communication between the Docker client and the Docker host is via a REST API. You can issue similar commands, such as a Docker Pull command, which would send an instruction to the daemon and perform the operation by interacting with other components (image, container, registry). The Docker daemon itself is actually a server that interacts with the operating system and performs services. As you’d imagine, the Docker daemon constantly listens across the REST API to see if it needs to perform any specific requests. If you want to trigger and start the whole process, you’ll need to use the Dockered command within your Docker daemon, which will start all of your performances. Then you have a Docker host, which lets you run the Docker daemon and registry.

Now let’s talk about the actual structure of a Docker image in this getting started with docker tutorial. A Docker image is a template that contains instructions for the Docker container. That template is written in a language called YAML, which stands for Yet Another Markup Language.

### **Docker Image**

Next in the getting started with docker tutorial, we will learn all about Docker Image. The Docker image is built within the YAML file and then hosted as a file in the Docker registry. The image has several key layers, and each layer depends on the layer below it. Image layers are created by executing each command in the Dockerfile and are in the read-only format. You start with your base layer, which will typically have your base image and your base operating system, and then you will have a layer of dependencies above that. These then comprise the instructions in a read-only file that would become your Dockerfile.



Here we have four layers of instructions: From, Pull, Run and CMD. What does it actually look like? The From command creates a layer based on Ubuntu, and then we add files from the Docker repository to the base command of that base layer.

* Pull: Adds files from your Docker repository
* Run: Builds your container
* CMD: Specifies which command to run within the container

In this instance, the command is to run Python. One of the things that will happen as we set up multiple containers is that each new container adds a new layer with new images within the Docker environment. Each container is completely separate from the other containers within the Docker environment, so you can create your own separate read-write instructions within each layer. What’s interesting is that if you delete a layer, the layer above it will also get deleted.

What happens when you pull in a layer but something changes in the core image? Interestingly, the main image itself cannot be modified. Once you’ve copied the image, you can modify it locally. You can never modify the actual base image.

### **Docker Registry**

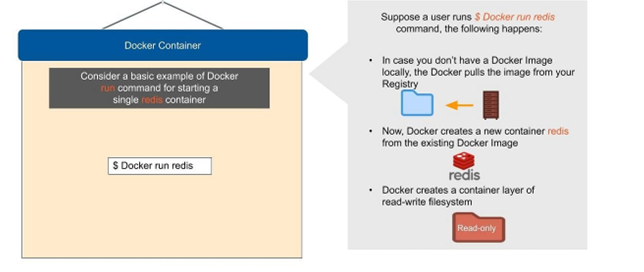
The Docker registry is where you would host various types of images and where you would distribute the images from. The repository itself is just a collection of Docker images, which are built on instructions written in YAML and are very easily stored and shared. You can give the Docker images name tags so that it’s easy for people to find and share them within the Docker registry. One way to start managing a registry is to use the publicly accessible Docker hub registry, which is available to anybody. You can also create your own registry for your own use internally.

The registry that you create internally can have both public and private images that you create. The commands you would use to connect the registry are  Push and Pull. Use the Push command to push a new container environment you’ve created from your local manager node to the Docker registry, and use a PullL command to retrieve new clients (Docker image) created from the Docker registry. Again, a Pull command pulls and retrieves a Docker image from the Docker registry, and a Push command allows you to take a new command that you’ve created and push it to the registry, whether it’s Docker hub or your own private registry.

### **Docker Container**

The Docker container is an executable package of applications and its dependencies bundled together; it gives all the instructions for the solution you’re looking to run. It’s really lightweight due to the built-in structural redundancy. The container is also inherently portable. Another benefit is that it runs completely in isolation. Even if you are running a container, it’s guaranteed not to be impacted by any host OS securities or unique setups, unlike with a virtual machine or a non containerized environment. The memory for a Docker environment can be shared across multiple containers, which is really useful, especially when you have a virtual machine that has a defined amount of memory for each environment.

The container is built using Docker images, and the command to run those images is Run. Let’s go through the basic steps of running a Docker image in this getting started with docker tutorial.



Consider a basic example of Docker run command for starting a single container called redis:

$ Docker run redis

If you don’t have the Redis image locally installed, it will be pulled from the registry. After this, the new Docker container Redis will be available within your environment so you can start using it.

Now let’s look at why containers are so lightweight. It’s because they do not have some of the additional layers that virtual machines do. The biggest layer Docker doesn’t have is the hypervisor, and it doesn’t need to run on a host operating system.

Now that you know the basic Docker components, let’s now look into advanced Docker components in this getting started with docker tutorial.

## **Advanced Docker Components**

After going through the various components of Docker, the next focus of this Docker tutorial are the advanced components of Docker:

* Docker compose
* Docker swamp

### **Docker Compose**

Docker compose is designed for running multiple containers as a single service. It does so by running each container in isolation but allowing the containers to interact with one another. As noted earlier, you would write the compose environments using YAML.

So in what situations might you use Docker compose? An example would be if you are running an Apache server with a single database and you need to create additional containers to run additional services without having to start each one separately. you would write a set of files using Docker compose to do that.

### **Docker Swarm**

Docker swarm is a service for containers that allows IT administrators and developers to create and manage a cluster of swarm nodes within the Docker platform. Each node of Docker swarm is a Docker daemon, and all Docker daemons interact using the Docker API. A swarm consists of two types of nodes: a manager node and a worker node. A manager node maintains cluster management tasks. Worker nodes receive and execute tasks from the manager node.

After having looked into all the components of Docker, let us advance our learning in this getting started with docker tutorial on the Docker commands and use case.

## 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

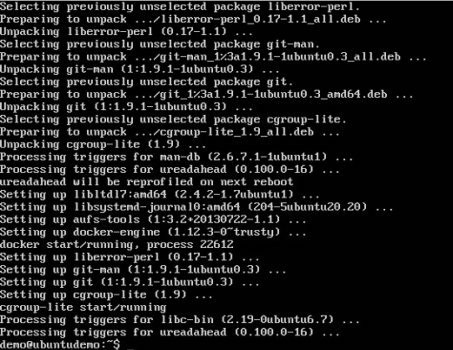
The output will provide the various details of the Docker version installed on the system.

###### 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.

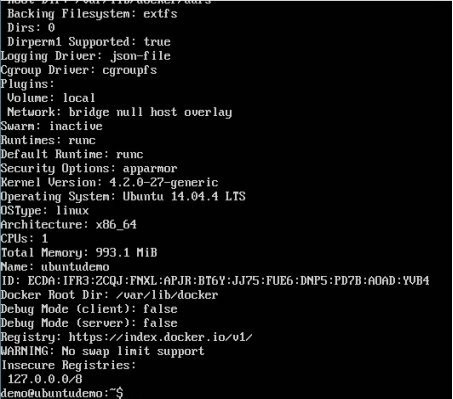
###### Return Value

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/](https://docs.docker.com/docker-for-windows/)

## Docker ToolBox

Docker ToolBox has been designed for older versions of Windows, such as Windows

* 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>

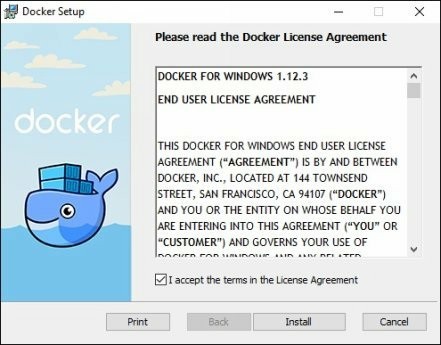
# Docker - Installation

Let’s go through the installation of each product.

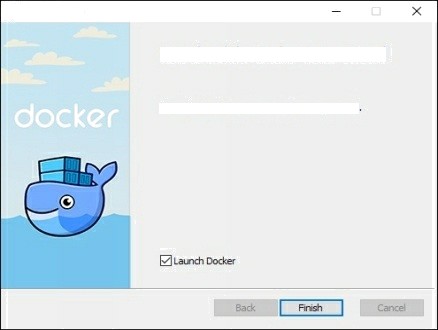
## Docker for Windows

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.



DmCnSrtup

Completed the Dodger Sdup Wtzard

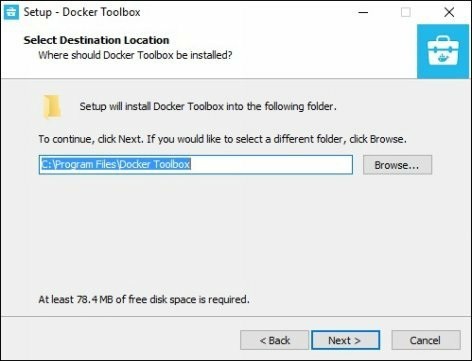
## Docker ToolBox

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

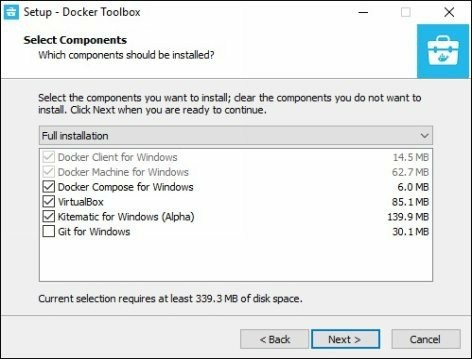
**Step 1** − Click the Next button on the start screen.



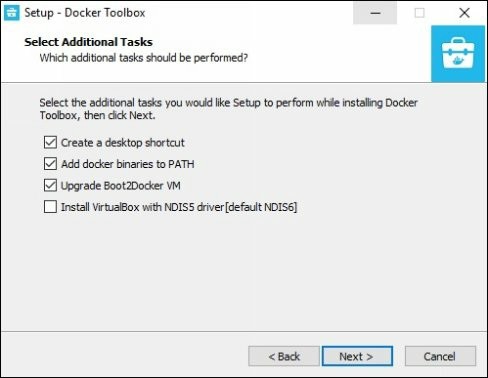
**Step 2** − Keep the default location on the next screen and click the Next button.



**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.



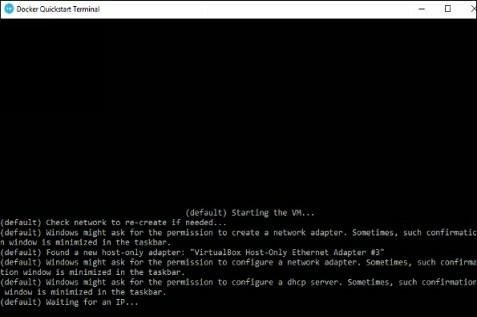
**Step 5** − On the final screen, click the Install button.



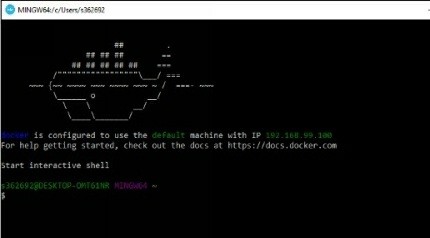
## Working with Docker Toolbox

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.

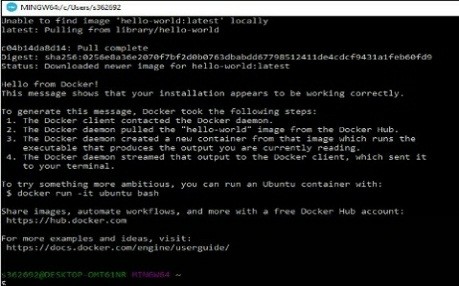
###### Example

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.

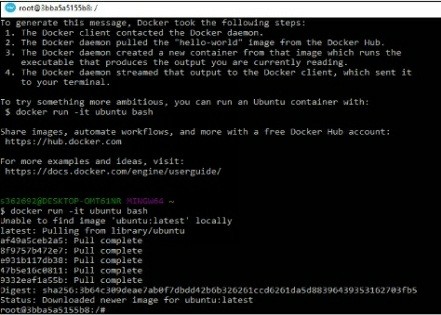
###### Output

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

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](https://www.docker.com/community-edition%23/add_ons)

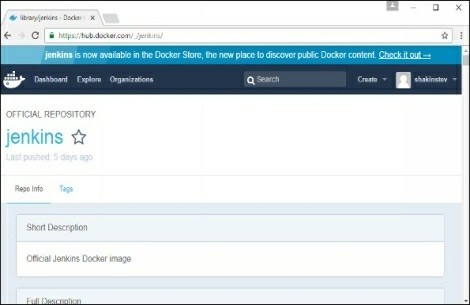
**Step 1** − First you need to do a simple sign-up on Docker hub.



**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 pullcommand. This will be used to download the Jenkins image onto the local Ubuntu server.



n your dosJgr ard rolp yau find era g ono is worâaJg m u m

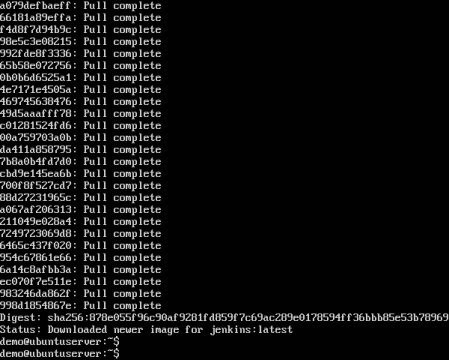
##### Documentation





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

sudo docker pull jenkins

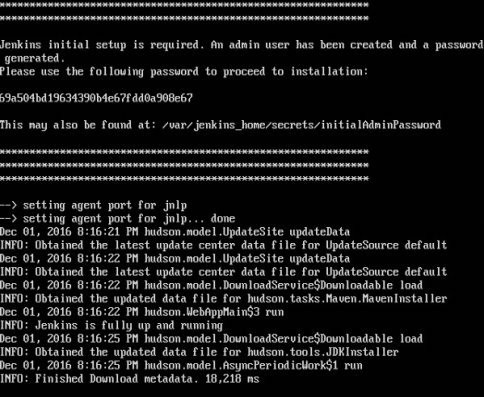


To run Jenkins, you need to run the following command −

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 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 rootaccess.
    - 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

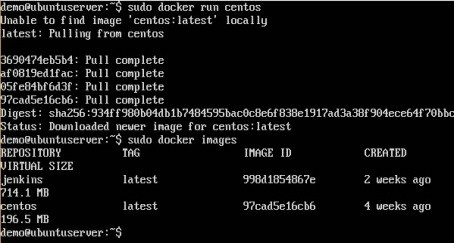
This command will download the centos image, if it is not already present, and run the OS as a container.

###### 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.

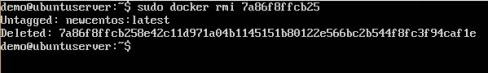
###### Example

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

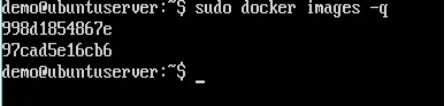
The output will show only the Image ID’s of the images on the Docker host.

###### 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 - Containers

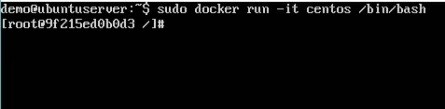
Containers are instances of Docker images that can be run using the Docker run command. The basic purpose of Docker is to run containers. Let’s discuss how to work with containers.

## 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

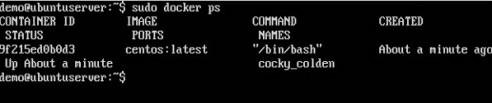
None

###### 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.

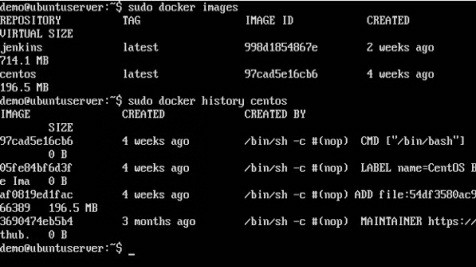
###### Example

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 - 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.

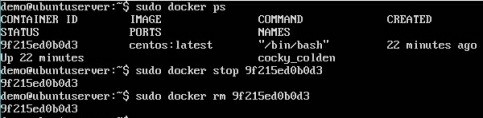
###### Example

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.

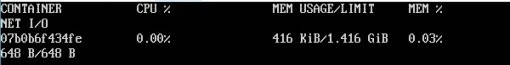
###### Example

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

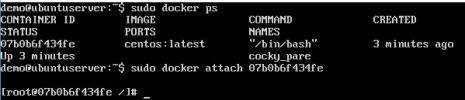
###### Example

sudo docker attach 07b0b6f434fe

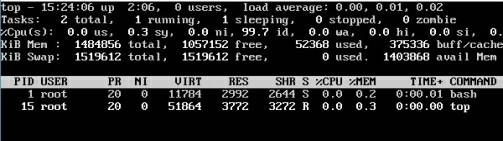
The above command will attach to the Docker container 07b0b6f434fe.

###### 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.

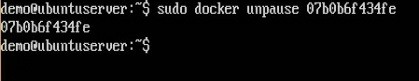
###### Example

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.

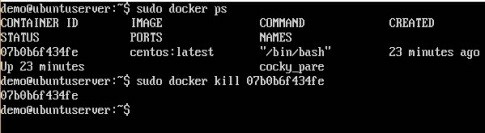
###### Example

sudo docker kill 07b0b6f434fe

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

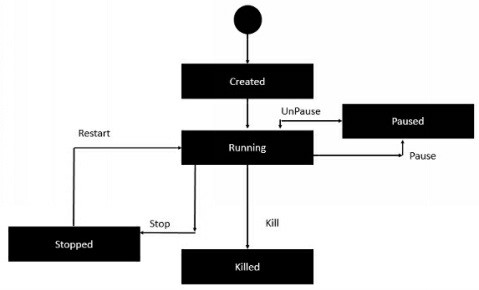
###### Output

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



## Docker – Container Lifecycle

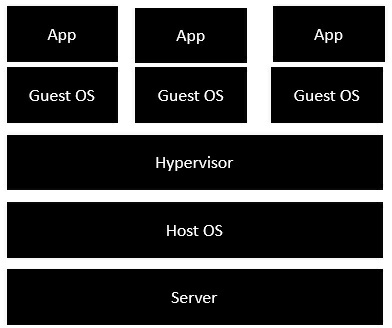
The following illustration explains the entire lifecycle of a Docker container.



* + - 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.

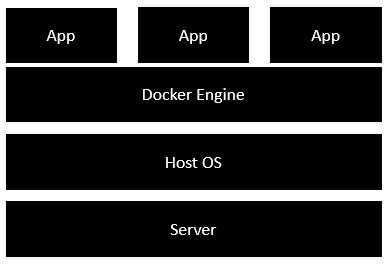
# Docker - Architecture

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.

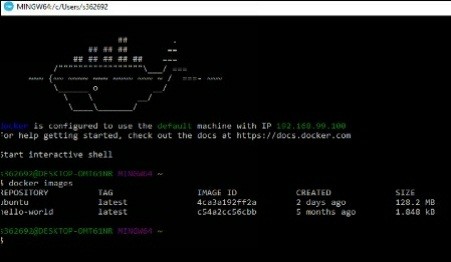
The clear advantage in this architecture is that you don’t need to have extra hardware for Guest OS. Everything works as Docker containers.

# 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

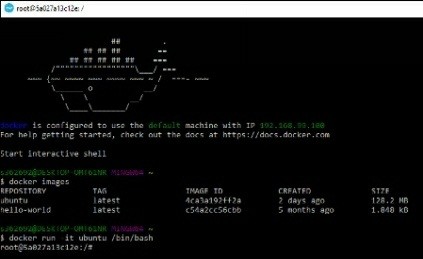
Let’s run the Docker images command on the Windows host.



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

## Running a Container

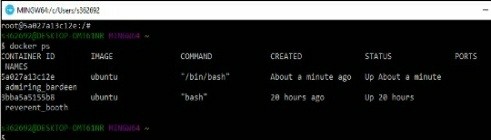
Now let’s run a container in the Windows Docker host.



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

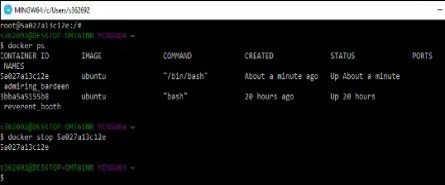
## Listing All Containers

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

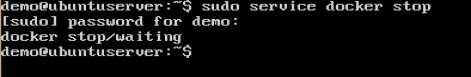
A message showing that the Docker process has stopped.

###### 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.

###### Synax

service docker start

###### Options

None

###### Return Value

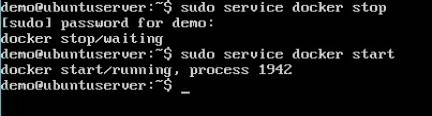
A message showing that the Docker process has started.

###### Example

sudo service docker start

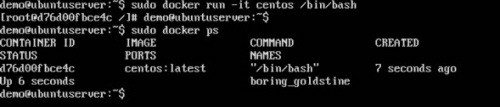
###### Output

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



# Docker - Containers and Shells

By default, when you launch a container, you will also use a shell commandwhile 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

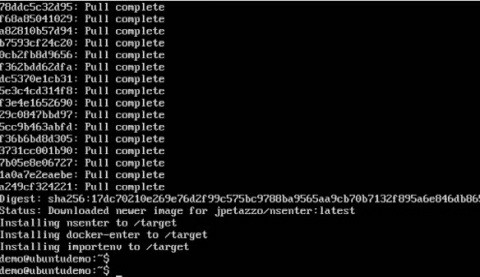
We used this command to create a new container and then used the Ctrl+P+Q command to exit out of the container. It ensures that the container still exists even after we exit from the container.

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.

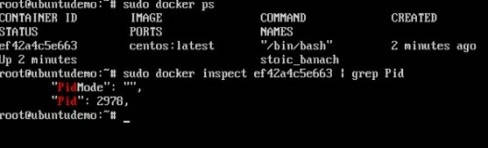
Now there is an easier way to attach to containers and exit them cleanly without the need of destroying them. One way of achieving this is by using the nsenter command.

Before we run the nsenter command, you need to first install the nsenterimage. 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.

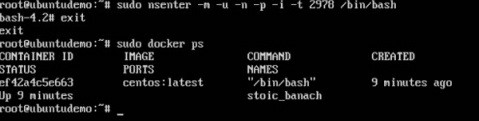
###### Return Value

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 FROM ubuntu

[MAINTAINER demousr@gmail.com](mailto:MΑINTΑINERdemousr@gmαil.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.



# 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

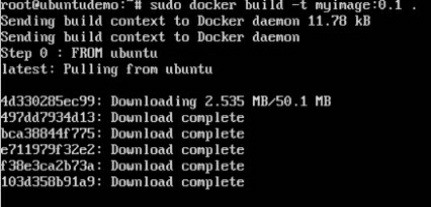
Here, myimage is the name we are giving to the Image and 0.1 is the tag number we are giving to our image.

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

command to signify the present working directory.

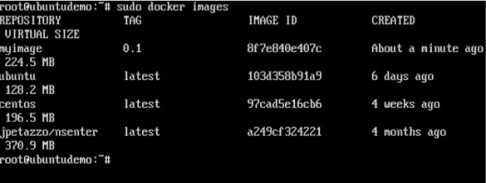
###### Output

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.

A computer screen with white text

Description automatically generated

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.

A screenshot of a computer

Description automatically generated

**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.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generatedOnce 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 myimageto 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.

A computer screen with white text

Description automatically generated

## 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

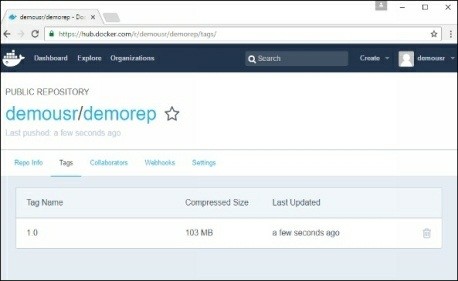
The long ID of the repository pushed to Docker Hub.

###### 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.

A screenshot of a computer

Description automatically generated

**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.

A screenshot of a computer

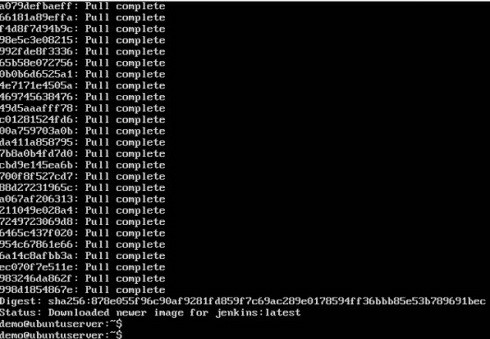
Description automatically generated

**Step 4** − If you scroll down on the same page, you can see the Docker pullcommand. 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

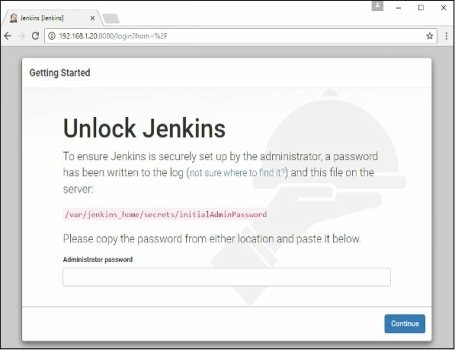
The output of the inspect command gives a JSON output. If we observe the output, we can see that there is a section of "ExposedPorts" and see that there are two ports mentioned. One is the data port of 8080 and the other is the control port of 50000.

To run Jenkins and map the ports, you need to change the Docker runcommand 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.



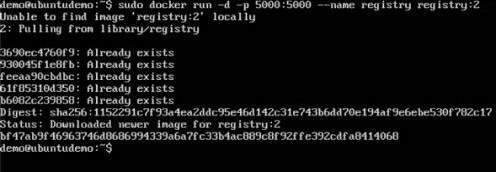
# Docker - Private Registries

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.

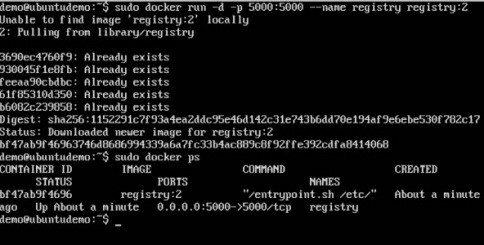
**Step 1** − Use the Docker run command to download the private registry. This can be done using the following command.

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.

A computer screen with white text

Description automatically generated

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

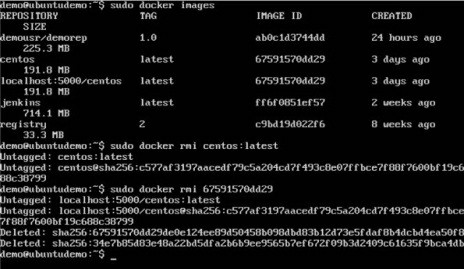
sudo docker push localhost:5000/centos

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



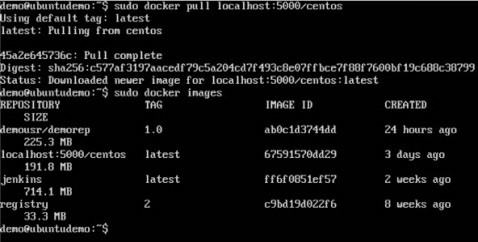
**Step 5** − Now let’s delete the local images we have for centos using the docker rmi commands. We can then download the required centos image from our private repository.

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.

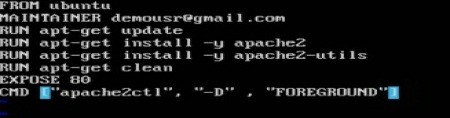
FROM ubuntu RUN apt-get update

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 −

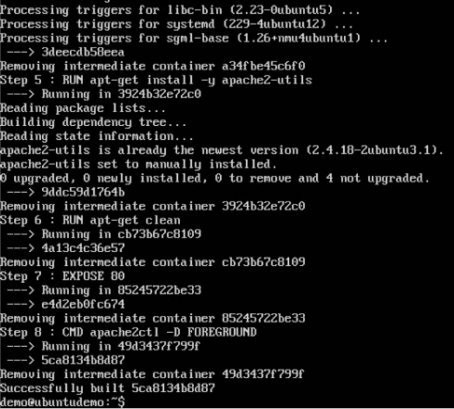
* + - We are first creating our image to be from the Ubuntu base image.
    - Next, we are going to use the RUN command to update all the packages on the Ubuntu system.
    - Next, we use the RUN command to install apache2 on our image.
    - Next, we use the RUN command to install the necessary utility 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 –pcommand, 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](mailto:MΑINTΑINERdemousr@gmαil.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](mailto:MΑINTΑINERdemousr@gmαil.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](mailto:MΑINTΑINERdemousr@gmαil.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

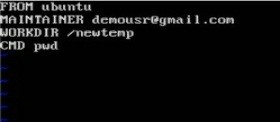
In our example, we will enter a simple echo command in our Docker File and create an image and launch a container from it.

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

FROM ubuntu

[MAINTAINER demousr@gmail.com](mailto:MΑINTΑINERdemousr@gmαil.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.



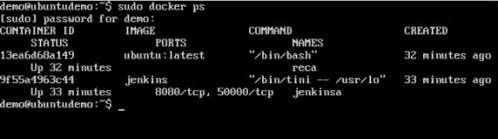
**Step 3** − Next, it is time to launch the destination container, but this time, we will link it with our source container. For our destination container, we will use the standard Ubuntu image.

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

## Btrfs

* 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.
  + 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

## ZFS

* 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.
  + 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

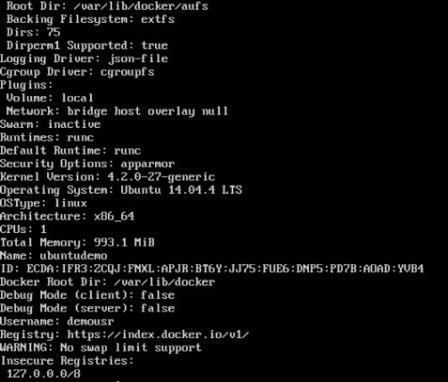
The command will provide all relative information on the Docker component installed on the Docker Host.

###### 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.

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.



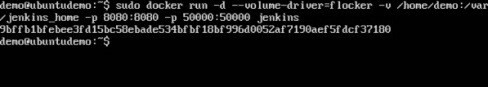
## Changing the Storage Driver for a Container

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 pscommand 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 −

###### Synt a x

docker volume ls

###### Options

None

###### Return V a lue

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

###### Ex a mple

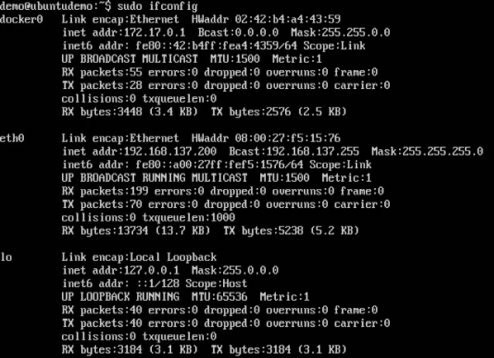
sudo docker volume ls

###### 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

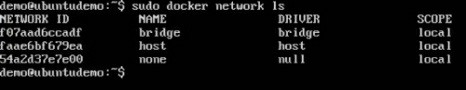
The command will output all the networks on the Docker Host.

###### 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 - Setting Node.js

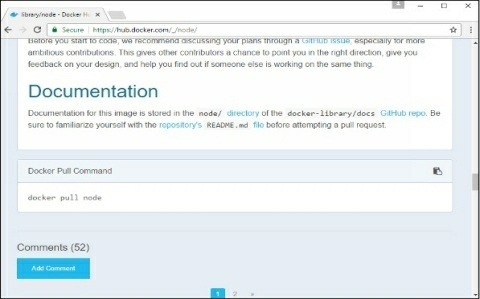
Node.js is a JavaScript framework that is used for developing server-side applications. It is an open source framework that is developed to run on a variety of operating systems. Since Node.js is a popular framework for development, Docker has also ensured it has support for Node.js applications.

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.

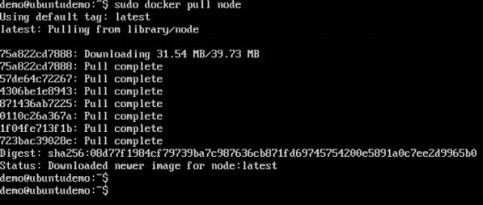


**Step 2** − You will see that the Docker pull command for node in the details of the repository in Docker Hub.



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

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



**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.

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.



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-node -v "C:/Personal/D Drive/CG/Java:/usr/src/app" -w /usr/src/app node node HelloWorld.js

HelloWorld.js

// Import the HTTP module

const http = require('http');

// Create an HTTP server

http.createServer(function (req, res) {

// Send the HTTP header

// HTTP Status: 200 : OK

// Content Type: text/plain

res.writeHead(200, {'Content-Type': 'text/plain'});

// Send the response body as "node js hello world !!!"

res.end('node js hello world !!! \n');

}).listen(8000);

// Console will print this message

console.log('Server is running live at http://127.0.0.1:8000/');

console.log('node js hello world');

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.

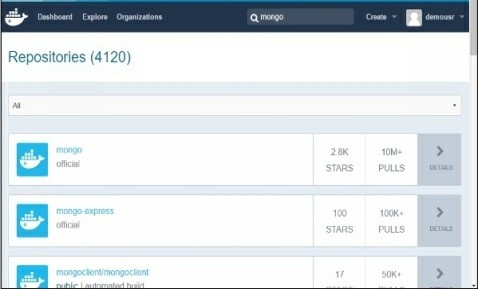
We will then get the following output. And from the output, we can clearly see that the Node container ran as a container and executed the HelloWorld.js script.

# Docker - Setting MongoDB

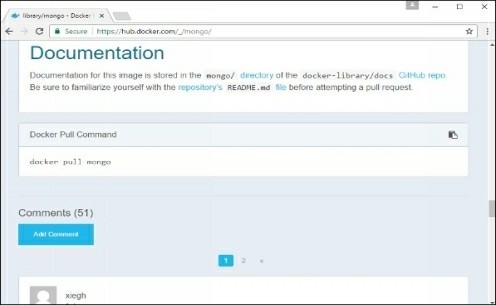
MongoDB is a famous document-oriented database that is used by many modern- day web applications. Since MongoDB is a popular database for development, Docker has also ensured it has support for MongoDB.

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

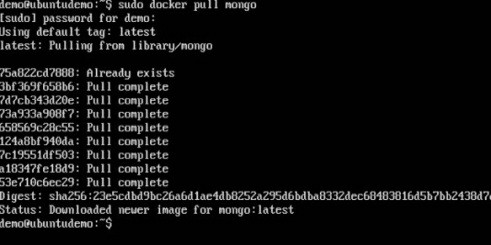
below. Just type in Mongo in the search box and click on the Mongo (official) link which comes up in the search results.

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



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





**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 run -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 −



Take a note of the following points −

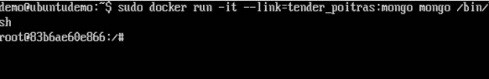
* + The name of the container is tender\_poitras. This name will be different since the name of the containers keep on changing when you spin up a container. But just make a note of the container which you have launched.
  + Next, also notice the port number it is running on. It is listening on the TCP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

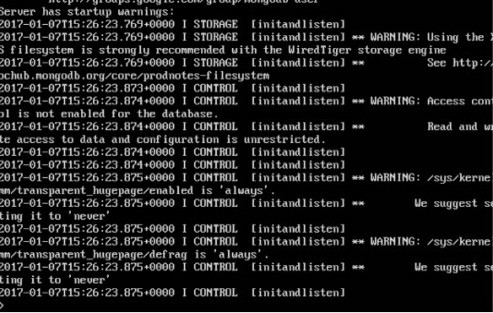
sudo docker run –it –link=tender\_poitras:mongo mongo /bin/bash

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.



You will now be in the new container.

**Step 6** − Run the env command in the new container to see the details of how to connect to the MongoDB server container.

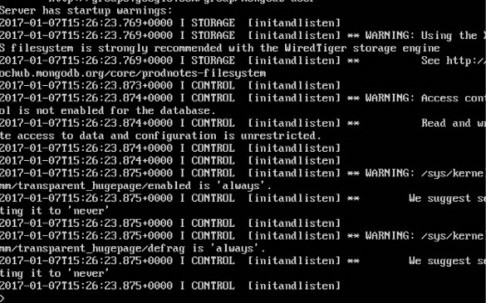
**Step 7** − Now it’s time to connect to the MongoDB server from the client container. We can do this via the following command −

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 envcommand.

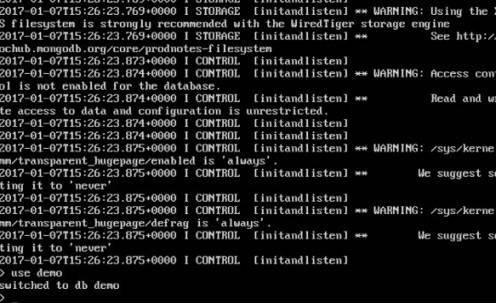
Once you run the command, you will then be connected to the MongoDB database.



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.



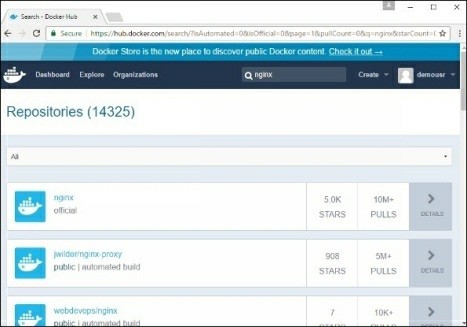
Now you have successfully created a client and server MongoDB container.

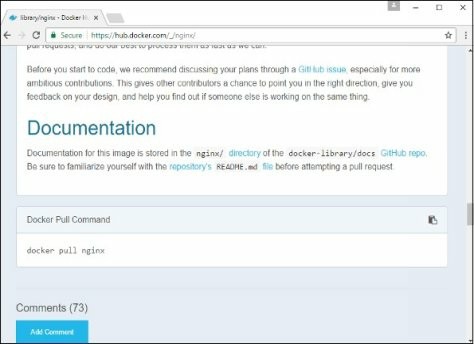
# Docker - Setting NGINX

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 nginxup 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.



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

A computer screen with white text

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

**Step 4** − Now let’s run the nginx container via the following command.

sudo docker run –p 8080:80 –d nginx

A black background with white text

Description automatically generatedWe are exposing the port on the nginx server which is port 80 to the port 8080 on the Docker Host.

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 ngnix 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



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



Let’s then run the following Docker command.

sudo docker run –p 8080:80 –v

“$PWD”:/usr/share/nginx/html:ro –d nginx

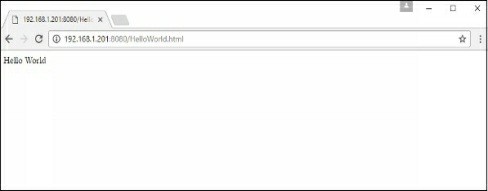
The following points need to be noted about the above command −

* + 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.



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



# 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



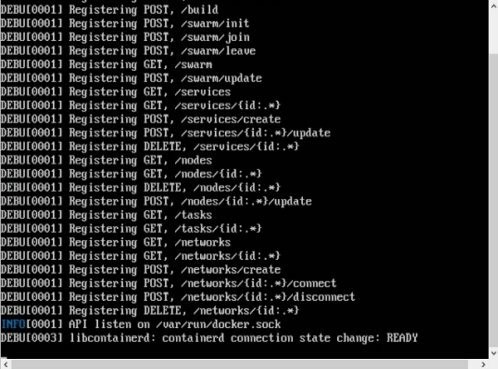
**Step 2** − Now we need to start the docker daemon process. But this time, we need to append the –l parameter to specify the logging option. So let’s issue the following command when starting the docker daemon process.

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.

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



A computer screen with white text

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

## 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

A black background with white text

Description automatically generated

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 6bfb1271fcdd

###### Output

A screenshot of a computer

Description automatically generated

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

# Docker - Compose

[Docker](https://www.docker.com/) provides lightweight containers to run services in isolation from our infrastructure so we can deliver software quickly. Docker Compose is used to run multiple containers as a single service. For example, suppose you had an application which required NGNIX 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 NGNIX 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/dockercompose

-$(uname -s) -$(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/.

A computer screen with white text

Description automatically generated

**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

A computer screen with white text

Description automatically generated

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

###### Synax

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 −

A computer screen with white text

Description automatically generated

## 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

A black screen with white text

Description automatically generated

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 −

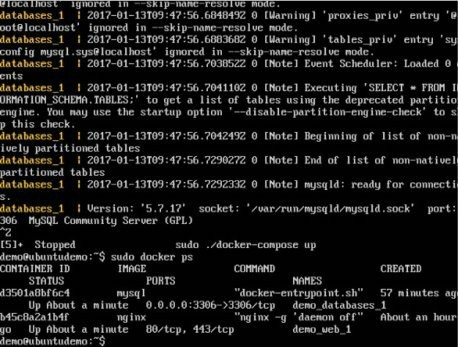
sudo ./docker-compose up

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

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

A computer screen shot of a computer program

Description automatically generated

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

## Overview

Assume that we have a Spring Boot Application working with MySQL database.  
The problem is to containerize a system that requires more than one Docker container:

* Spring Boot for Rest API
* MySQL for database

Docker Compose helps us setup the system more easily and efficiently than with only Docker. We’re gonna following these steps:

* Create Spring Boot App working with MySQL database.
* Create Dockerfile for Spring Boot App.
* Write Docker Compose configurations in YAML file.
* Set Spring Boot Docker Compose Environment variables.
* Run the system.

Using the code base above, we put the Spring Boot project in **bezkoder-app** folder without the need of **resources**/application.properties. It is because Environment variables will be exported to .env file.

## Create Dockerfile for Spring Boot App

Dockerfile defines a list of commands that Docker uses for setting up the Spring Boot application environment. So we put the file in **bezkoder-app** folder.

Because we will use Docker Compose, we won’t define all the configuration commands in this Dockerfile.

**bezkoder-app**/Dockerfile

# FROM maven:3.8.2-jdk-8 # for Java 8

FROM maven:3.8.5-openjdk-17

WORKDIR /bezkoder-app

COPY . .

RUN mvn clean install

CMD mvn spring-boot:run

Let me explain some points:

* FROM: install the image of the Maven – JDK version.
* WORKDIR: path of the working directory.
* COPY: copy all the files inside the project directory to the container.
* RUN: execute a command-line inside the container: mvn clean install to install the dependencies in pom.xml.
* CMD: run script mvn spring-boot:run after the image is built.

## Write Docker Compose configurations

On the root of the project directory, we’re gonna create the docker-compose.yml file. Follow [version 3](https://docs.docker.com/compose/compose-file/compose-file-v3/) syntax defined by Docker:

version: '3.8'

services:

mysqldb:

app:

volumes:

* version: Docker Compose file format version will be used.
* services: individual services in isolated containers. Our application has two services: app (Spring Boot) and mysqldb (MySQL database).
* [volumes](https://docs.docker.com/storage/volumes/): named volumes that keeps our data alive after restart.

Let’s implement the details.

docker-compose.yml

version: "3.8"

services:

mysqldb:

image: mysql:5.7

restart: unless-stopped

env\_file: ./.env

environment:

- MYSQL\_ROOT\_PASSWORD=$MYSQLDB\_ROOT\_PASSWORD

- MYSQL\_DATABASE=$MYSQLDB\_DATABASE

ports:

- $MYSQLDB\_LOCAL\_PORT:$MYSQLDB\_DOCKER\_PORT

volumes:

- db:/var/lib/mysql

app:

depends\_on:

- mysqldb

build: ./bezkoder-app

restart: on-failure

env\_file: ./.env

ports:

- $SPRING\_LOCAL\_PORT:$SPRING\_DOCKER\_PORT

environment:

SPRING\_APPLICATION\_JSON: '{

"spring.datasource.url" : "jdbc:mysql://mysqldb:$MYSQLDB\_DOCKER\_PORT/$MYSQLDB\_DATABASE?useSSL=false",

"spring.datasource.username" : "$MYSQLDB\_USER",

"spring.datasource.password" : "$MYSQLDB\_ROOT\_PASSWORD",

"spring.jpa.properties.hibernate.dialect" : "org.hibernate.dialect.MySQL5InnoDBDialect",

"spring.jpa.hibernate.ddl-auto" : "update"

}'

volumes:

- .m2:/root/.m2

stdin\_open: true

tty: true // which can be particularly useful for running interactive processes or debugging within the container. It enables you to attach to the container and interact with it more dynamically, similar to how you would in a standard terminal session.

volumes:

db:

– **mysqldb**:

* image: official Docker image
* restart: configure the [restart policy](https://docs.docker.com/config/containers/start-containers-automatically/#use-a-restart-policy)
* env\_file: specify our .env path that we will create later
* environment: provide setting using environment variables
* ports: specify ports will be used
* volumes: map volume folders

– **app**:

* [depends\_on](https://docs.docker.com/compose/compose-file/compose-file-v3/#depends_on): dependency order, **mysqldb** is started before **app**
* build: configuration options that are applied at build time that we defined in the Dockerfile with relative path
* environment: environmental variables that Spring Boot application uses
* stdin\_open and tty: keep open the terminal after building container

You should note that the host port (LOCAL\_PORT) and the container port (DOCKER\_PORT) is different. Networked service-to-service communication uses the container port, and the outside uses the host port.

## Docker Compose Environment variables

In the service configuration, we used environmental variables defined inside the .env file. Now we start writing it.

.env

MYSQLDB\_USER=root

MYSQLDB\_ROOT\_PASSWORD=123456

MYSQLDB\_DATABASE=bezkoder\_db

MYSQLDB\_LOCAL\_PORT=3307

MYSQLDB\_DOCKER\_PORT=3306

SPRING\_LOCAL\_PORT=6868

SPRING\_DOCKER\_PORT=8080

## Run the Spring Boot microservice with Docker Compose

We can easily run the whole with only a single command:  
docker compose up

Docker will pull the MySQL and Maven images (if our machine does not have it before).

The services can be run on the background with command:  
docker compose up -d

$ docker compose up -d

Creating network "spring-boot-mysql\_default" with the default driver

Creating volume "spring-boot-mysql\_db" with default driver

Pulling mysqldb (mysql:5.7)...

5.7: Pulling from library/mysql

e1acddbe380c: Pull complete

bed879327370: Pull complete

03285f80bafd: Pull complete

ccc17412a00a: Pull complete

1f556ecc09d1: Pull complete

adc5528e468d: Pull complete

1afc286d5d53: Pull complete

4d2d9261e3ad: Pull complete

ac609d7b31f8: Pull complete

53ee1339bc3a: Pull complete

b0c0a831a707: Pull complete

Digest: sha256:7cf2e7d7ff876f93c8601406a5aa17484e6623875e64e7acc71432ad8e0a3d7e

Status: Downloaded newer image for mysql:5.7

Building app

Sending build context to Docker daemon 22.02kB

Step 1/5 : FROM maven:3.8.2-jdk-8

---> 80704b8c5fbd

Step 2/5 : WORKDIR /bezkoder-app

---> Running in f63e76f45fcc

Removing intermediate container f63e76f45fcc

---> 10802ac64cea

Step 3/5 : COPY . .

---> 9dcd16082f00

Step 4/5 : RUN mvn clean install

---> Running in 288bea890f74

[INFO] Scanning for projects...

Downloading from central: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-starter-parent/2.2.1.RELEASE/spring-boot-starter-parent-2.2.1.RELEASE.pom

Downloaded from central: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-starter-parent/2.2.1.RELEASE/spring-boot-starter-parent-2.2.1.RELEASE.pom (8.1 kB at 4.2 kB/s)

Downloading from central: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-dependencies/2.2.1.RELEASE/spring-boot-dependencies-2.2.1.RELEASE.pom

Downloaded from central: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-dependencies/2.2.1.RELEASE/spring-boot-dependencies-2.2.1.RELEASE.pom (127 kB at 201 kB/s)

...

[INFO] Installing /bezkoder-app/target/spring-boot-data-jpa-0.0.1-SNAPSHOT.jar to /root/.m2/repository/com/bezkoder/spring-boot-data-jpa/0.0.1-SNAPSHOT/spring-boot-data-jpa-0.0.1-SNAPSHOT.jar

[INFO] Installing /bezkoder-app/pom.xml to /root/.m2/repository/com/bezkoder/spring-boot-data-jpa/0.0.1-SNAPSHOT/spring-boot-data-jpa-0.0.1-SNAPSHOT.pom

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------

[INFO] Total time: 02:41 min

[INFO] Finished at: 2021-08-18T04:10:08Z

[INFO] ------------------------------------------------------------------------

Removing intermediate container 288bea890f74

---> adddf4648410

Step 5/5 : CMD mvn spring-boot:run

---> Running in c81f8028e2eb

Removing intermediate container c81f8028e2eb

---> 1f710daedbf2

Successfully built 1f710daedbf2

Successfully tagged spring-boot-mysql\_app:latest

WARNING: Image for service app was built because it did not already exist. To rebuild this image you must use `docker compose build` or `docker compose up --build`.

Creating spring-boot-mysql\_mysqldb\_1 ... done

Creating spring-boot-mysql\_app\_1 ... done

Now you can check the current working containers:

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

5ad28f104e8b spring-boot-mysql\_app "/usr/local/bin/mvn-…" 3 minutes ago Up 3 minutes 0.0.0.0:6868->8080/tcp, :::6868->8080/tcp spring-boot-mysql\_app\_1

ba9281773e7f mysql:5.7 "docker-entrypoint.s…" 3 minutes ago Up 3 minutes 33060/tcp, 0.0.0.0:3307->3306/tcp, :::3307->3306/tcp spring-boot-mysql\_mysqldb\_1

And Docker images:

$ docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

spring-boot-mysql\_app latest 1f710daedbf2 5 minutes ago 672MB

mysql 5.7 6c20ffa54f86 6 minutes ago 448MB

maven 3.8.2-jdk-8 80704b8c5fbd 6 minutes ago 525MB

Send a HTTP request to the Spring Boot – MySQL system:

## Stop the Application

Stopping all the running containers is also simple with a single command:  
docker compose down

$ docker compose down

Stopping spring-boot-mysql\_app\_1 ... done

Stopping spring-boot-mysql\_mysqldb\_1 ... done

Removing spring-boot-mysql\_app\_1 ... done

Removing spring-boot-mysql\_mysqldb\_1 ... done

Removing network spring-boot-mysql\_default

If you need to stop and remove all containers, networks, and all images used by any service in docker-compose.yml file, use the command:  
docker compose down --rmi all

# 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.

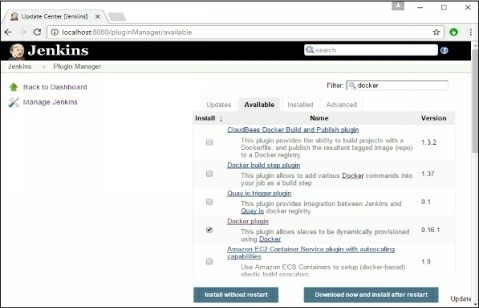
A screenshot of a computer

Description automatically generated

**Step 2** − Go to Manage Plugins.

A screenshot of a computer

Description automatically generated

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

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

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated**Step 5** − In the job, when you go to the Build step, you can now see the option to start and stop containers.

**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.

A screenshot of a computer

Description automatically generated

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.

A computer screen with a computer code

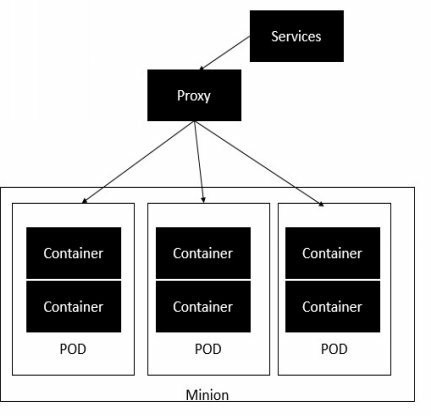
Description automatically generated with medium confidence

# 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 &mius;

* + 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.

# Docker - Working of Kubernetes

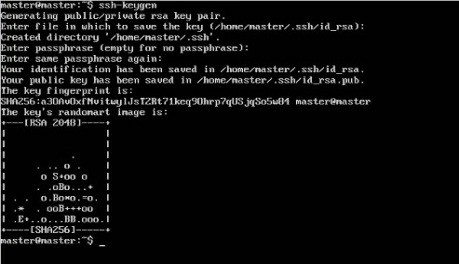
In this chapter, we will see how to install Kubenetes via kubeadm. This is a tool which helps in the installation of Kubernetes. Let’s go step by step and learn how to install Kubernetes.

**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 sshlogin. You can do this using the following command.

ssh-keygen

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



**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 kubernetessite 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](http://αpt.kubernetes.io/kubernetes-xeniαlmαin)” | sudo tee /etc/apt/sources.list.d/docker.list

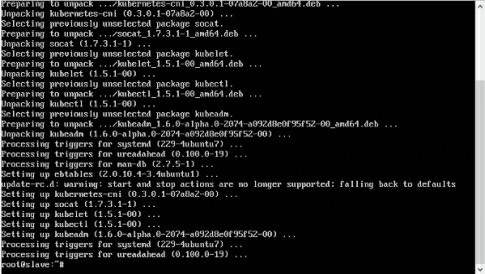
A screen shot of a computer

Description automatically generated**Step 4** − We then issue an apt-get update to ensure all packages are downloaded on the Ubuntu server.

**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



**Step 7** − Once all kubernetes packages are downloaded, it’s time to start the kubernetes controller using the following command −

kubeadm init

A screenshot of a computer error

Description automatically generated

Once done, you will get a successful message that the master is up and running and nodes can now join the cluster.

## 1. Overview

Docker containers are used to run applications in an isolated environment. By default, all the changes inside the container are lost when the container stops. If we want to keep data between runs, Docker volumes and bind mounts can help.

In this tutorial, we’ll learn about Docker Volumes, and how to manage and connect them to containers.

## 2. What Is a Volume?

### 2.1. The Docker File System

A docker container runs the software stack defined in an [image](https://www.baeldung.com/docker-images-vs-containers). Images are made of a set of read-only layers that work on a file system called the Union File System. When we start a new container, Docker adds a read-write layer on the top of the image layers allowing the container to run as though on a standard Linux [file system](https://www.baeldung.com/ops/docker-container-filesystem).

So, any file change inside the container creates a working copy in the read-write layer. However, **when the container is stopped or deleted, that read-write layer is lost**.

We can try this out by running a command that writes and then reads a file:

$ docker run bash:latest \

bash -c "echo hello > file.txt && cat file.txt"

The result is:

hello

But if we run the same image with just the command to output the file’s contents:

$ docker run bash:latest bash -c "cat file.txt"

cat: can't open 'file.txt': No such file or directory

The second run of the container runs on a clean file system, so the file is not found.

### 2.2. Bind Mounts

A Docker [bind mount](https://docs.docker.com/storage/bind-mounts/) is a high-performance connection from the container to a directory on the host machine. **It allows the host to share its own file system with the container**, which can be made read-only or read-write.

This allows us to use a container to run tools that we don’t want to install on our host, and yet still work with our host’s files. For example, if we wanted to use a custom version of bash for a particular script, we might execute that script in a bash container, mounted to our current working directory:

$ docker run -v $(pwd):/var/opt/project bash:latest \

bash -c "echo Hello > /var/opt/project/file.txt"

The –v option can be used for all forms of mounting and specifies, in this instance, the source on the host – the working directory in the output of $(pwd)  – and the target mount point in the container – /var/opt/project.

After running this command, we will find file.txt in the working directory of our host machine. This is a simple way to provide persistent files between invocations of a Docker container, though it is most useful for when the container is doing work on behalf of the host.

One good use case for this would be executing various versions of a language’s build tools in Docker to avoid having conflicting installations on a developer machine.

We should note that $(pwd -W) is sometimes needed on Windows bash shells to provide the working directory in a form that the bash shell can pass to Docker.

### 2.3. Docker Volumes

A bind mount uses the host file system, but [**Docker volumes**](https://docs.docker.com/storage/volumes/)**are native to Docker**. The data is kept somewhere on storage attached to the host – often the local filesystem. The volume itself has a lifecycle that’s longer than the container’s, allowing it to persist until no longer needed. Volumes can be shared between containers.

In some cases, the volume is in a form that is not usable by the host directly.

## 3. Managing Volumes

Docker allows us to manage volumes via the [docker volume](https://docs.docker.com/engine/reference/commandline/volume/) set of commands. We can give a volume an explicit name (named volumes), or allow Docker to generate a random one (anonymous volumes).

### 3.1. Creating Volumes

We can create a volume by using the create subcommand and passing a name as an argument:

$ docker volume create data\_volume

data\_volume

If a name is not specified, Docker generates a random name:

$ docker volume create

d7fb659f9b2f6c6fd7b2c796a47441fa77c8580a080e50fb0b1582c8f602ae2f

### 3.2. Listing Volumes

The ls subcommand shows all the volumes known to Docker:

$ docker volume ls

DRIVER VOLUME NAME

local data\_volume

local d7fb659f9b2f6c6fd7b2c796a47441fa77c8580a080e50fb0b1582c8f602ae2f

We can filter using the -f or –filter flag and passing key=value parameters for more precision:

$ docker volume ls -f name=data

DRIVER VOLUME NAME

local data\_volume

### 3.3. Inspecting Volumes

To display detailed information on one or more volumes, we use the inspect subcommand:

$ docker volume inspect ca808e6fd82590dd0858f8f2486d3fa5bdf7523ac61d525319742e892ef56f59

[

{

"CreatedAt": "2020-11-13T17:04:17Z",

"Driver": "local",

"Labels": null,

"Mountpoint": "/var/lib/docker/volumes/ca808e6fd82590dd0858f8f2486d3fa5bdf7523ac61d525319742e892ef56f59/\_data",

"Name": "ca808e6fd82590dd0858f8f2486d3fa5bdf7523ac61d525319742e892ef56f59",

"Options": null,

"Scope": "local"

}

]

We should note that the Driver of the volume describes how the Docker host locates the volume. Volumes can be on remote storage via nfs, for example. In this example, the volume is in local storage.

**AD**

### 3.4. Removing Volumes

To remove one or more volumes individually, we can use the rm subcommand:

$ docker volume rm data\_volume

data\_volume

### 3.5. Pruning Volumes

We can remove all the unused volumes with the prune subcommand:

$ docker volume prune

WARNING! This will remove all local volumes not used by at least one container.

Are you sure you want to continue? [y/N] y

Deleted Volumes:

data\_volume

## 4. Starting a Container with a Volume

### 4.1. Using -v

As we saw with the earlier example, we can start a container with a bind mount using the -v option:

$ docker run -v $(pwd):/var/opt/project bash:latest \

bash -c "ls /var/opt/project"

This syntax also supports mounting a volume:

$ docker run -v data-volume:/var/opt/project bash:latest \

bash -c "ls /var/opt/project"

As our volume is empty, this lists nothing from the mount point. However, if we were to write to the volume during one invocation of the container:

$ docker run -v data-volume:/var/opt/project bash:latest \

bash -c "echo Baeldung > /var/opt/project/Baeldung.txt"

Then our subsequent use of a container with this volume mounted would be able to access the file:

$ docker run -v data-volume:/var/opt/project bash -c "ls /var/opt/project"

Baeldung.txt

The -v option contains three components, separated by colons:

* Source directory or volume name
* Mount point within the container
* (Optional) ro if the mount is to be read-only

### 4.2. Using the –mount Option

We may prefer to use the more self-explanatory –mount option to specify the volume we wish to mount:

**AD**

$ docker run --mount \

'type=volume,src=data-volume,\

dst=/var/opt/project,volume-driver=local,\

readonly' \

bash -c "ls /var/opt/project"

The input to –mount is a string of key-value pairs, separated by commas. Here we’ve set:

* type – as volume to indicate a volume mount
* src – to the name of the volume, though this could have been a source directory if we’d been making a bind mount
* dst – as the destination mount point in the container
* volume-driver – the local driver in this case
* readonly – to make this mount read-only; we could have chosen rw for read/write

We should note that the above command will also create a volume if it does not already exist.

### 4.3. Using –volumes-from to Share Volumes

We should note that attaching a volume to a container creates a long-term connection between the container and the volume. Even when the container has exited, the relationship still exists. This allows us to use a container that has exited as a template for mounting the same set of volumes to a new one.

Let’s say we ran our echo script in a container with the data-volume mount. Later on, we could [list all containers](https://www.baeldung.com/ops/docker-list-containers) we have used:

$ docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

4920602f8048 bash "docker-entrypoint.s…" 7 minutes ago Exited (0) 7 minutes ago exciting\_payne

We could run our next container, by copying the volumes used by this one:

$ docker run --volumes-from 4920 \

bash:latest \

bash -c "ls /var/opt/project"

Baeldung.txt

In practice –volumes-from**is usually used to link volumes between running containers**. Jenkins uses it to share data between agents running as Docker containers.