# Docker Essentials

About CloudThat

* CloudThat is the first company in India to Cloud Training & Consulting services for mid-market & enterprise clients around the world. With expertise in major Cloud platforms including Microsoft Azure, Amazon Web Services (AWS) and Google Cloud Platform (GCP). CloudThat is uniquely positioned to be the single technology source for organizations looking to utilize the flexibility and power Cloud Computing provides.
* CloudThat is focused on quickly empowering IT professionals and organizations with leveraging Cloud, Big Data & IoT. Founded by Bhavesh Goswami, an ex-Microsoft and ex-Amazonian who was part of the Microsoft and AWS product development teams.
* Till date we have trained more than 200,000 IT professionals and conducted corporate training for some of the fortune 500 companies which include Accenture, Infosys, Fidelity, HCL, Intuit, GE, TCS, HP, SAP, Oracle, Western Union, Philips, Flipkart, L&T and Samsung, just to name a few.
* We have presence in Bengaluru, USA & UK, but offer on-site and pre-scheduled public batches in different IT centric cities of India and Overseas.
* CloudThat is a Microsoft Gold Partner, Advanced AWS Consulting partner, Google Consulting Partner, Red Hat Certified Training Partner, MongoDB Ready Partner, and part of Pearson Testing Network.
* Our current course offerings are on Azure, Dynamics 365, Microsoft Security Suite, AI & Machine Learning, Cloud Security, Analytics, Red Hat, IoT, DevOps, Chef, Docker, Ansible, Kubernetes, Blockchain, Big Data, etc. We are constantly adding more courses and more consulting offerings.

About the Author

**Haris AK**

**Cloud Solutions Architect - DevOps, CloudThat**

Haris works as Cloud Solutions Architect in CloudThat technologies, being passionate about ever evolving technology he works on Docker, Kubernetes, and DevOps Technologies.

Haris architects’ solutions on Cloud as well on-Premises using wide array of platforms/technologies like AWS, Azure and VMWare. Having core training and consulting experience, he is involved in delivering Git, Jenkins, Ansible, Docker, Kubernetes training to corporates like Infosys, Wipro, TCS, Accenture, NetApp, EY, Optym, Valtech etc.

Copyright

**Copyright @2012-21, CloudThat. All Rights Reserved.**

The contents of this course, its modules, and its related materials, including handouts to participants, are Copyright @2012-21, CloudThat. If you are using this curriculum, you are agreeing to comply with and be bound by the following terms and conditions of use, which together with our privacy policy govern ‘CloudThat’ relationship with the participant in relation to this curriculum.

No part of this curriculum may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording or otherwise, without the prior written permission of CloudThat. Please note that the content in this curriculum is protected under copyright law.

The content of this curriculum cannot use in part or whole for conducting any form of training without written consent of CloudThat. The content of this guide is provided for informational use only and is subject to change without notice and should not be rendered as a commitment by CloudThat. CloudThat assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this curriculum.

Please remember this curriculum contains material, which is owned by or licensed to us, any existing artwork, or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names in sample files are for demonstration purposes only and are not intended to refer to any actual organization. Please remember that use of any information of this curriculum is entirely at the personal expense of the participant for which CloudThat will not be liable. The participant is responsible for ensuring that any products, services, or information available through this curriculum meet their specific requirements.

All trademarks reproduced in this curriculum, which is not the property of, or licensed to the operator, are acknowledged in the curriculum. From time to time, this curriculum may also include links to other websites. These links are provided for the participant’s convenience to provide further information. They do not signify that CloudThat endorses the website(s). CloudThat has no responsibility for the content of the linked website(s).

Unauthorized use of this curriculum will give a right to CloudThat to a claim for damages and/or be a criminal offense.

# Table of Content

[Chapter One: Introduction to Docker 7](#_Toc83985982)

[Introduction to Docker 7](#_Toc83985983)

[How is Docker Different from Virtual Machines 7](#_Toc83985984)

[Key Benefits of Docker 7](#_Toc83985985)

[Control Groups / Namespaces 8](#_Toc83985986)

[Control Groups (cgroups) 9](#_Toc83985987)

[Another Union File System (AuFS) 9](#_Toc83985988)

[Lab 1: Creating an EC2 Instance in AWS and Installing Docker 11](#_Toc83985989)

[Task 1: Launching EC2 instances 11](#_Toc83985990)

[Task 2: Connecting to EC2 Instances using SSH 18](#_Toc83985991)

[Task 3: Installing Docker on Ubuntu 18.04 operating system 29](#_Toc83985992)

[Chapter Two: Working with Docker 32](#_Toc83985993)

[Concept of Containers 32](#_Toc83985994)

[Images 33](#_Toc83985995)

[Running Containers as a User 33](#_Toc83985996)

[Setting Environment Variables 34](#_Toc83985997)

[Lab 2: Basic Docker Commands 35](#_Toc83985998)

[Task 1: Creating your first Docker container 35](#_Toc83985999)

[Task 2: Basic Commands to run the Container in Interactive mode 35](#_Toc83986000)

[Task 3: Port Mapping from Docker Host to container 39](#_Toc83986001)

[Lab 3: Docker Lifecycle 43](#_Toc83986002)

[Using Volumes with Docker Containers 52](#_Toc83986003)

[Use cases 54](#_Toc83986004)

[Lab 4: Working with volume mounts in Docker 56](#_Toc83986005)

[Task 1: Starting Docker Containers Bind Mounts 56](#_Toc83986006)

[Task 2: Create a bind mount with --mount option and verify it 59](#_Toc83986007)

[Lab 5: Volume Mounting with Docker Containers 63](#_Toc83986008)

[Task 1: Creating a new docker volume and inspecting containers 63](#_Toc83986009)

[Task 2: Launching a Nginx container mapped to a specific docker volume and verification 63](#_Toc83986010)

[Task 3: Deleting container and attaching the volume to another container 67](#_Toc83986011)

[Task 4: Create a container with tmpfs mount and verify it 70](#_Toc83986012)

[Chapter Three: Docker Networking 72](#_Toc83986013)

[Introduction 72](#_Toc83986014)

[Container Linking 72](#_Toc83986015)

[Binding Containers to Host Port or a Unix Socket 73](#_Toc83986016)

[Docker Bridge 73](#_Toc83986017)

[Configuring DNS 73](#_Toc83986018)

[Multi Host Networking 74](#_Toc83986019)

[Lab 6: Docker Networking 80](#_Toc83986020)

[Task 1: Create a new docker bridge and check connectivity between containers of same bridge 80](#_Toc83986021)

[Task 2: Create a new docker bridge and check connectivity between containers of different bridges 84](#_Toc83986022)

[Task 3: Using ‘Docker network connect’ command create a successful connection between containers of different bridges 87](#_Toc83986023)

[Task 4: Launch a container to host network 90](#_Toc83986024)

[Task 5: Launch a container to none network 92](#_Toc83986025)

[Chapter Four: Dockerfile 93](#_Toc83986026)

[Dockerfile Introduction 93](#_Toc83986027)

[Automating a Build Using Dockerfile 93](#_Toc83986028)

[RUN Instruction 93](#_Toc83986029)

[FROM Instruction 94](#_Toc83986030)

[Docker Cmd and Entrypoint 94](#_Toc83986031)

[Docker Environment Variables 95](#_Toc83986032)

[Lab 7: Building a Dockerfile to setup an Ubuntu container with WordPress application 96](#_Toc83986033)

[Task 1: Deploying MySQL and WordPress containers 96](#_Toc83986034)

[Task 2: Deploying WordPress 100](#_Toc83986035)

[Chapter Five: Docker Hub 106](#_Toc83986036)

[Docker Hub 106](#_Toc83986037)

[Committing Changes 106](#_Toc83986038)

[Private Registry 106](#_Toc83986039)

[Automated Builds 107](#_Toc83986040)

[Advantages of Automated Builds 107](#_Toc83986041)

[Lab 8: Pushing the image to DockerHub 108](#_Toc83986042)

[Chapter Six: Docker Compose 116](#_Toc83986043)

[Docker Compose Introduction 116](#_Toc83986044)

[Working of Docker Compose 116](#_Toc83986045)

[Features of Docker Compose 116](#_Toc83986046)

[Lab 9: Docker Compose 118](#_Toc83986047)

[Task 1: Docker Compose Installation 118](#_Toc83986048)

[Task 2: Create a docker-compose.yaml for Wordpress application 118](#_Toc83986049)

[Chapter Seven: Docker Machine 128](#_Toc83986050)

[Docker Machine Introduction 128](#_Toc83986051)

[How Docker Machine Works 128](#_Toc83986052)

[Creating Secure Connection with the Container using Docker Machine 129](#_Toc83986053)

[Docker Machine on Public Cloud 130](#_Toc83986054)

[Chapter Eight: Docker Swarm 131](#_Toc83986055)

[Docker Swarm 131](#_Toc83986056)

[Key Features of Docker Swarm 131](#_Toc83986057)

[Explanation of Swarm Cluster Creation 131](#_Toc83986058)

[Lab 10: Docker Swarm 133](#_Toc83986059)

[Task 1: Launch Swarm Leader and Slave with Docker Machine 133](#_Toc83986060)

[Task 2: Configuring Swarm Cluster 136](#_Toc83986061)

[Task 3: Deploying Apache Service on Docker Swarm 140](#_Toc83986062)

[Task 4: Verifying the working of apache service 142](#_Toc83986063)

[Task 5: Removing the services and terminating the Swarm Cluster 142](#_Toc83986064)

[Release Notes: 143](#_Toc83986065)

# Chapter One: Introduction to Docker

**Topics covered in this unit:**

* Introduction to Docker
* Why Docker
* Key Benefits of Docker
* Linux Containers (LXC/LXD)
* Control Groups / Namespaces
* Another Union File System (AUFS)

## Introduction to Docker

Docker is used as an open platform for developing, shipping, and running applications. Docker enables you to separate the application from your infrastructure, making software delivery faster. With the help of Docker, we can run an application in a completely isolated environment called Container. There may be more than one container running on a single host. It allows you to package an application with all its dependencies into a standardized unit together and can be moved to another environment. Docker guarantees that packaged image of your application will run the same, regardless of the environment it is running on. It also provides a platform for managing the whole lifecycles of the containers.

In 2011, Solomon Hykes and Sebastien Pahl founded Docker Inc., and then it was introduced to the public in 2013. Ever since, Docker has gained popularity and is being currently used by various popular platforms like Adobe, Netflix and PayPal.

## How is Docker Different from Virtual Machines

Each virtual machine runs its own operating system whereas, in the case of docker, containers share the same kernel of the host. Dockers are relatively lightweight when we compare it with virtual machines, as it is build up with Base OS image which is very small in size and it only has the needed OS features and utilizes the Kernel features from the Host OS. Virtual machines take few minutes to boot up while docker containers take few milliseconds to boot, the Guest OS of the VMs takes time to boot up as a normal full-fledged OS, whereas Docker containers boot up quick owing to the base image which is very small in size.

## Key Benefits of Docker

* Hardware and cost optimization
* Dockers are lightweight and fast
* It takes very less time to boot
* Agile development with the higher velocity of development
* Optimized architecture design
* Simplified maintenance

## Control Groups / Namespaces

**Namespaces**

Containers use namespaces that provide a private view of the system by isolating the containers. When a container is started, Docker creates a set of namespaces for that container. Some of the namespaces that the container uses are:

* **The PID namespace**: PID stands for process id. Each process has a separate and unique PID, and it is used for isolation of each process.
* **The NET namespace**: It stands for networking namespace, and it is used for the management of network interfaces.
* **The MNT namespace**: MNT stands for mount point and is used for managing and mounting filesystems.
* **The UTS namespace**: UTS stands for Unix Timesharing System, and it is used for isolation of kernel and version identifiers.
* **The IPC namespace**: IPC stands for Inter-Process Communication, and it is used for management of access to IPC resources.

Namespaces are used to isolate the host and the container, which means process running within a container cannot see or affect the processes running in another container or the host system.

## Control Groups (cgroups)

Control groups limit an application to use a specific set of resources and allow Docker engine to share available hardware resources to containers. It optionally enforces limits and constraints on resources as well. For example, with the help of cgroups you can limit the memory available to a specific container. They provide a lot of other useful metrics, and they also help in ensuring that each container gets its fair share of memory, CPU, disk I/O and more importantly, that a single container cannot bring down the system by exhausting one of those resources.

Example: Apache web server or MySQL server allowed to take only the required amount of CPU/throughput for every process.

## Another Union File System (AuFS)

AuFS stands for Another Union File System. AuFS takes an existing filesystem and transparently overlays it on a newer filesystem. It allows directories and files of the different filesystem to co-exist under a single roof. AuFS can merge several directories and provides a single merged view of it. This unification process is referred to as a union mount.

How container reads and writes work with AuFS:

**Reading files:**

Consider the following three situations when a container opens a file for read access with overlay:

* File does not exist in the container layer: when a specific file is opened for read access, and it does not exist in the container layer then the storage driver searches for the files in the image layers, starting with the layer just below the container layer unless it finds the specified file, and reads from the layer where it is found.
* The file exists in the container layer: when the file which is to be read is present in the container layer, then it is opened and read from there.
* Files exist in both the container layer and the image layer: when the file opened for read access and the file exists in the container layer as well as one or more image layers, the file is read from container layer. The files in the container layer obscure files with the same name in the image layers.

When Docker first starts a container, the initial read-write layer is empty. As changes occur, they are applied to the empty layer; for example, if we want to change a file, then that file will be copied from the read-only layer below into the read-write layer. The read-only version of the file will still exist but is now hidden underneath the copy. This pattern is traditionally called "copy on write" and is one of the features that made Docker so powerful. Each read-only image layer never changes. When a container is created, Docker builds from the stack of images and then adds the read-write layer on top.

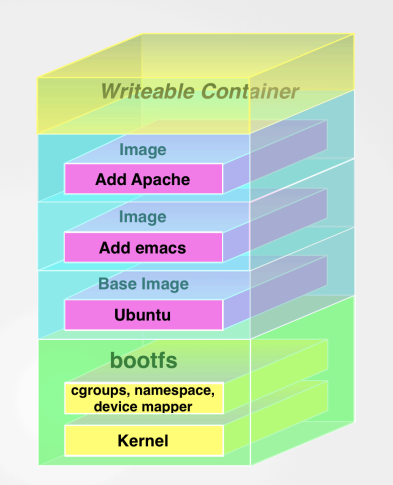


Illustration of a Docker File System

# Lab 1: Creating an EC2 Instance in AWS and Installing Docker

This lab will demonstrate the creation of an EC2 virtual machine with Ubuntu 18.04 operating system image in AWS Cloud Platform and Installing Docker on Ubuntu 18.04 operating system.

## Task 1: Launching EC2 instances

1. Click **Services** dropdown on the AWS console and select **EC2**

Graphical user interface, text, website

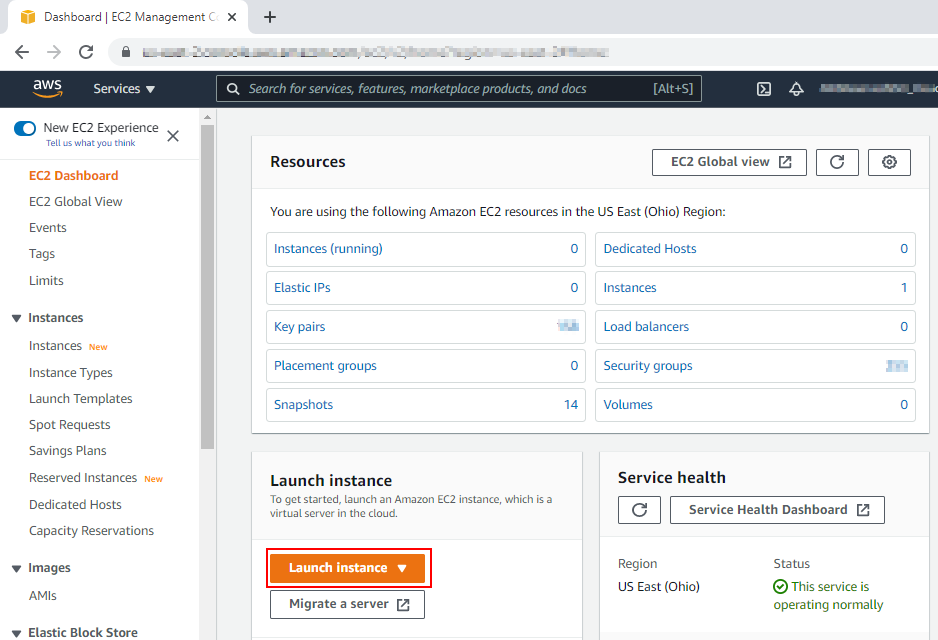
Description automatically generated

1. Select **AWS Region** as **Asia Pacific (Mumbai)**

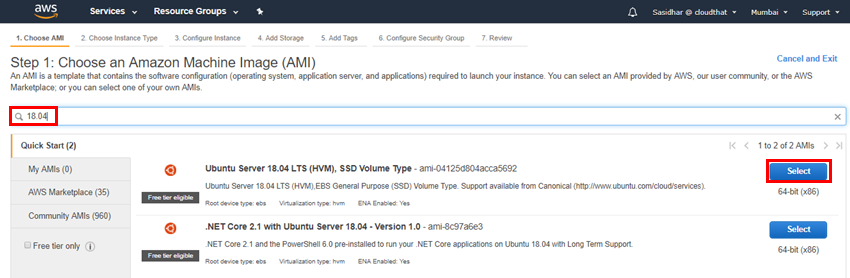
Graphical user interface, text, application

Description automatically generated

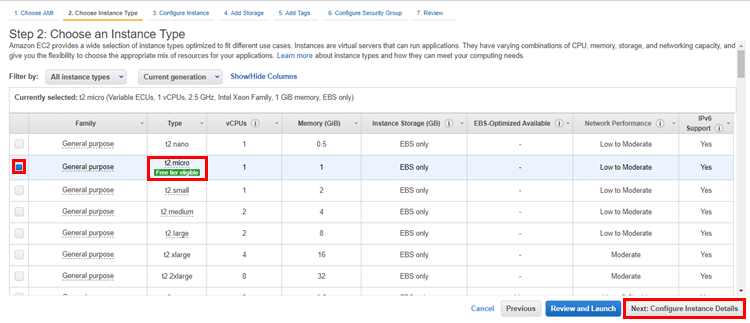
1. Click **Launch Instance** drop down and again click on **Launch Instance**



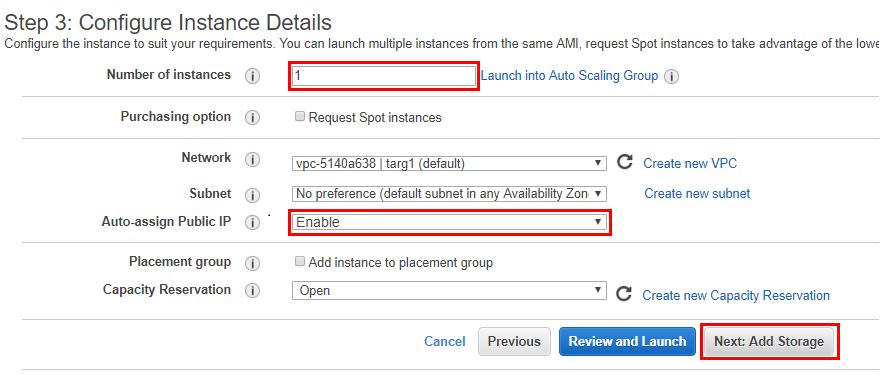
1. In **Choose AMI** section, search for ‘**18.04**’and click **Select** against **Ubuntu Server 18.04 LTS (HVM) Free tier eligible** image



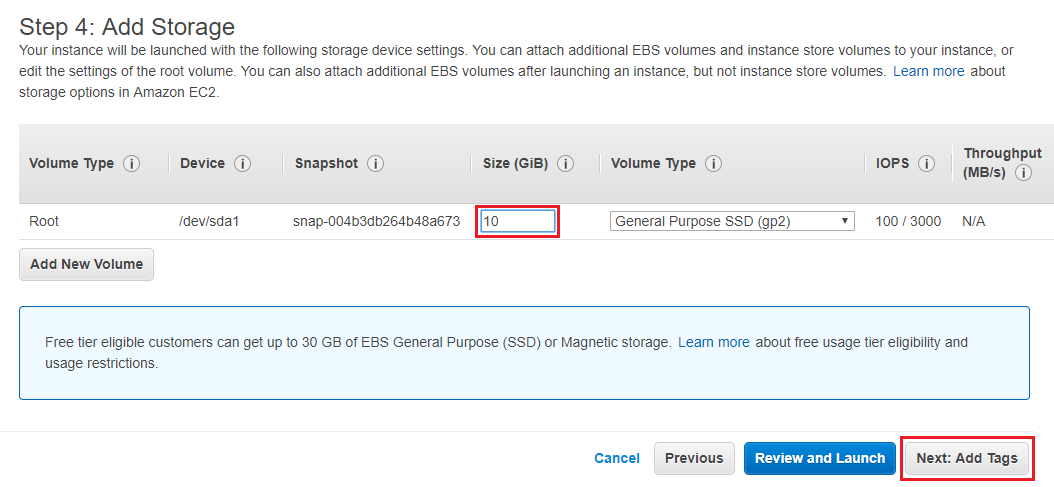
1. In **Choose Instance Type,** check the box against **t2.micro.** Then click **Next: Configure Instance Details**



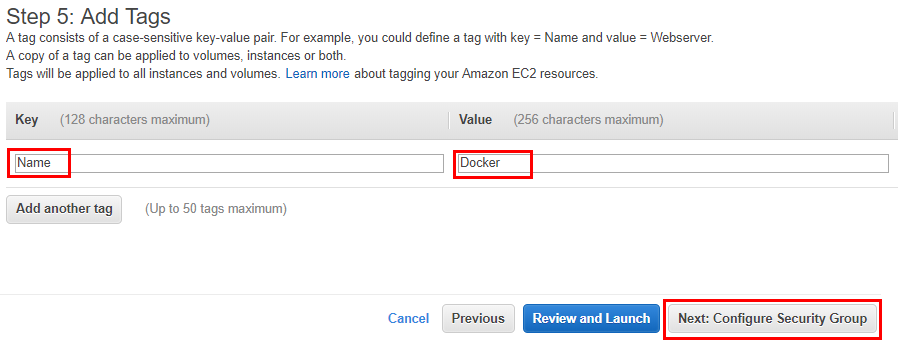
1. In **Configure Instance,** make sure **Number of instances** is set to **1**. Set **Auto-assign Public IP** to **Enable** and Click **Next: Add Storage**



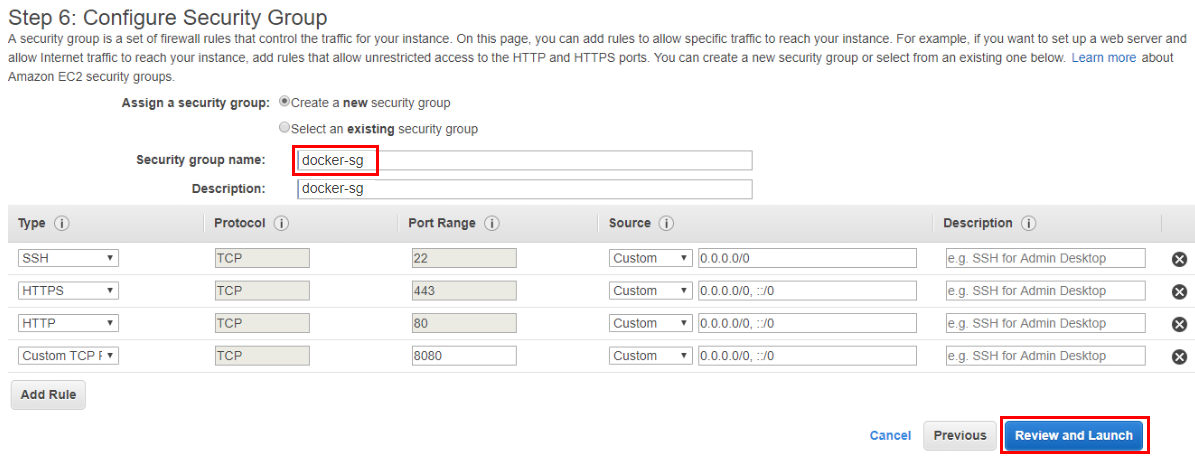
1. In **Add Storage,** ensure size is set to **10 GiB.** Then click on **Next: Add Tags**



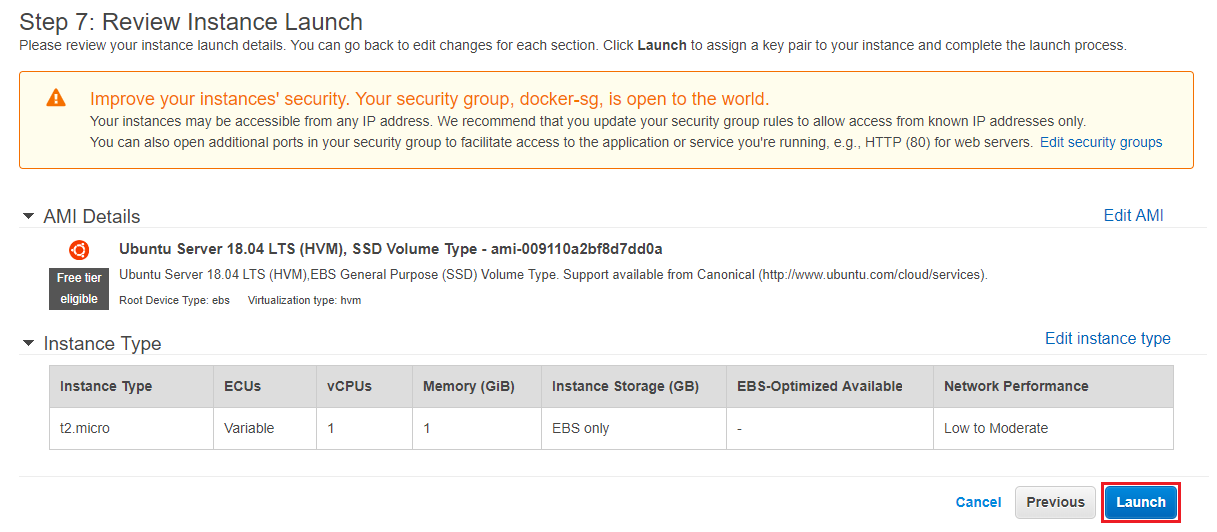
1. In **Add Tags,** set **Key** as **Name** and **Value** as **Docker**. Then click **Next: Configure Security Group**



1. In **Configure Security Group,** select **Create a new security group**
2. Enter **Security group name** as **docker-sg**
3. Enter **Description** as **docker-sg**
4. Ensure **SSH** is added, and it is open to the internet by selecting source **Anywhere** (**0.0.0.0/0**)
5. Add **HTTP** make open to the internet, by selecting source **Anywhere** (**0.0.0.0/0**)
6. Add **HTTPS** make open to the internet, by selecting source **Anywhere** (**0.0.0.0/0**)
7. Add **Custom TCP Ports** **8080** and open them to internet by selecting source **Anywhere** (**0.0.0.0/0**)
8. Then click **Review and Launch**



1. Review the configuration of the instance and click **Launch**

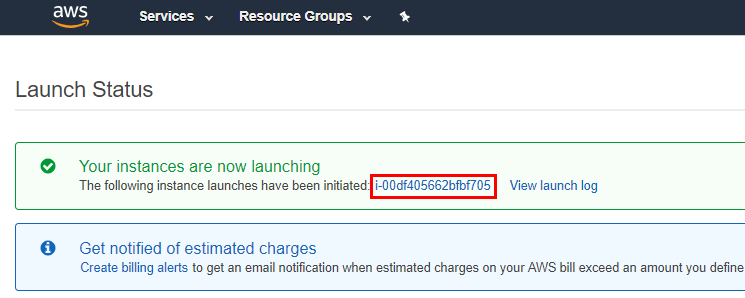


1. Then, select **Create a new key pair** from the drop down. Name it ‘docker-kp’ and then click on **Download Key Pair** button. Once the key file is downloaded, click on **Launch Instances**

A screenshot of a social media post

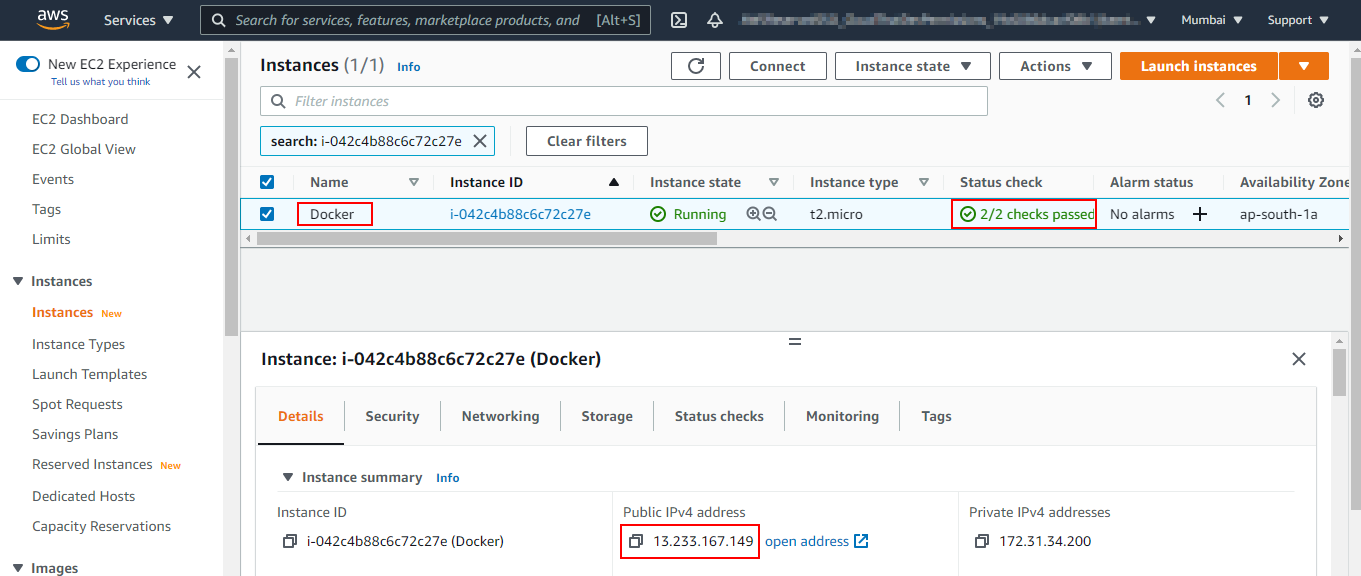
Description automatically generated

1. Click on the instance id to view its status. Then wait for the instance to come up

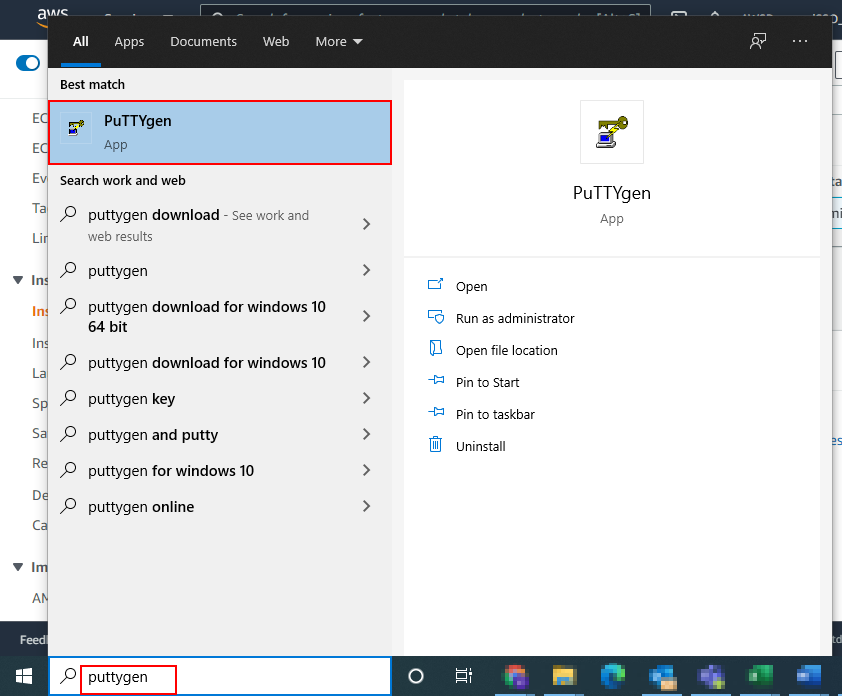


## Task 2: Connecting to EC2 Instances using SSH

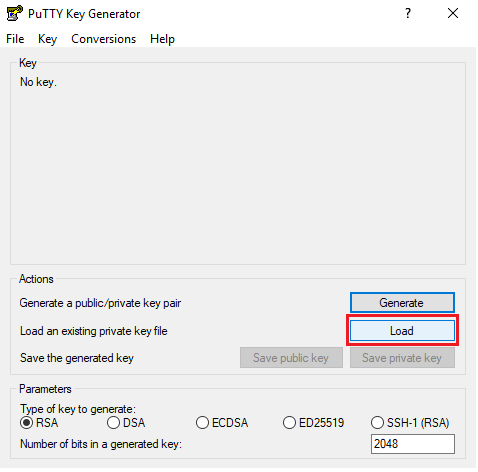
1. In the EC2 console delete the filter you will see one instance is being created and make sure **2/2 checks** have passed and copy the **IPv4** **Public IP**



1. Search for **PuTTYgen** software on your laptop. If not installed already, download and install the latest **putty-X.XX-installer.msi** package from **https://www.putty.org/**. Open puTTYgen



1. On the PuTTY Key Generator window click on the **Load** button

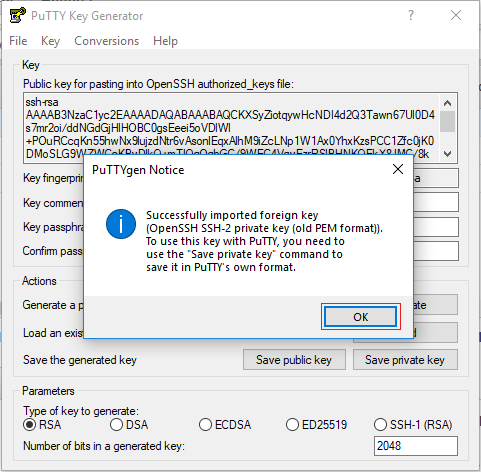


1. In Load Private Key window, make file type as **All files**, search for **docker-kp** key pair, select and click **Open**

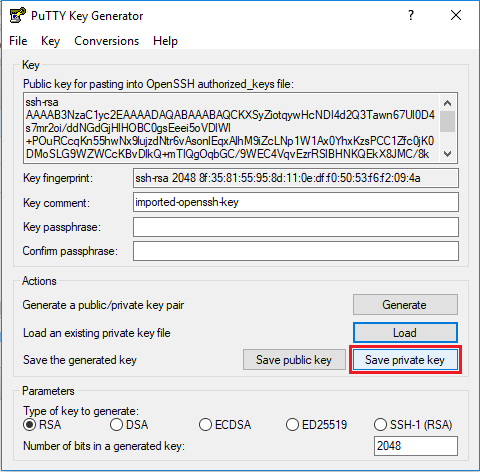
A screenshot of a social media post

Description automatically generated

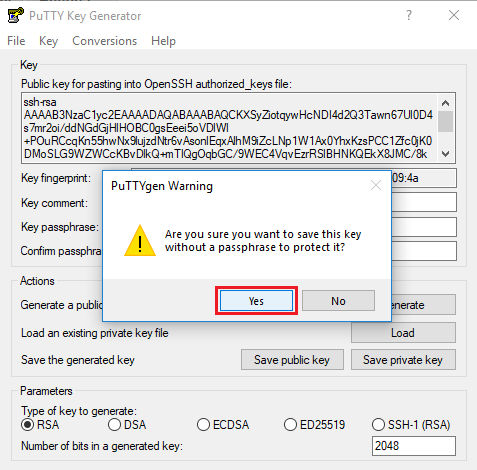
1. Click **OK** on the PuTTYgen Notice window



1. Click on **Save private key** to create **docker-kp.ppk** file



1. In the PuTTYgen Warning window, click **Yes** to save the key without a password

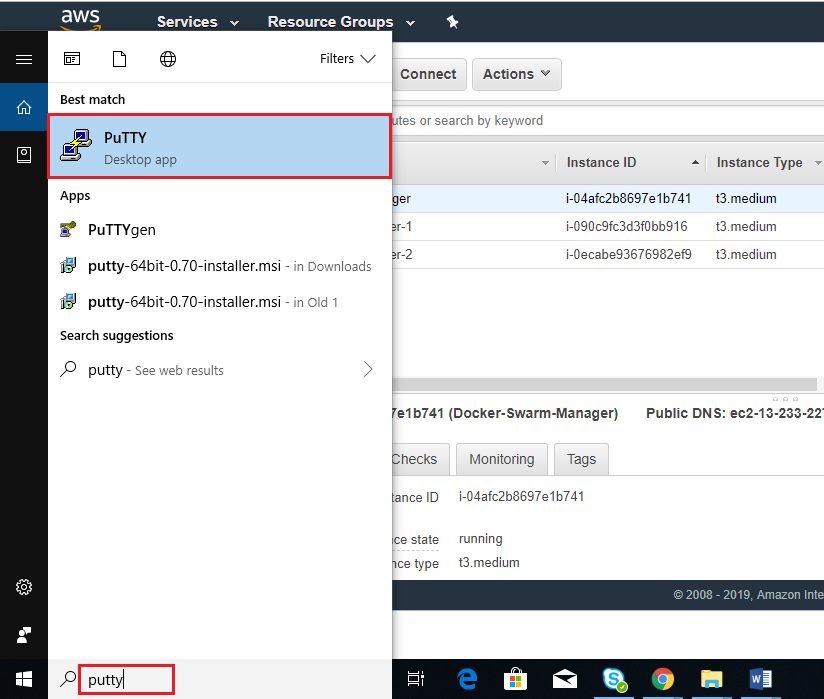


1. Name the file as **docker-kp** and **Save** the file in the desired location. **Close** the **PuTTYgen** window after successfully saving a **docker-kp** file

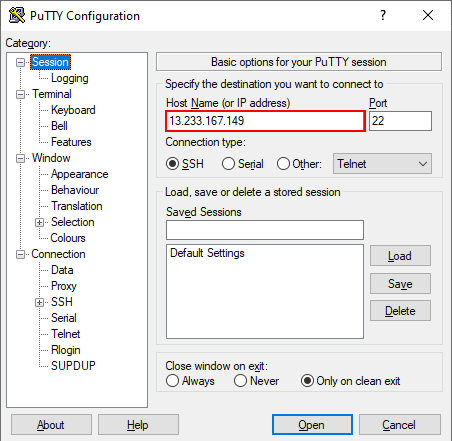
A screenshot of a cell phone

Description automatically generated

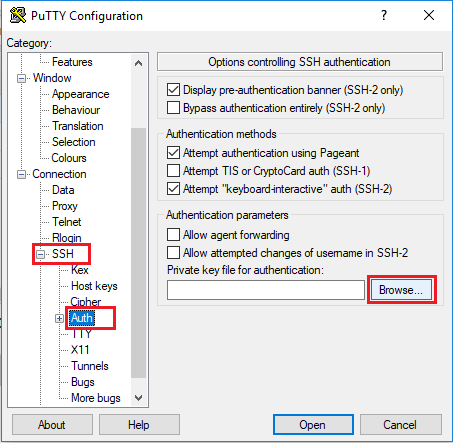
1. Now Search for **PuTTY** software in your laptop and Open **PuTTY**



1. In **PuTTY Configuration** window, Paste your EC2 **Public IP or Public DNS** in the **Hostname (or IP Address)** field



1. Then expand the **SSH** section and select **Auth**, click on the **Browse** button to select your **docker-kp.ppk** file

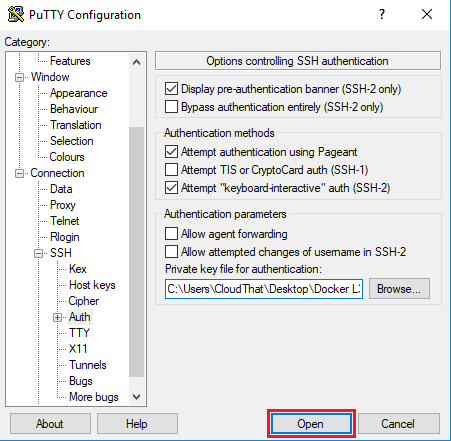


1. In the Select private key window, select your **docker-kp.ppk** file and click **Open**

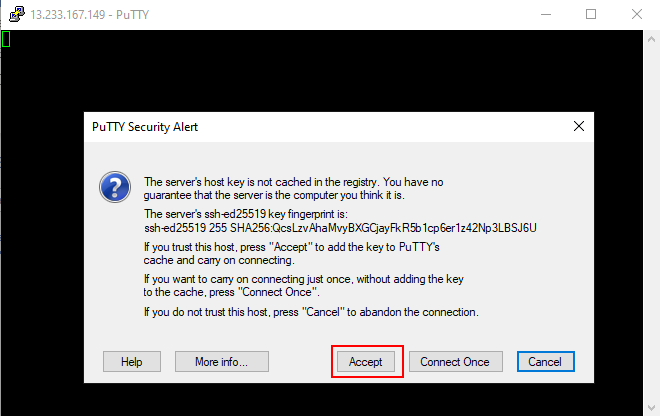
A screenshot of a social media post

Description automatically generated

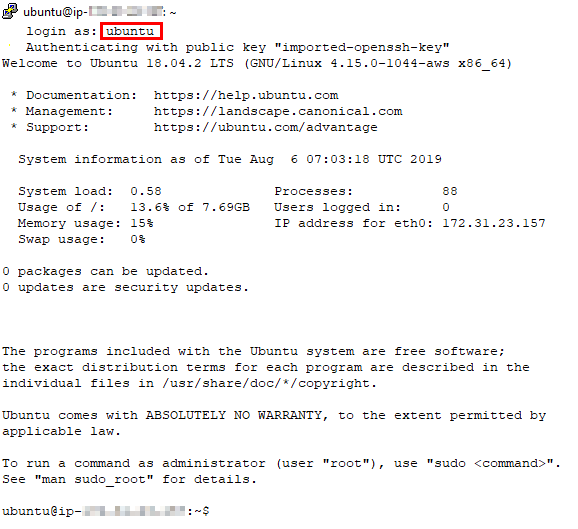
1. Now click **Open** on the PuTTY Configuration window



1. Click **Yes** on the PuTTY Security Alert window



1. Login as the **ubuntu** user



## Task 3: Installing Docker on Ubuntu 18.04 operating system

1. Run the below command to **change** the **hostname** to **docker**

$ sudo hostnamectl set-hostname docker

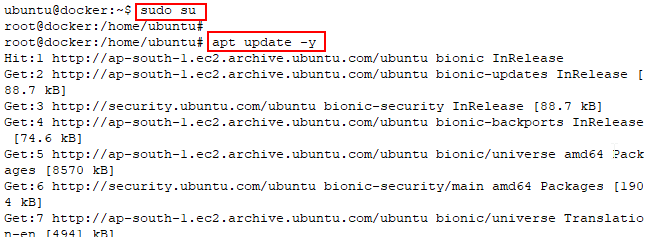


**Note:** Restart or Duplicate the session, you can see the hostname changed to docker

1. Once Logged in, install Docker using the following steps, switch to **root** user and update **apt** repository

$ sudo su

# apt update -y



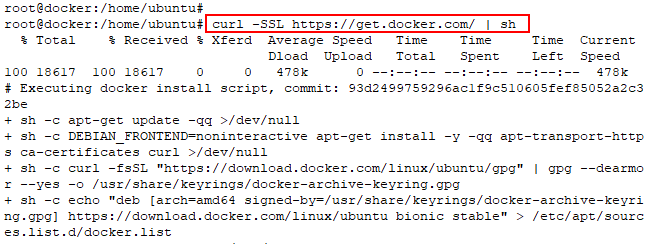
1. Then install the latest version of **curl** using **apt install**

# apt install curl -y



1. Install the latest **Community Edition** of **Docker** using the following command

# curl -SSL https://get.docker.com/ | sh



1. **Docker service must have been started,** and you can verify the service status using the below command and enter ctrl+c

# service docker status



1. The Docker daemon binds to a UNIX socket, which is owned by root user and other users need sudo privilege to access it. To avoid using sudo every time before **docker commands**, add user **ubuntu** to the **docker group** by using the following command

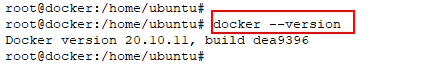
# usermod -aG docker ubuntu



**Note**: To make this work, log out of the EC2 instance and login again.

1. Check the docker version using **docker command**

# docker --version



# Chapter Two: Working with Docker

**Topics covered in this unit:**

* Concept of Containers
* Images
* Volume Mapping
* Running Containers as a User
* Setting Environment Variables

## Concept of Containers

Dockerreplaces Hypervisors with the help of the Container. The container in Docker is usually virtualized at the Operating system level, whereas Hypervisors are virtualized at the hardware level.Containersabstract the application from the operating system and packs everything needed to run in an application (i.e., all its parts and dependencies and the stack which runs on it) and all are combined and packaged into a box.Containerscan boot in 500ms and will be application ready in the time specified and create the latest designs for rapid scaling.

Containers do not rely on the underlying host for running their applications and provide guaranteed execution irrespective of the underlying environment. Containers are made up of portable images, like LXC/VZ images, and they are much more powerful.

**Commands on Containers**

**docker start:** This command takes one or more containers name or container Id and starts the containers.

**docker stop:** This command takes one or more container name or container Id and stops the running containers.

**docker ps:** This command is used to list all the running containers.

**docker ps –a:** We can view all the running and stopped containers using this command.

**docker ps –q:** We can view the running containers IMAGEID.

**docker rm:** We can provide one or multiple Id's of containers to delete with this command.

**docker logs:** This command is used to showcase the standard output of a container.

**docker inspect:** This command is used to list all the details of a specific container.

We can start, stop, move, and delete our containers.

## Images

An Image is an unchangeable and static file that’s originally a snapshot of a container. Images are created using the build command. Images are the basics for the Docker Containers.

Dockerfile are used to build a Docker image, which sits on top of the operating system. Images are completely wrapped applications. From the Images, we will be able to run as many containers as we wish efficiently.

Docker uses the images to run our applications, and it is not the Dockerfile. Some contents which are included in the docker files are:

* **IMAGEID**: It is the first 12 characters, which specify the true identifier of the image. We can create as many tags for the containers, but the IMAGEID will be the same.
* **VIRTUAL SIZE**: Here the size virtual because it’s adding up all the distinct underlying layers.
* **REPOSITORY**: Value in this column comes from the “-t” flag of the Docker build command used to create image or the Docker tag of the existing image.
* The TAG column is just the tag name of the image. Images of docker are stored as series of read-only layers.
* **LATEST tag**: It’s the default tag when you don’t specify any tag to the image.

Whenever a container is started, Docker takes the read-only image and adds a read-write layer on top of it. When an existing file is modified by the running container that file is copied out of the underlying read-only layer and on the top-most read-write layer where the changes are applied. When we stop the container, the existing copy will also be deleted, and Docker calls these combinations of read-only layers with a read-write layer on top a UNION FILE SYSTEM and starting it will launch a fresh container.

**Docker command for Images**

* **docker build:** The Dockerfile is used to build images using Docker build command.
* **docker commit:** This command requires the name or Id of a running container and a name which will be given to the newly created image.
* **docker tag:** This command is used to give a name to an image.
* **docker push:** This command is used to push an image in the repository, the image should have a proper name to push in any specific repository.

## Running Containers as a User

Usually, the docker daemon binds with a UNIX socket instead of the TCP port. The root user owns this UNIX socket, and hence no other users have the permission. To access by others, they should use sudo. Because of this reason, Docker daemon always runs as the root user.

To overcome the problem of sudo when using the Docker, we should create a UNIX group called Ubuntu and add users into it. When the Docker daemon starts, it makes the ownership of the TCP socket read/writable by the Docker group.

The command for creating user and groups:

$ sudo usermod –aG docker <username>

**Note**: For this to get succeed, we must logout and login.

## Setting Environment Variables

There are environment variables for Docker; the following are the environment variables supported by the Docker:

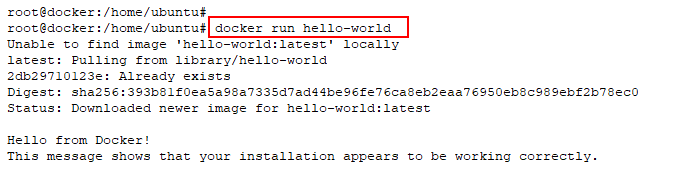
* **DOCKER\_CONFIG**: This specifies the location of client configuration files.
* **DOCKER\_CERT\_PATH**: This specifies the location of our authentication keys.
* **DOCKER\_DRIVER**: This is graph driver to use.
* **DOCKER\_HOST**: This specifies the daemon socket to connect to.
* **DOCKER\_NOWARN\_KERNAL\_VERSION**: This prevents warnings that our Linux kernel is unsuitable for Docker.
* **DOCKER\_RAMDIST**: If this is set, this will disable ‘pivot\_root’.

# Lab 2: Basic Docker Commands

## Task 1: Creating your first Docker container

1. Create your first **docker container** from the image **hello-world** using **docker run command** and verify the **docker** installation is proper

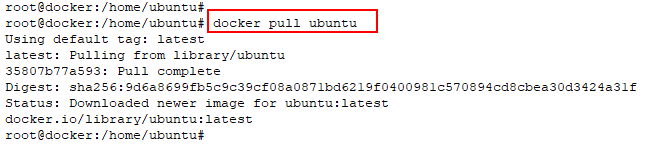
# docker run hello-world



## Task 2: Basic Commands to run the Container in Interactive mode

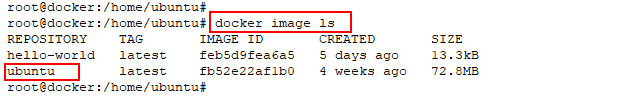
1. Pull the **ubuntu:latest** image from the **DockerHub** public registry using **docker pull** command

# docker pull ubuntu



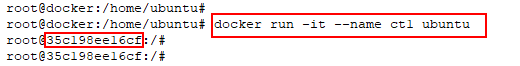
1. Use the below command to list the **docker images** which are pulled to the system

# docker image ls



1. Create a docker container from the existing **ubuntu docker image**. Use the below command to create and start a container with **container name** as **ct1** and switch to the containers shell using **-it (interactive terminal)** option

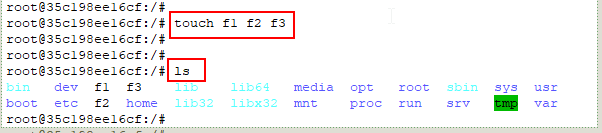
# docker run -it --name ct1 ubuntu



1. Create some files inside the container using **touch** command

# touch f1 f2 f3

# ls

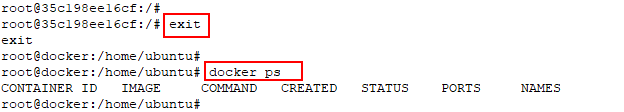


1. Exit from the container by using **exit** command to switch back to **Docker Host** and check for running containers using **docker ps** command, you will not see any running containers. Check **docker ps -a** commandyou will see both containers are in **Exited state**

# exit

# docker ps

# docker ps -a

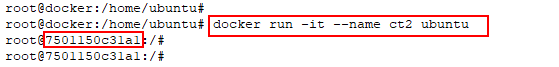


Text

Description automatically generated

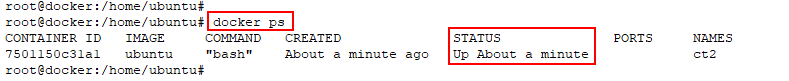
1. Create another docker container named **ct2** using the **ubuntu** image

# docker run -it --name ct2 ubuntu



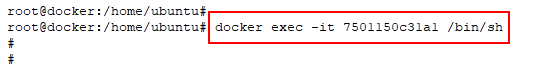
1. Come out of the container’s shell to Docker host shell using **Crtl+P+Q** and check the container status using **docker ps** command

# docker ps



1. Use the **docker exec** **-it** commandto get a **second shell** in the container

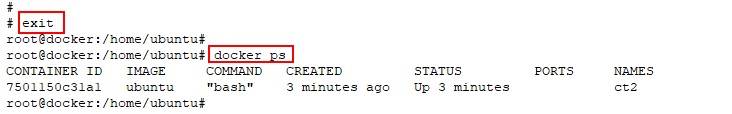
# docker exec -it < container id/name > /bin/sh



1. Exit the container using **exit command** and check the container status using **docker ps**. The container status will be still be **UP** as we have exited from the secondary shell and primary shell is still running inside the container

# exit

# docker ps



1. Use the **docker attach** command to enter the primary shell of the container

# docker attach < container id/name >

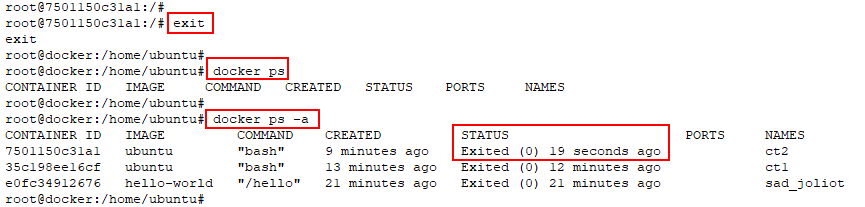


1. Now exit the container to **kill** the **primary process** of the container and check the status of the container using **docker ps command** the container will not be running, check the **docker ps -a command** to verify that the container is in **Exited state**

# exit

# docker ps

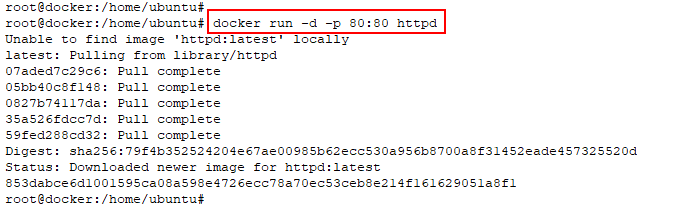
# docker ps -a



## Task 3: Port Mapping from Docker Host to container

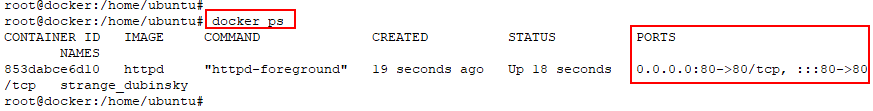
1. Create and start an **apache application** **container** from image **httpd** in **detached mode** using **docker run -d command**, use **-p option** to bind the **host port** to **container port.** In this example we will bind port number 80 of the Docker Host (LHS) to map to port number 80 (RHS) of the container using **-p 80:80**. Notice that we are not going inside container as we are not using interactive mode, the container will be running in the background if use –**d** flag

# docker run -d -p 80:80 httpd

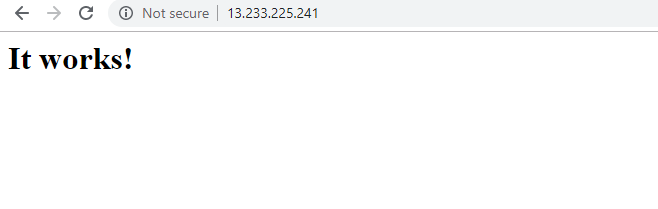
****

1. Verify the container status is **UP** using **docker ps command**, also see the **port mapping rule** in the **ports** section

# docker ps



1. Copy the **EC2 PublicIP** of Docker Host from AWS Console, paste it in browser to verify the working of apache, you will get a “**It works!**” page



1. Use the command **docker exec -it** to get a shell in the container

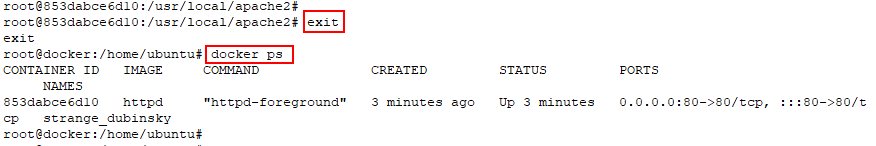
# docker exec -it < container id/name > /bin/bash



1. Exit from the container and check the docker container status. It will be still **UP** as we entered the shell as a **secondary process** and for this container primary process is **apache**

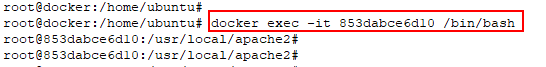
# exit

# docker ps



1. Use the command **docker exec -it** to get back to the shell of the container

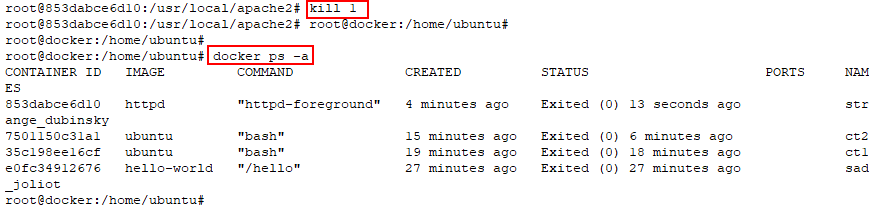
# docker exec -it < container id/name > /bin/bash



1. Apply **kill** **command** on the **primary process** of the container (**PID 1**). The container will be exited as the primary process got exited, verify the status is exited using **docker ps -a** command

# kill 1

# docker ps -a



1. Clean up of resources created in this lab by below command for each container and images.

# docker container stop < container id/name >

# docker container rm < container id/name >

# docker image ls

# docker image rm < image id >

Text

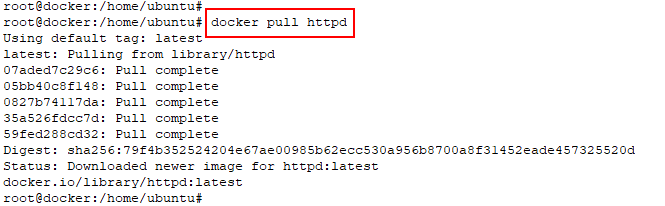
Description automatically generated



# Lab 3: Docker Lifecycle

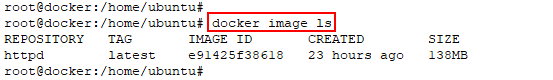
1. Run the following command to pull httpd image from Docker Hub

# docker pull httpd



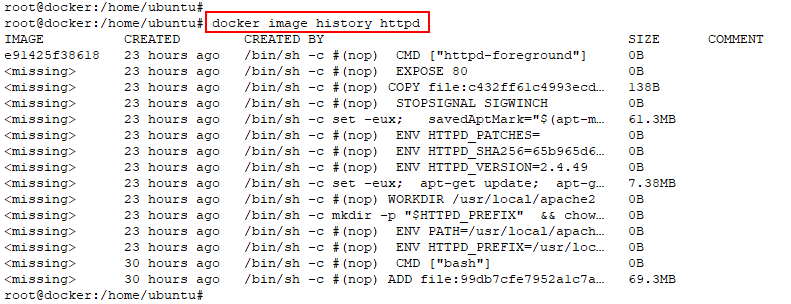
1. Check the pulled httpd image by running below command

# docker image ls



1. Check the httpd image layers using **docker image history**

# docker image history httpd



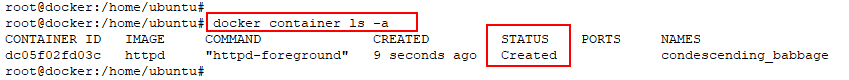
1. Create docker container **read write layer** using below command of the above pulled image

# docker container create httpd



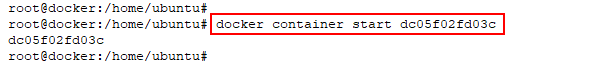
1. Check the status of the container just created, the status should be created

# docker container ls -a



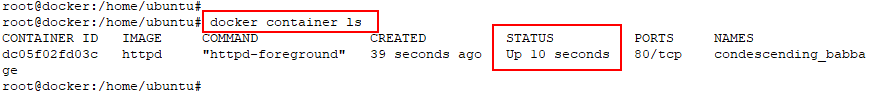
1. Start the container using below command

# docker container start < container id/name >



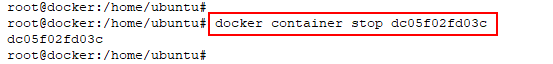
1. Check the status of the started container as **“Up”**

# docker container ls



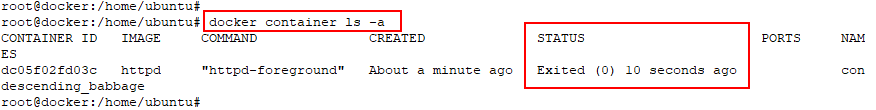
1. Stop the container using below command

# docker container stop < container id/Name >



1. Verify the status of the container is **“Exited”**

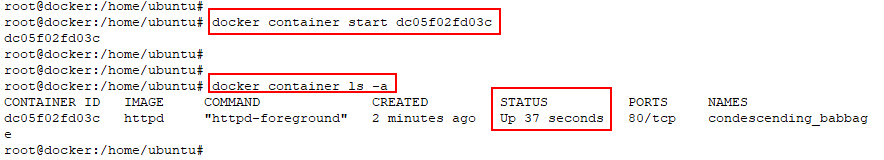
# docker container ls -a



1. Again, start the container by running below command

# docker container start < container id/Name >

# docker container ls -a



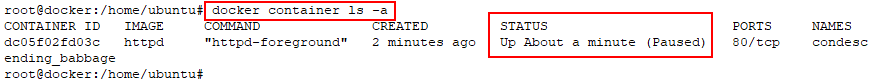
1. **Pause** the running container

# docker container pause < container id/Name >



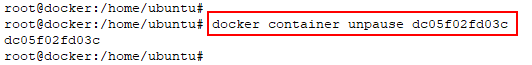
1. Check the status of container as **“Paused”**

# docker container ls -a



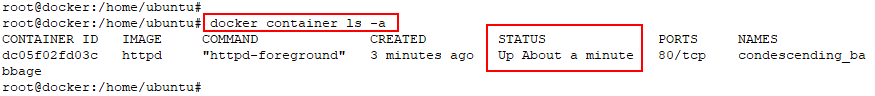
1. Now **unpause** the container

# docker container unpause < container id/Name >



1. Check the status of container after unpause

# docker container ls -a



1. Now enter the container using exec command as shown below

# docker exec -it < container id/name > bash



1. Go to htdocs directory inside the container terminal

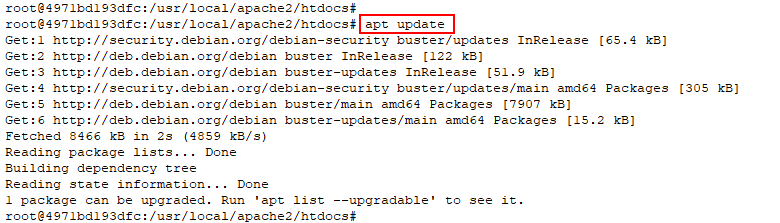
# cd htdocs

Text

Description automatically generated

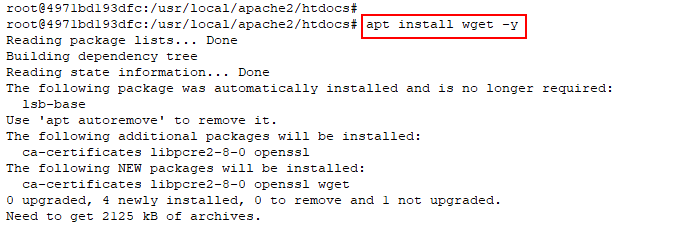
1. Run **apt update** command inside the container.

# apt update



1. Install **wget** package inside the container using **apt install**

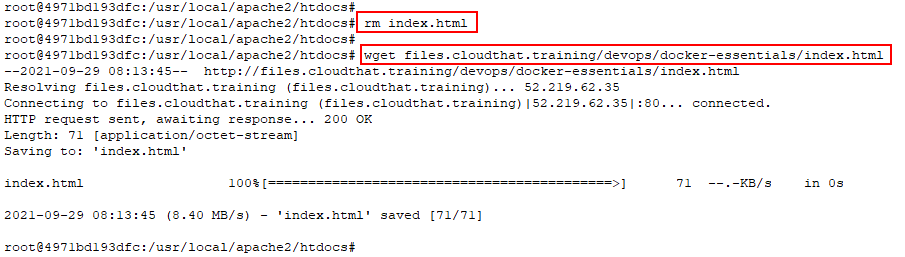
# apt install wget -y



1. Now remove default index.html page and download the custom index.html using below command

# rm index.html

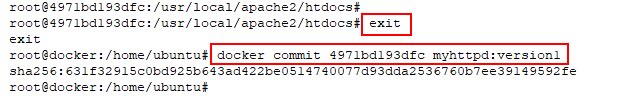
# wget files.cloudthat.training/devops/docker-essentials/index.html



1. Exit from the container and run **docker commit** command to create an image of the modified container, name the new image as **myhttpd:version1**

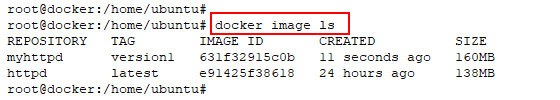
# exit

# docker commit < container id/name > myhttpd:version1



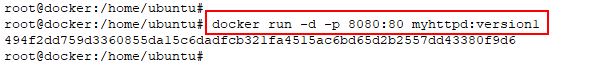
1. After running the commit verify the new image is present in the docker host local image cache.

# docker image ls



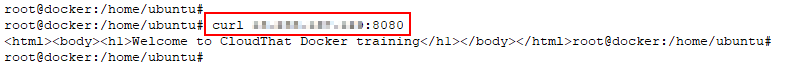
1. Now use your image and create new container in **detached** mode. Use **–publish** option of the docker to see the webpage given by container from Docker Host.

# docker run -d -p 8080:80 myhttpd:version1



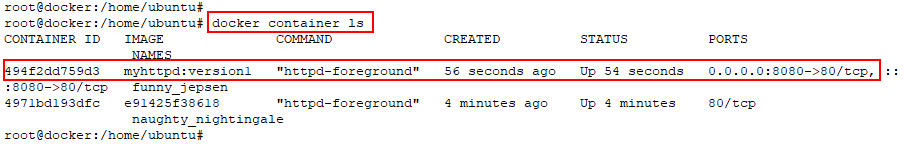
1. Run a curl command with your Public IP of your EC2 instance. You need to see the HTML page

# curl < public IP>:8080



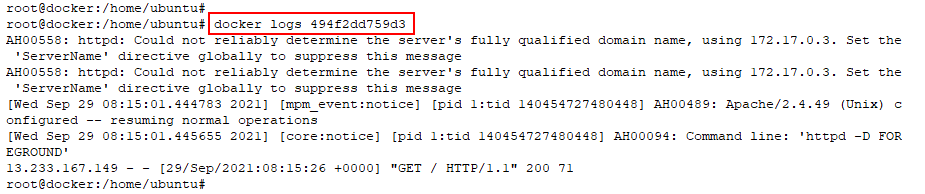
1. List the running docker containers

# docker container ls



1. Check the logs from your container using **docker logs** command

# docker logs < container id/name >



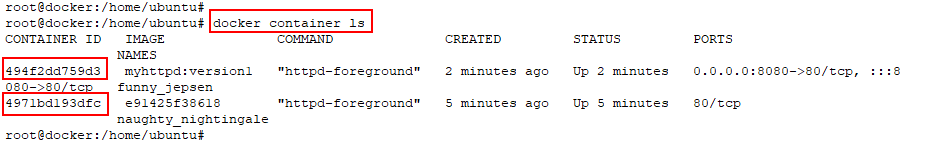
1. Check the metrics consumed by your container using **docker stats** command

# docker stats < container id/name >



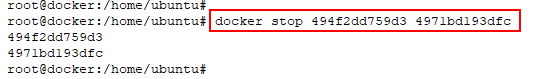
1. List the docker containers and make a note of your container

# docker container ls



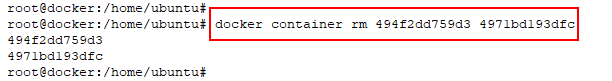
1. Stop the running containers by using below command

# docker stop < container id/name > < container id/name >



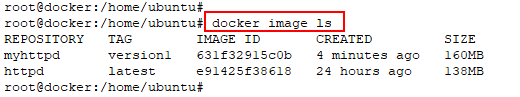
1. Delete the stopped containers completely

# docker container rm < container id/name >



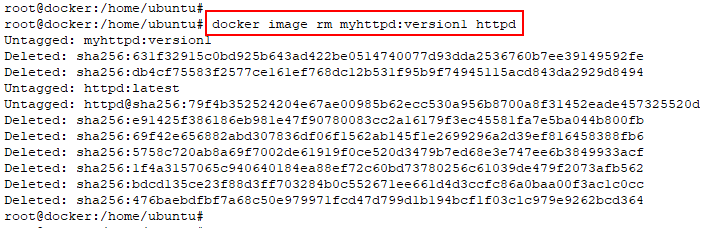
1. List all the images present in Docker Host and make note of image id

# docker image ls



1. Delete the docker images by using below command

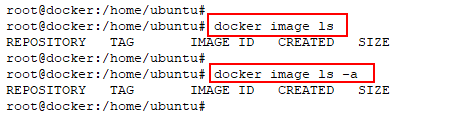
# docker image rm < image id/name > < image id/name >



1. Execute the below command to make sure all the images are deleted

# docker image ls

# docker image ls -a

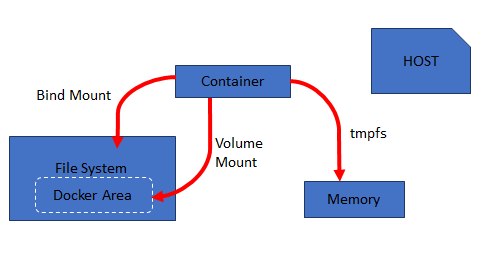


## Using Volumes with Docker Containers

Data Persistency in Docker

We already know how the files inside a container are created on a writable layer of the filesystem inside a container. The problems here are the persistence of data while the container is not running, dependency on the host machine, and the latency while using a storage driver provided a union file system when compared to regular data volumes.

Docker solves this problem by providing filesystem mounts from the container to the host machine. Mounts will ensure data persistency even while the container is not running. There are three types of storage solutions available with Docker; they are **volume mount, bind mount,** and **tmpfs mount**.



**Docker Volumes**

A volume when created is stored within a storage directory of the Docker host, which is mounted to the container as you mount the volume to the container. These volumes are completely managed by Docker and are isolated from other services of the host machine. We can create additional volumes using **docker volume create** command. Generally, the volumes are created by Docker while creating containers. A single volume can be attached to multiple containers, even if no container is using the volume, it will be still available in the Docker storage directory unless we remove them manually by docker volume prune command.

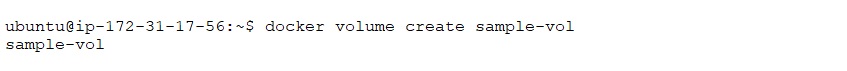
We can name the volume while we mount it, if not Docker will assign a unique anonymous name to the volume. With the help of volume drivers, we can store data on remote hosts or with cloud providers, which will support encryption and other additional functionalities.

The main advantages of volumes are

* Easy backup and migration
* Easy to manage using Docker CLI and Docker API
* Supports both Linux and Windows environments
* A single volume can be shared across multiple containers

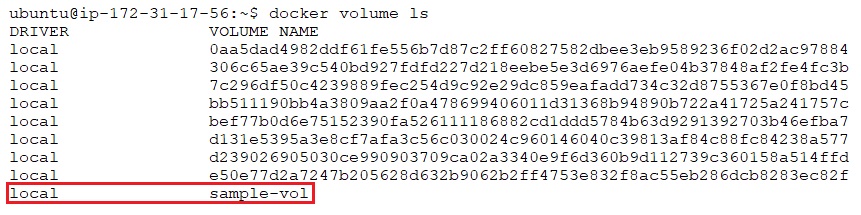
**To create a volume:**

$ docker volume create sample-vol



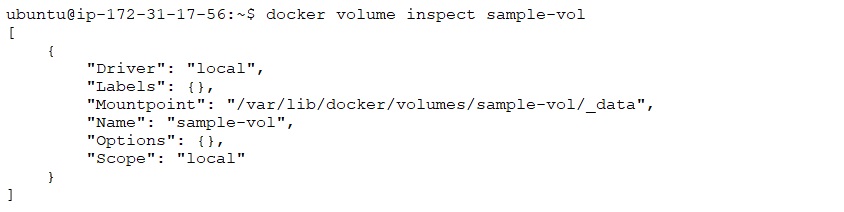
**To list volumes:**

$ docker volume ls



**To inspect a volume:**

$ docker volume inspect sample-vol



**To remove a volume:**

$ docker volume rm sample-vol



**Bind mounts**

Bind mount works in a similar way of volume; the main difference is that any file or directory in the host machine can be mounted into a container. These host directories or files are not managed and isolated by Docker, and hence it depends on the directory structure of the host. There are plenty of advantages if we use volumes instead of bind mounts and functionality of bind mounts when compared to volumes are limited. One of the main disadvantages of bind mount is that processes other than Docker can alter them any time. Also, Docker CLI commands cannot be used to directly manage bind mounts. The advantages include performance and availability of data at the host level. Docker recommends using named volumes instead of bind mounts for new applications.

We have already performed labs on bind mount using -v option. We can also use --mount option to mount volumes. The only difference here is -v creates an endpoint as a directory in the docker host if it is not already present. --mount will not create a directory or file endpoint if it is not already present in the host system. Instead, it will generate an error.

**tmpfs mount**

Unlike volumes and bind mounts, tmpfs mount doesn’t provide data persistency. tmpfs is a temporary mount between the container and the host system’s memory. Although files are created outside containers writable layer, they are not stored in the host filesystem. The mount is removed when the container stops, and the files will not be persisted.

tmpfs is supported only if we are using Docker on Linux platform. It is not possible to share a tmpfs mount between multiple containers. tmpfs mounts are used internally by swarm services to mount secrets into a service’s containers. --tmpfs flag or --mount flag can be used for tmpfs mounts. Docker recommends using --mount flag for both containers and services.

If your application needs to write large volume of non-persistent data and you need performance, tmpfs mounts are the best choice you can make.

## Use cases

**Volumes**

* A single volume can be shared across multiple containers
* Volumes are not removed even while the container is removed
* Completely managed by Docker, no dependency on host directory structure
* Can store data on remote hosts and cloud facilities
* Easy migration, backup, and restore of data

**Bind mounts**

* For creating a single point of sharing for configuration files from host to containers.
* Like /etc/resolv.conf is mounted from host to each container for DNS resolution
* If data needs to be updated from host level like source code or build artifacts
* Like if containers /var/www/html is mounted to host we can modify it any time
* If we know the directory structure of the host, bind mounts are performant

**tmpfs mounts**

* If you don’t want your data to be persisted in host machine or container
* If you have secured data that must be erased on stopping the container
* If your application needs to process huge amount of data with maximum performance

# Lab 4: Working with volume mounts in Docker

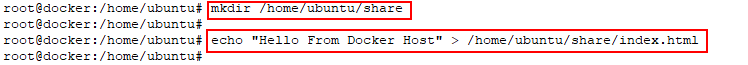
## Task 1: Starting Docker Containers Bind Mounts

Now let us start a Docker container with volume mapping and share the host machine volume with the container.

1. Create a directory **“share”** in home directory of your host machine

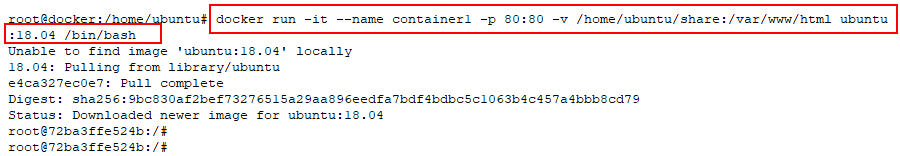
# mkdir /home/ubuntu/share

# echo "Hello From Docker Host" > /home/ubuntu/share/index.html



1. Start a docker container with volume mapped. The directory “/var/www/html” inside the docker container will be mapped to the directory “/home/ubuntu/share” on the host machine
2. Use the following command to achieve the task

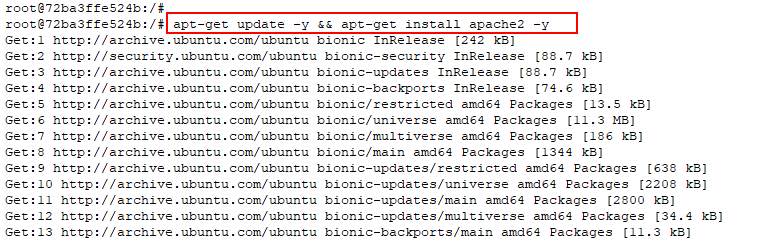
# docker run -it --name container1 -p 80:80 -v /home/ubuntu/share:/var/www/html ubuntu:18.04 /bin/bash



**Note**: The above command will create a Docker container with name container1 and volume mapped.

1. We will install apache2 inside the Docker container, as we did in the previous task, using the following commands

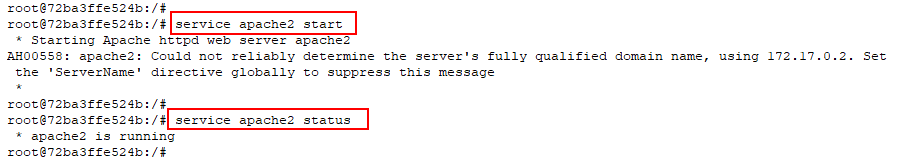
# apt-get update -y && apt-get install apache2 -y



1. Once Apache is installed, start the Apache service inside the Docker container using the following command and check the service status

# service apache2 start

# service apache2 status



1. Paste the Public Ip or public DNS URL of the Instance in a browser and make sure the default page is loading

A screenshot of a cell phone

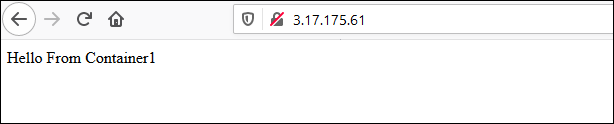
Description automatically generated

1. Use the echo command to input the value ‘Hello From Container1’ to the index file in the container

# echo 'Hello From Container1' > /var/www/html/index.html



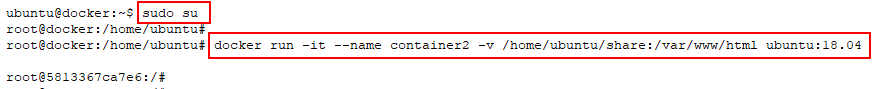
1. Again, paste the public DNS URL of the Instance in a browser and make sure the default page is loading



1. **Now open another terminal and log in to the Instance** . Let us start another container, which will also be mapped to the same folder on the host machine to which container1 is mapped. Use the following command

$ sudo su

# docker run -it --name container2 -v /home/ubuntu/share:/var/www/html ubuntu:18.04



**Note**: This will create a container2, which is also mapped to the same folder as container1. But no port mapping is done to the container2, so you cannot access it from the browser.

1. Input the below content in the index file of the new container using the below command

# echo 'Hello From Container2' > /var/www/html/index.html



1. Check the public IP in your browser

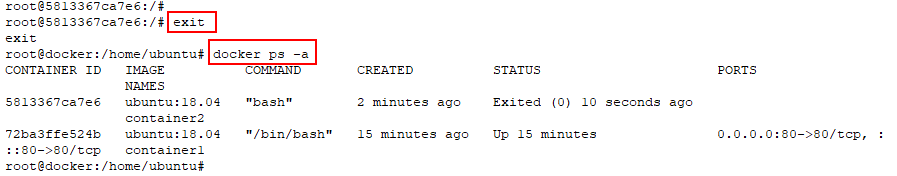
A screenshot of a cell phone

Description automatically generated

1. We just made changes to the /var/www/html/index.html file, which in turn will change the contents of the index.html file in container1. This is because both the containers are mapped to the same volume of the host.
2. Log out of the Ubuntu container. Find the Ubuntu container id by listing the containers and note the id. Use the id in the next command

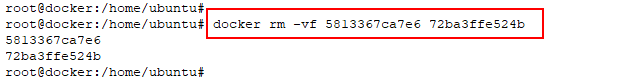
# exit

# docker ps -a



1. Remove the container with the help of command written below

# docker rm -vf < container1 id/name > < container2 id/name >



## Task 2: Create a bind mount with --mount option and verify it

1. Use the below command to bind-mount the **‘’share’’** directory into your container at /app/

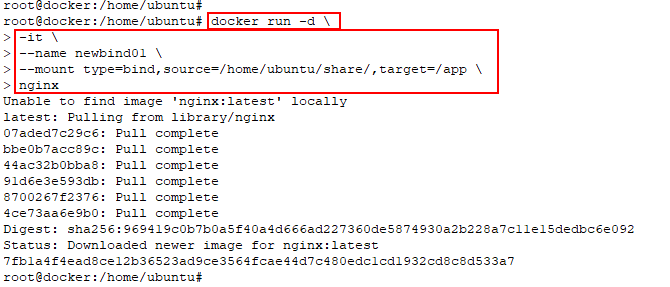
# docker run -d \

> -it \

> --name newbind01 \

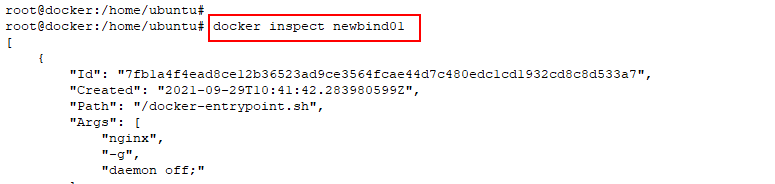
> --mount type=bind,source=/home/ubuntu/share/,target=/app \

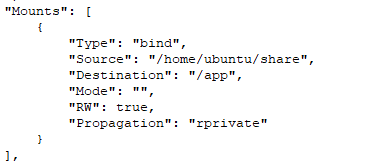
> nginx:latest



1. Use the below command to verify if the bind-mount was created correctly

# docker inspect newbind01

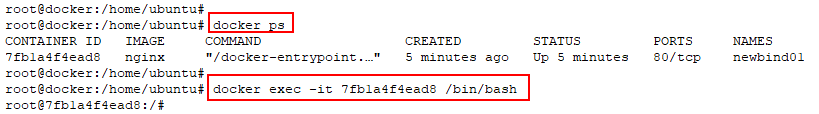




1. Login to the container and verify

# docker ps

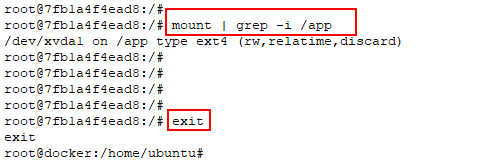
# docker exec -it <container id/name> /bin/bash



1. If you run the below command, you can see a drive mounted in the system as well. Exit from the container

# mount | grep -i /app

# exit



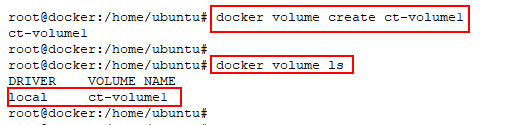
# Lab 5: Volume Mounting with Docker Containers

## Task 1: Creating a new docker volume and inspecting containers

1. Use **docker volume create command** for creating a new **docker volume**

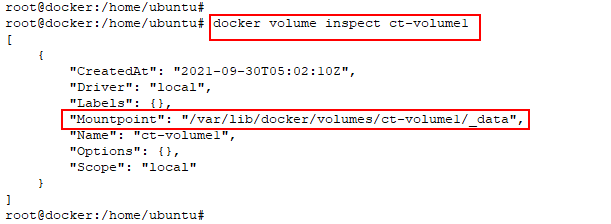
# docker volume create < volume name >

# docker volume ls



1. Inspecting the Docker Volume

# docker volume inspect < volume name >



## Task 2: Launching a Nginx container mapped to a specific docker volume and verification

1. Create **nginx-container** using **nginx:latest** image, provide volume mapping from the **ct-volume1** docker volume **source** to the document root of **nginx-container** **destination**, verify the status of container using **docker ps**

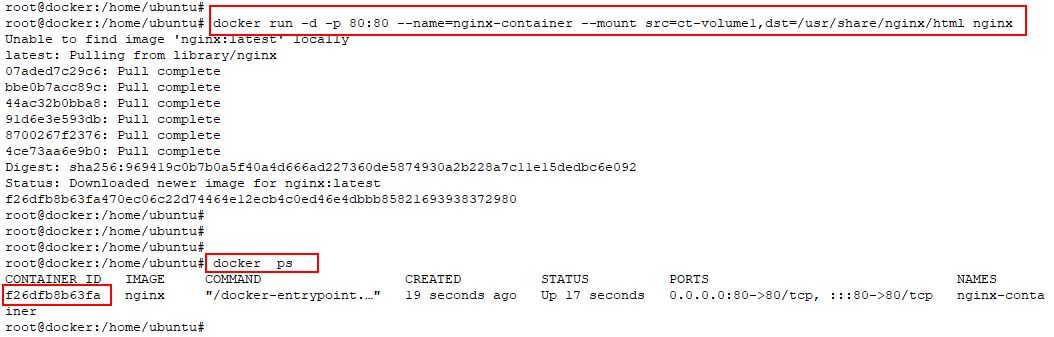
**Note**: We can use shortcuts for source and destination in below command.

Source = src

Destination = dst

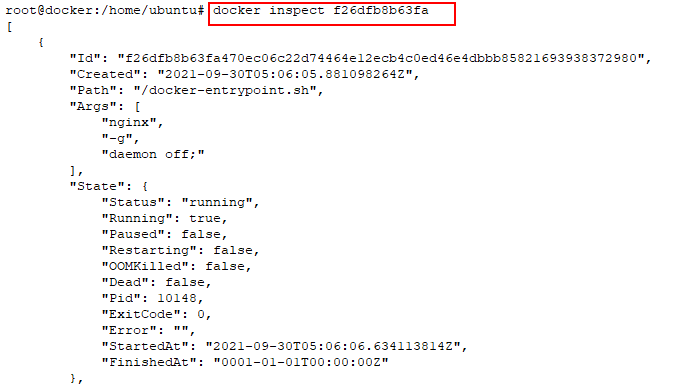
# docker run -d -p 80:80 --name=nginx-container --mount src=**<volume name>**,dst=/usr/share/nginx/html nginx

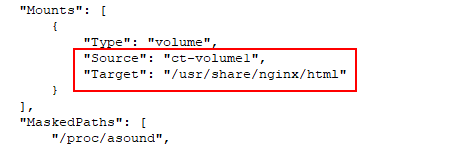
# docker ps



1. Inspect the **nginx-container** and verify the **Mounts** section

# docker inspect < container id/name >





1. List the contents of **/var/lib/docker/ct-volume1/\_data** you should be able to see **index.html** from container in this host location

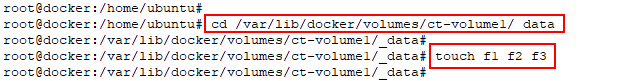
# ls /var/lib/docker/volumes/< volume name >/\_data/



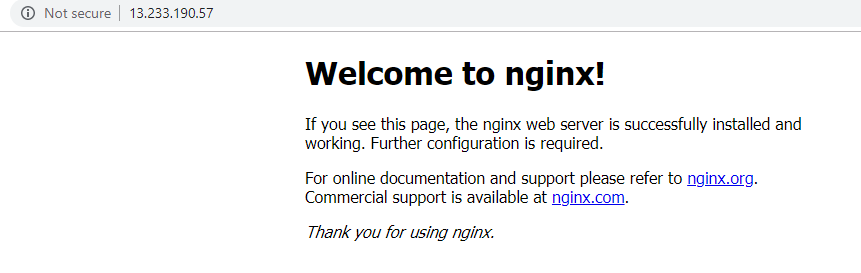
1. Create some files in the host volume location using the **touch command**

# cd /var/lib/docker/volumes/< volume name >/\_data

# touch f1 f2 f3



1. Now we will use the EC2 **PublicIP** of our Docker Host to access the **nginx webpage** from **nginx-container**



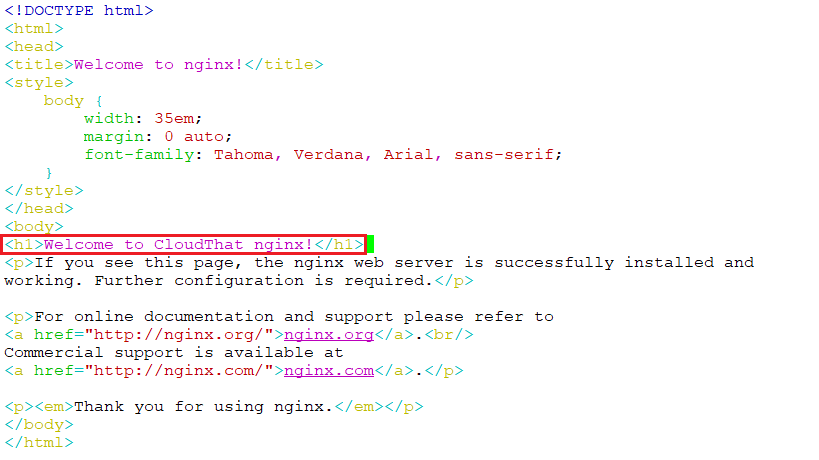
1. Modify the **index.html** of **nginx-container** from **Host mount location**, as shown below

# vi /var/lib/docker/volumes/< volume name >/\_data/index.html

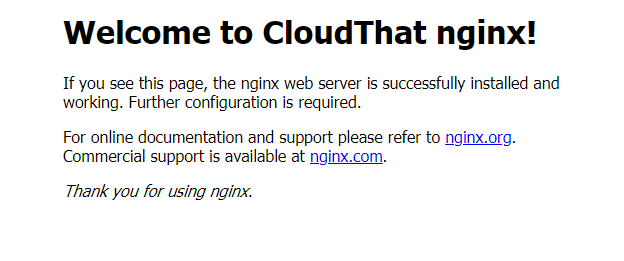
# cd /home/ubuntu/

Graphical user interface, text, application

Description automatically generated



1. Now again use the EC2 **PublicIP** of the instance and check for the changes



## Task 3: Deleting container and attaching the volume to another container

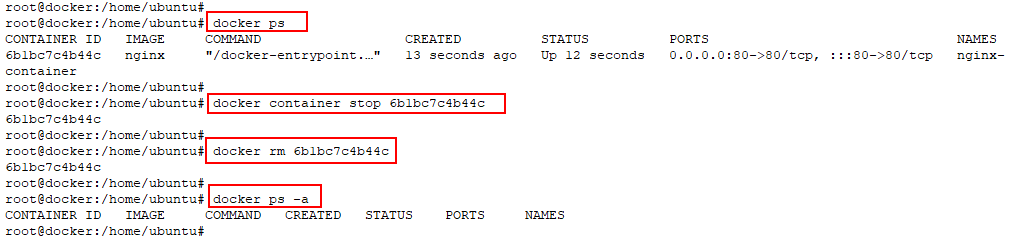
1. Stop the container using **docker stop command** and remove the container using **docker container rm.**

# docker ps

# docker container stop < container name/id >

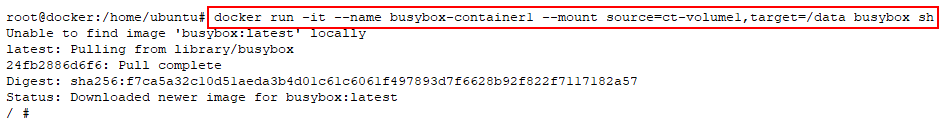
# docker rm < container name/id >

# docker ps -a



1. Create a new **busybox-container** andattach the existing volume to the container

# docker run -it --name busybox-container1 --mount source=< volume name >,target=/data busybox sh

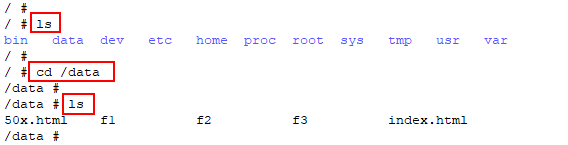


1. Navigate to the volume directory which is **/data** directory inside the container and check for the files created in the last task is present

# ls

# cd /data

# ls



1. Exit from the container, stop the container and remove the container

# exit

# docker stop <container name>

# docker rm <container name>

# docker ps -a

Graphical user interface, text, application, email

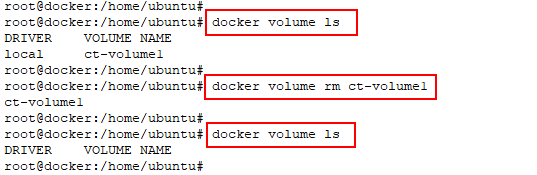
Description automatically generated

1. Remove the volume created using **docker volume rm command** and verify

# docker volume ls

# docker volume rm <volume name>

# docker volume ls



## Task 4: Create a container with tmpfs mount and verify it

1. As there is no ‘source’ required in tmpfs mount, use the below command with ‘—mount’ flag with type=tmpfs and destination options.

# docker run -d \

> -it \

> --name tmpmount \

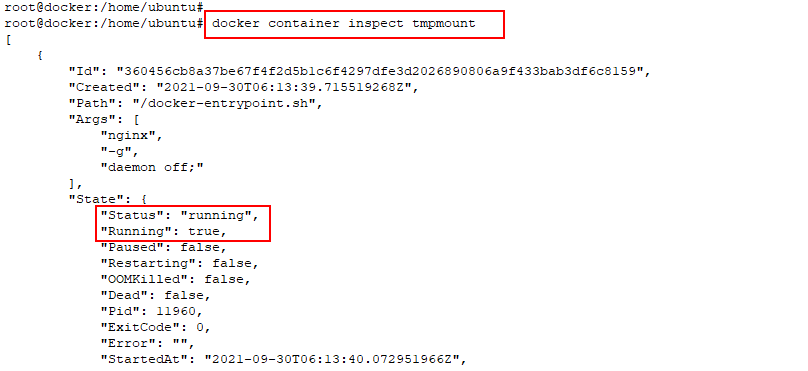
> --mount type=tmpfs,destination=/app \

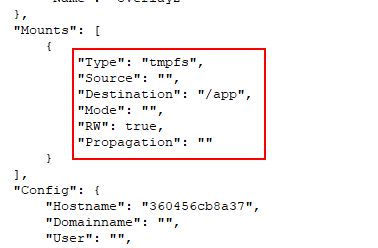
> nginx:latest



1. Verify the mount is tmpfs by running the below given command

# docker container inspect tmpmount





# Chapter Three: Docker Networking

**Topics covered in this unit:**

* Container Linking
* Binding Containers to Host Port or a Unix Socket
* Docker0 Bridge
* Configuring DNS
* Multi-Host Networking

## Introduction

Networking by Docker provides the ability for the containers to communicate with the host, with other containers and containers across hosts. Whether containers ports are exposed or not, every container on the same host will be accessible to the host system by default. Exposing the container port helps in documenting the port use, and this information is available for automated mappings and linkings.

For establishing communication, containers exposes their ports to the host, on which they receive traffic forwarded from the outside world. Exposed ports of the container can be mapped to the host system, either by selecting a specific host port or allowing Docker to choose a random, high, unused port. Docker handles the forwarding rules and iptables configuration to correctly route packets in such situations.

Docker can automatically configure iptables rules to allow for forwarding and configures NAT masquerading for originating traffic ondocker0 destined for the outside world.

## Container Linking

Docker containers can connect to one another with a Docker linking system that allows you to link multiple containers together and send connection information from one to another. Container linking helps to share information about a source container to a recipient container.

Containers can be linked together so that they can communicate and discover each other and transfer information from one container to another securely. When a link is set up, a secure channel is created between the source and recipient container. The recipient can then access select data about the source.

A source container is created and to create a link between containers, the –link flag is used when creating the recipient container. Using the -- link flag, Docker creates a secure tunnel between the containers that don’t need to expose any ports externally on the source container.

## Binding Containers to Host Port or a Unix Socket

Containers can be bind to a specific Host port or a UNIX socket, which is not advisable. Docker containers can be bind to a specific TCP port by changing the default docker binding. This increases the security risks of Docker networking as this setup allows non-root users to gain root access on the host. The security risk increases as the binding to the TCP port allows anybody with the access to that port to have full access over the host.

Docker daemon can be made to listen on a specific IP and port with –H flag. By default, Docker daemon will listen on **unix:///var/run/docker.sock.** This allows only local connections by the *root* user. This can be modified to 0.0.0.0:2375 or a specific host IP to give access to everybody, but that is not recommended because then the host is vulnerable for someone to gain root access to the host where the daemon is running.

## Docker Bridge

When the Docker process starts, a new virtual bridge interface called docker0 is configured on the host system. This interface creates a virtual subnet in Docker for use among the containers that are started. The bridge will serve as the main point of interface between networking within a container and networking on the host.

Docker uses virtual Ethernet (veth) bridge and automatically forwards packets between other network interfaces that are attached to it. The bridge allows the container to communicate with both host as well as other containers. Every time a Docker container is created, it creates a pair of peer interfaces, one is assigned to the container, and other is named uniquely (ex: veth0000 which out of namespace of host machine) which kept by Docker itself. By binding every “veth\*” interface to the docker0 bridge, Docker creates a virtual subnet shared between the host machine and every container.

## Configuring DNS

The hostname and DNS configuration is supplied to each container by docker without the need of building a custom image with hostname written within the image. Docker achieves this by overlaying three crucial /etc. files inside each container with virtual files where it can write fresh information. This is done at the start of each container. The /etc. files mount point can be obtained by executing the “mount” command inside the container.

This arrangement allows Docker to keep resolv.conf file updated across all containers when the host machine receives new configuration over DHCP. This configuration varies across different versions of Docker. Hence the following Docker options can be used:

Four options that affect the container DNS:

* ‘-h HOSTNAME’ or ‘- -hostname=HOSTNAME’
* ‘- -link=container\_name\_or\_ID:ALIAS’
* ‘- -dns=ip\_address’
* ‘- -dns-search= DOMAIN

‘-h HOSTNAME’ or ‘- -hostname=HOSTNAME’

This option sets the hostname by which the container knows itself. The host entries are made in “/etc/hostname”. Also, entries will be made into “/etc/hosts” as the name of the containers host-facing IP address. It is not easy to see the host name from outside the container. It will neither appear in ‘docker ps’ command, nor in the ‘/etc/hosts’ file of any other container.

‘- -link=container\_name\_or\_ID:ALIAS’

The container which is started using the --link option will have an extra entry named ‘ALIAS’ inside its ‘/etc/hosts’. The ‘ALIAS’ will point to the ip address of the container identified by the container name or id. The processes inside the new container connect to the hostname ‘ALIAS’ without having to know the IP of the container launched with - -link tag. The container name used in - -link option should either be an auto-assigned Docker name, or the user-defined name given using the ‘- -name’ tag. Hostname of a container cannot be used, as docker will not recognize it in the context of the ‘- -link’ option.

‘- -dns=ip\_address’

This option sets the IP address added as a server line to the ‘/etc/resolv.conf’ file of the container. When confronted with a host name that is not available in ‘/etc/hosts’, the processes in the container will connect to these ip addresses on port 53 looking for name resolution services.

‘- -dns-search= DOMAIN’

This option will set the domain names that are searched when a bare unqualified hostname is used inside of the container. This is done by writing ‘search’ lines inside the container’s ‘/etc/resolv.conf’ file. Whenever an attempt to access the host is made by a process of a container and if a search domain, for example – ‘test.com’ is set, the DNS logic will not only look up to host, but also the ‘host.test.com’.

## Multi Host Networking

User can create their own user defined networks that isolates the containers. You can create overlay network, Network plugin, Remote network with a custom specification. More than one container can be added into one network. Container can communicate with other containers within network but not across network.

Overlay Network also supports multi-host networking natively. It requires valid key-value service. Currently, libKU library is used. Now Docker Host in a network will be able to communicate each container. Each host should run a Docker engine inside it. The UDP (4789) and TCP (7946) should be open.

To create Network following is the command

$ docker network create <networkname>

Diagram

Description automatically generated

Illustration of Multiple Host in one Docker Container

Now launch the container with specifying Network name, once all host are connected All containers can communicate with each other.

To run the container use the following command:

$ docker run -itd --net=my-multi-host-network busybox

**Docker Container Networks**

The networking feature of docker provides more security for web applications running on containers. Docker Networking gives us complete isolation on containers from other containers as well as hosts to enhance security.

We will discuss the types of networks available for Docker in the following section.

**Docker Network**

We can find three networks created automatically when we install Docker. We can list these networks using the following command

**#sudo docker network ls**

**NETWORK ID NAME DRIVER**

87305a73fad6 bridge bridge

819f0be25214 host host

0987171459a5 none null

Here you can find three types of Network:

* Bridge
* Host
* None

These three are the Dockers network implementation. We must use **--net=<NETWORK>** option to specify which network option our container should use.

**Bridge Networking Mode**

The following discussion provides us an overview of default network that Docker Engine chooses when starting a container. The Bridge network is the default network in Docker.

We can view the bridge network details using the following command.

# docker network inspect bridge

[ { "Name": "bridge",

"Id": "87305a73fad618de3f9a24575731856e358cfa38b0b2702c3674d12cdbf9cf64",

"Scope": "local",

"Driver": "bridge",

"EnableIPv6": false,

"IPAM": {"Driver": "default",

"Options": null,

"Config": [{"Subnet": "172.17.0.0/16"}] },

"Internal": false,

"Containers": {},

"Options": {

"com.docker.network.bridge.default\_bridge": "true",

"com.docker.network.bridge.enable\_icc": "true",

"com.docker.network.bridge.enable\_ip\_masquerade": "true",

"com.docker.network.bridge.host\_binding\_ipv4": "0.0.0.0",

"com.docker.network.bridge.name": "docker0",

"com.docker.network.driver.mtu": "1500"

},

"Labels": {}

} ]

The bridge network name here is **docker0**. When we start a container, each container gets its own virtual Ethernet interface connected to the default Docker bridge, and an IP address is allocated to this virtual interface.

This bridge is the part of host network stack, use the following command to view this.

# ifconfig

docker0 ink encap:Ethernet HWaddr 02:42:f6:55:5e:9b

Inet addr:172.17.0.1  Bcast:0.0.0.0  Mask:255.255.0.0

UP BROADCAST MULTICAST  MTU:1500  Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:0

RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

The bridge will forward packets between any other network interfaces that are attached to it and allows containers to communicate with the host machine as well.

**None Networking Mode**

This type allows the container to be added to any container-specific network. This network disables the network interface from Docker. This is useful when we are using our own custom networking.

Use --net flag with none as an argument **--net=none** to get this network.

Get inside the container and use ifconfig command to view network stack:

# ifconfig

lo Link encap:Local Loopback

inet addr:127.0.0.1  Mask:255.0.0.0

inet6 addr: ::1/128 Scope:Host

UP LOOPBACK RUNNING  MTU:65536  Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:0

RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

**Host Networking Mode**

In host mode networking, the container shares the networking namespace of the host. Here, containers are directly exposed to a public network; we should have a port mapping for coordination.

Use –net with the host as an argument **--net=host** to get this network interface.

Here, the containers will have the same IP address as the host because here it inherits the IP address from the host. This is faster than the docker0 bridge networking because here, there are no routing overheads.

For example:

root@ip-10-0-4-46:/# ifconfig | grep -A 2  eth0

eth0 Link encap:Ethernet  HWaddr 0a:d2:98:ee:82:47

inet addr:10.0.4.46  Bcast:10.0.4.255  Mask:255.255.255.0

inet6 addr: fe80::8d2:98ff:feee:8247/64 Scope:Link

Here you see, container has the same IP address as the host (10.0.4.46).

**Overlay Networking Mode**

In Overlay mode networking we can connect multiple Docker daemons together, swarm services can be enabled to communicate with each other. This type of networking mode is used to allow communication between standalone container and swarm services, or between two standalone containers on different hosts having Docker daemons.

Overlay network driver helps in creating distributed network among multiple hosts running Docker daemon. It sits on top of host network and helps containers to communicate ith each other. Docker handles routing of packets to and from the correct Docker daemon host and the container.

On creation of Swarm cluster, two new networks are created on that Docker host:

* **Ingress**: This is the overlay network which handles control and data traffic related to swarm services. On creation of swarm service, it joins by default to the ingress network if we do not specify any custom overlay network.
* **docker\_gwbridge**: This is the bridge network which connects the individual Docker daemon to the other daemons in the swarm cluster.

# Lab 6: Docker Networking

## Task 1: Create a new docker bridge and check connectivity between containers of same bridge

1. Check the networks available in docker host by **docker network ls** command, it displays the default **bridge, host** and **none** networks

# docker network ls

Text

Description automatically generated

1. Create a new bridge network named **ct-bridge1** with the **docker network create command**

# docker network create --driver bridge ct-bridge1

Graphical user interface

Description automatically generated with low confidence

1. **Inspect** the newly created **ct-bridge1** bridge network and see the IP address range

# docker network inspect ct-bridge1

Text

Description automatically generated

1. List the **docker** **networks** again to see the new network

# docker network ls

Text

Description automatically generated

1. Create two **containers** with name **ct-c1** and **ct-c2** in the newly created bridge network **ct-bridge1** using **docker run command** with --**network** option, use **busybox** image for **ct-c1** container

# docker run -it --network ct-bridge1 --name=ct-c1 busybox

Graphical user interface, text

Description automatically generated

1. Press ‘**control+p+q’** to detach the container and switch to shell of docker host
2. Create second container named **ct-c2**, using **busybox** image

# docker run -it --network ct-bridge1 --name=ct-c2 **busybox**

A picture containing shape

Description automatically generated

1. Press ‘**control+p+q’** to detach the container and switch to shell of docker host
2. Inspect **ct-bridge1** network again. See the bridge network now has the details of the two containers created in that network

# docker network inspect ct-bridge1

Text

Description automatically generated

1. Check for the running status of two running containers with **docker ps command**

# docker ps

Text

Description automatically generated

1. Attach to the shell of **ct-c2** container by **docker attach** command

# docker attach ct-c2

A picture containing graphical user interface

Description automatically generated

1. **Ping** the **ct-c1 container** in the same bridge from **ct-c2 container** using **container name** and verify that **ping** gets a reply, hence proving service discovery using container names within same Docker bridge network

# ip addr

# ping -c 5 ct-c1

Text

Description automatically generated

1. Press ‘**control+p+q’** to detach the container and continue it to run in background

## Task 2: Create a new docker bridge and check connectivity between containers of different bridges

1. Create a second bridge network named **ct-bridge2** with the **docker network create command**

# docker network create --driver bridge ct-bridge2

Graphical user interface, application

Description automatically generated with medium confidence

1. Create two containers with the name **ct-c3** and **ct-c4** with busybox image in **ct-bridge2** bridge network by the given commands. Press ‘**control+p+q’** to detach the container and to switch to docker host shell

# docker run -it --network ct-bridge2 --name=ct-c3 busybox

# docker run -it --network ct-bridge2 --name=ct-c4 busybox

Graphical user interface, text, application

Description automatically generated

1. Attach to the shell of **ct-c4** container by the **docker attach command**

# docker attach ct-c4



1. **Ping** the **ct-c3 container** in the same bridge from **ct-c4 container** using **container name** and verify that **ping** gets a reply as they are on same bridge network

/ # ping -c 5 ct-c3

/ # ip addr

Table

Description automatically generated

1. **Ping** the **ct-c1 container** and **ct-c2 container** in the different bridge from **ct-c4 container** using **container name** and verify that **ping fails to resolve the container names** and **shows error name or service unknown**, hence proves service discovery using container names between docker bridge networks are not possible, hence bridge networks provides complete network isolation for containers. Press ‘**control+p+q’** to detach the container and to switch to docker host shell.

/ # ping -c 5 ct-c1

/ # ping -c 5 ct-c2

A screenshot of a computer

Description automatically generated with medium confidence

## Task 3: Using ‘Docker network connect’ command create a successful connection between containers of different bridges

1. Use the network ls command as shown below to see all networks

# docker network ls

Text

Description automatically generated

1. Use the docker network connect command to connect the container ct-c1 to ct-bridge2 network

# docker network connect ct-bridge2 ct-c1



1. Use the **docker network inspect** command to verify the bridge network to have the container which we configured in step 2.

# docker network inspect ct-bridge2

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

1. Since ct-c4 container is already connected to ct-bridge2, there is no need to run a command trying to connect to it; and it is already verified in step 2 above.
2. Login to ct-c1 container using **docker attach** command and ping ct-c4 container. Also, use ip addr command to verify the network interfaces associated with ct-c1 container.

# docker attach ct-c1

/ # ping -c 5 ct-c4

/ # ip addr

Text

Description automatically generated

1. Check the routes configured in ct-c1 container using the ip route command

/ # ip route

Text

Description automatically generated

## Task 4: Launch a container to host network

1. Press ‘**control+p+q’** to detach the container and continue it to run in background
2. Use the **docker run** command to start a container ct-c5 with **busybox** image and using **--network** option to run it in **host** network

# docker run -it --network host --name=ct-c5 busybox

A picture containing diagram

Description automatically generated

1. Use the command **ip addr** to view the network interfaces associated with ct-c5 container. Press ‘**control+p+q’** to detach the container .Notice the eth0 interface IP address is the same as the virtual machine(host) IP address

/ # ip addr

# ifconfig

Text, letter

Description automatically generated

Text, letter

Description automatically generated

Text, letter

Description automatically generated

1. Use the command **docker network inspect host** command to verifythe newly created network and the associated container ct-c5

# docker network inspect host

Graphical user interface, text, application, email

Description automatically generated

Text

Description automatically generated with medium confidence

## Task 5: Launch a container to none network

1. Use the **docker run** command to start a container ct-c6 with **busybox** image and using **--network** option to run it in **none** network

# docker run -it --network none --name=ct-c6 busybox

A picture containing diagram

Description automatically generated

1. Use the command **ip addr** to view the network interfaces associated with ct-c6 container. Notice there are no IP addresses assigned to the container except for the loopback address 127.0.0.1

/ # ip addr

Text, letter

Description automatically generated

1. Press ‘**control+p+q’** to detach the container and continue it to run in background

# Chapter Four: Dockerfile

Topics covered in this unit:

* Dockerfile Introduction
* Automating a Build Using Dockerfile
* RUN Instruction
* FROM Instruction
* Docker cmd and Entrypoint
* Environment Variables

## Dockerfile Introduction

Dockerfile is a text document which contains a set of instructions, using which Docker builds images automatically by reading those instructions one by one. The Dockerfile contains all the commands which are required by a user to assemble an image. The main purpose of Dockerfile is to automate the building process and to reuse it if required in the future. Without Docker file, the user needs to do all the commands again one by one, but if same commands are set up inside a Dockerfile, it can be used any number of times in future.

## Automating a Build Using Dockerfile

Another great usage of Dockerfile is to set up an environment with automated builds. These automated builds can be linked with GitHub/BitBucket account. The user can push all the related files which are needed by Dockerfile to build an image to a Git repository. Then the Git Repository can be linked with Docker hub account’s public or private repository. So, whenever there is a push to the GitHub/BitBucket repository, it will trigger a build and the image will be synced with the Docker hub repository. A user can also select a specific branch of GitHub/BitBucket account on which the trigger should be activated for every push. The build status and history can also be traced. To see the build status or history, go to the Docker hub repository’s information page, and then select build details.

## RUN Instruction

To execute any commands in a Docker file, the RUN instruction is used. The commands given in RUN instruction are executed in a new layer on top of the current image and commit the result to a new image. The resulting newly committed image will be used by the other step/instructions in the Dockerfile. Layering RUN instructions with commits being generated go with the core Docker concepts, where commits are cheap, and containers can be created from any point in an image’s history of commits.

The RUN instruction has two forms:

* **RUN <command>:** This form of RUN instruction will execute the command in a shell. i.e. “/bin/sh –c” shell form.
* **RUN [“executable”, “param1”, “param2”]:** This form of RUN instruction can be used when we want our own execution form. This form will allow us to avoid shell string mugging and to RUN commands using the base image that does not contain /bin/sh. In exec form, only double quotes should be used, as it is parsed as a JSON array.

## FROM Instruction

Every valid Dockerfile should start with a FROM instruction because the FROM instruction is used to set the base image on which the subsequent instructions will be executed. The base image used in FROM can be any valid image, either pulled from the public repositories or from the private repositories as per the requirements.

The FROM instruction will be the first non-comment instruction in the Dockerfile. FROM instruction can be used multiple times in a single Dockerfile to create multiple images. To keep track of the multiple images created from a single Dockerfile, keep a note of the last image id output by the commit before a new FROM instruction is used.

FROM <image>

FROM <image>:<tag>

FROM <image>@<digest>

The above listed are few of the example syntax of FROM instructions. The tag and digest are optional values. If no tag or digest option is given, the Docker builder assumes the latest version by default. If the given tag value is not present in the repository, the builder will give an error.

## Docker Cmd and Entrypoint

CMD and ENTRYPOINT instructions are used to configure the executable to be run when a container is started from an image. If the image must run without any additional command line arguments with Docker run command, the user must use ENTRYPOINT or CMD. Trying to run an image without ENTRYPOINT or CMD and without any command line arguments will result in an error. Most of the images found on the Docker’s public registry uses a shell like /bin/sh or /bin/bash as the CMD executable so that anyone who runs those images will get into the interactive shell by default with used with options –I, i.e., interactive shell and –t for the pseudo-TTY terminal.

Only one CMD instruction per Dockerfile is accepted. If multiple CMD instructions are given, Docker will only take the last CMD and ignore the previous ones. CMD and ENTRYPOINT can be used together, to provide the default executable for our image with default arguments. In this case, the user and not the executable can override the arguments.

**CMD Instruction has three forms:**

* **CMD [“executable”, “param1”, “param2”]:** It is an executable form with arguments. This is the preferred form.
* **CMD [“param1”, “param2”]:** It will provide default parameters/arguments to the ENTRYPOINT instruction.
* **CMD command param1 param2:** It will provide default parameters/arguments to the ENTRYPOINT instruction.

**ENTRYPOINT Instruction has two forms**:

* **ENTRYPOINT [“executable”, “param1”, “param2”]:** It is the executable form with default arguments/parameters. These arguments/parameters can be overridden if used with CMD. This is the most preferred form.
* **ENTRYPOINT [“command”,” param1”, “param2”]:** It is the shell form, which will execute the given command in a shell.

## Docker Environment Variables

The environment variables can be set from a Dockerfile using the ENV instruction. The ENV instruction will set an environment variable <Variable Name. Key> with a user-defined value <value>. The value will be present in the environment for all the descendent Dockerfile commands and can also be replaced inline in many commands as well.

**The ENV instruction has 2 forms:**

ENV <KEY><VALUE>

ENV <KEY1>=<VALUE1><KEY2>=<VALUE2><KEY3>=<VALUE3

The first form can only set one single variable. The entire string after the first space will be treated as value to the key, including all the whitespaces and quotes. Whereas the second form can be used when we must set multiple environment variables with a single instruction. Like command line parsing, quotes or backslashes can be used to include spaces within values.

# Lab 7: Building a Dockerfile to setup an Ubuntu container with WordPress application

## Task 1: Deploying MySQL and WordPress containers

Dockerfile helps to automate the steps of launching the container and configure the container when it is launching. Dockerfile uses the basic DSL with instructions for building Docker images. The Dockerfile approach is recommended over Docker commit because it provides a more repeatable, transparent, and idempotent mechanism for creating images.

1. Create a **directory** to save the Dockerfile and the dependent files. Create a file with name “**Dockerfile**” that has the sequence of steps for launching the container

# mkdir wordpress

# cd wordpress

# vi Dockerfile

Graphical user interface, text, application

Description automatically generated

1. This directory is our build environment; this is the build context. Hence, Dockerfile is placed in this directory for the build. Enter the below lines of code in the **Dockerfile**

FROM ubuntu:18.04

MAINTAINER ADMIN "admin@cloudthat.com"

ENV DEBIAN\_FRONTEND=noninteractive

RUN apt-get update && \

apt-get -q -y install apache2 \

php7.2 \

php7.2-fpm \

php7.2-mysql \

libapache2-mod-php7.2

ADD http://wordpress.org/latest.tar.gz /tmp

RUN tar xzvf /tmp/latest.tar.gz -C /tmp \

&& cp -R /tmp/wordpress/\* /var/www/html

RUN rm /var/www/html/index.html && \

chown -R www-data:www-data /var/www/html

EXPOSE 80

CMD ["/bin/bash","-c","service apache2 start && sleep 5000"]



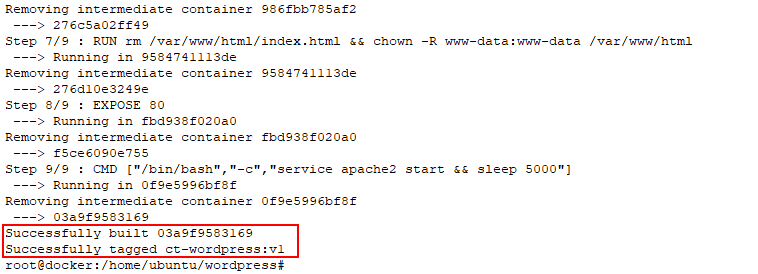
1. This is the Dockerfile to build the container with **WordPress**. It follows the following sequence:
2. **Pulls** the base **Ubuntu:18.04 image** from the public repo
3. Installs **Apache & php** on Ubuntu
4. Downloads **WordPress** in the container
5. **Extract** the **zip** file for WordPress
6. **Copies** the contents of the **WordPress** folder **to /var/www/html**, which is the default Document Root of Apache
7. **Removes** the **default index.html** from the Document Root
8. **Change** the **ownership** of the folder to **www-data**, which is the user for Apache
9. **CMD**: this is where all commands are executed inside the container

**Run** the **Docker build** in the directory where the Dockerfile is located. The command to build a container with Dockerfile:

# docker build -t ct-wordpress:v1 .

A screenshot of a social media post

Description automatically generated

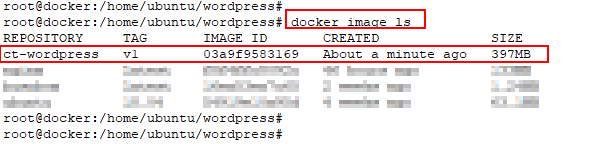


Here, build denotes the image must be built from a file named “Dockerfile”. **-t** denotes the addition of a tag to the image after building. “ct-wordpress” is the name given to the image that is created by the Dockerfile.

**Note:** Do not forget the “.” at the end of the command which is the current directory path.

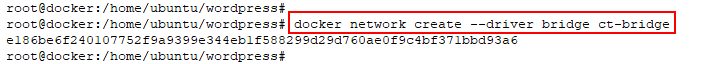
1. The Dockerfile is built and an image is generated in the Docker host. To find the image in the Docker host, enter the following command

# docker image ls



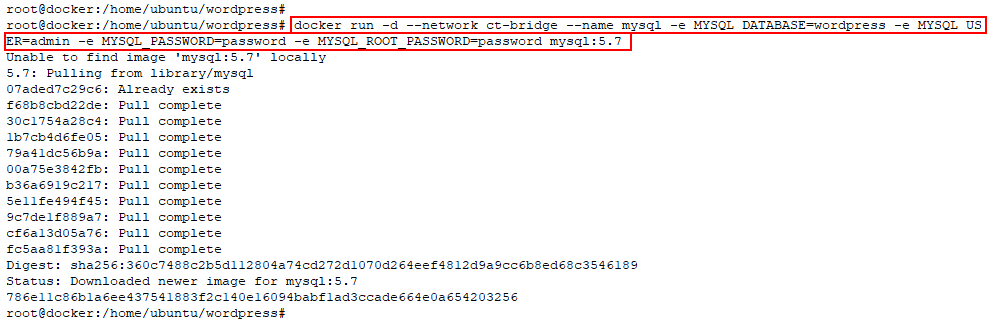
1. Create a new bridge network named **ct-bridge** with the **docker network create** command

# docker network create --driver bridge ct-bridge



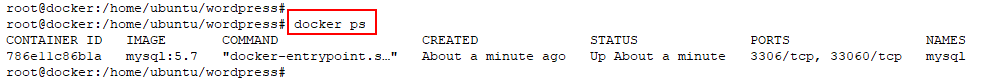
1. Launch a MySQL container in the above created bridge network using the **docker run** command

# docker run -d --network ct-bridge --name mysql -e MYSQL\_DATABASE=wordpress -e MYSQL\_USER=admin -e MYSQL\_PASSWORD=password -e MYSQL\_ROOT\_PASSWORD=password mysql:5.7



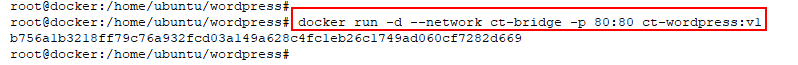
1. Check the running containers using docker ps command

# docker ps



1. Also, run a WordPress container in the same bridge network which we created above using the docker run command

# docker run -d --network ct-bridge -p 80:80 ct-wordpress:v1



1. Check the running containers using docker ps command

# docker ps



## Task 2: Deploying WordPress

1. Visit the WordPress site with the respective port to which we have mapped the WordPress container, for example, **"Public\_IP:80"** and click on ‘**Continue’**

A screenshot of a social media post

Description automatically generated

1. Click “**Let’s go!**” which takes to the database configuration page. Which requires the database details to be entered in the **“wp-config.php**” file

A screenshot of a social media post

Description automatically generated

1. **Database setup**

Enter the Database details for the WordPress to store the data.

A screenshot of a social media post

Description automatically generated

1. **Installation**

Once the database details are accepted, then move ahead to “Run the installation.”

A screenshot of a cell phone

Description automatically generated

**Note:** Remember the username and password that is set here to login the WordPress site again.

1. **Website Information**

This takes us to the installation page of WordPress, where the required fields to log in to the WordPress site have to be mentioned.

A screenshot of a social media post

Description automatically generated

A screenshot of a cell phone

Description automatically generated

1. **Admin login**

The login page of WordPress appears where we need to input the username and password which we configured in step 5 above

A screenshot of a cell phone

Description automatically generated

1. **Admin dashboard**

The WordPress dashboard appears with the Site name as **Docker Training** and with the username. Here we can add new posts, comments, media files, etc.

A screenshot of a social media post

Description automatically generated

# Chapter Five: Docker Hub

**Topics covered in this unit:**

* Committing Changes
* Private Registry
* Automated Builds

## Docker Hub

Docker Hub is a cloud-based registry service for building and shipping application or service containers. These containers enclose a piece of software in a complete file system which needs to run the code, runtime, system tools, system libraries that can install on a server.

It provides a centralized resource for storing Docker images on the cloud, with distribution and changing the management. It helps in workflow automation throughout the development lifecycle.

## Committing Changes

Any changes made to the base image must be committed to the latest version of the image, to make sure that the change persists. The Docker images are stored in a sequence of read-onlylayers. Whenever a new container has been started with the base image, which is of read-only format and creates a new layer on top of that with read-write permissions. Whenever an existing file is modified by the container, the file is copied out of the underlying read-only layer to the newly created read-write layer, and the changes are added. The read-write layer’s version of the file hides the underlying file of the read-only layer but does not destroy it. It will exist as it was in the underlying image. Once the Docker container is deleted or committed to a new version of the image, re-launching the base image will start a fresh container without any of the changes made in the previously running container, because those changes are not reflected in the base image, but only to the new layer created on top of it. And if the layer created on top of the base image is not committed to a new version of the image, the changes done to the base image will be lost.

## Private Registry

Private Registry serves the need of having a secure registry to share the images within the organization. Docker Hub offers only one private repository per account for free. Premium membership offers more number of private repositories based on the package selected. Our own private registry can also be created using Dockers private registry server container. The repositories of our private registry can be stored either on local disk or on s3. Private Registry uses SSL certificate to validate the authorization for Docker private registry server. The private registry is very secure and reliable. Self-Signed Certificates can also be used, but SSL is recommended.

## Automated Builds

Automated builds build clusters to automatically create images from a GitHub or BitBucket repository containing Dockerfile. It starts its work by adding a commit hook to the GitHub or BitBucket triggering a build which will update whenever you push commit. The image can only be modified by adding the required changes to the Dockerfile and then committing it to the GitHub/ BitBucket repository; the image will be uploaded to Docker hub registry and mark it as Automated Build.

## Advantages of Automated Builds

* Automated builds maintain the repository up to date
* Dockerfile can be accessed by anybody who can access the repository on the Docker Hub registry

# Lab 8: Pushing the image to DockerHub

1. Create a DockerHub account at [https://hub.docker.com](https://hub.docker.com/)

A screenshot of a cell phone

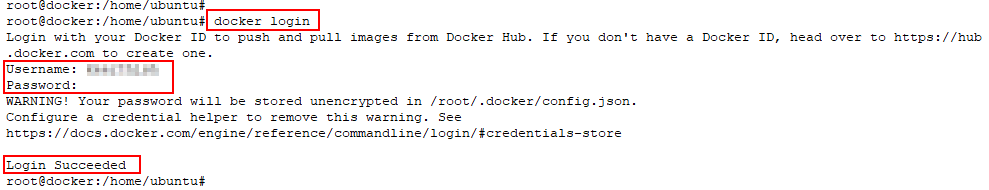
Description automatically generated

1. Once Sign Up is completed and you are logged in, click on repositories where the images that you push will be shown later

A screenshot of a social media post

Description automatically generated

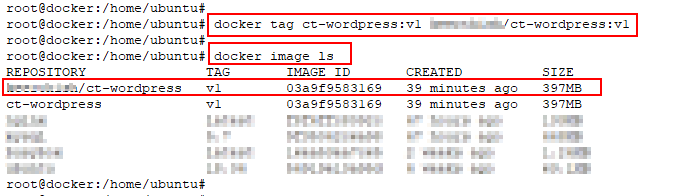
1. In your VM terminal Login to your DockerHub account using the **docker login** command; provide your DockerHub username and password when prompted and make sure the login is successful



1. Now tag **ct-wordpress:v1** image as **<your dockerhub account name>/ct-wordpress:v1** for pushing the image to DockerHub. Verify the tag is correct

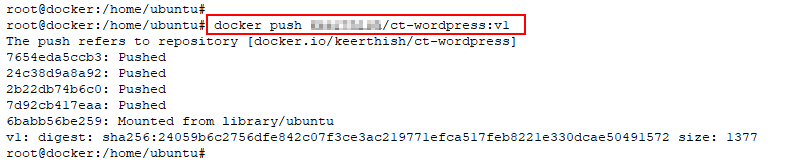
# docker tag ct-wordpress:v1 <dockerhub account>/ct-wordpress:v1

# docker image ls

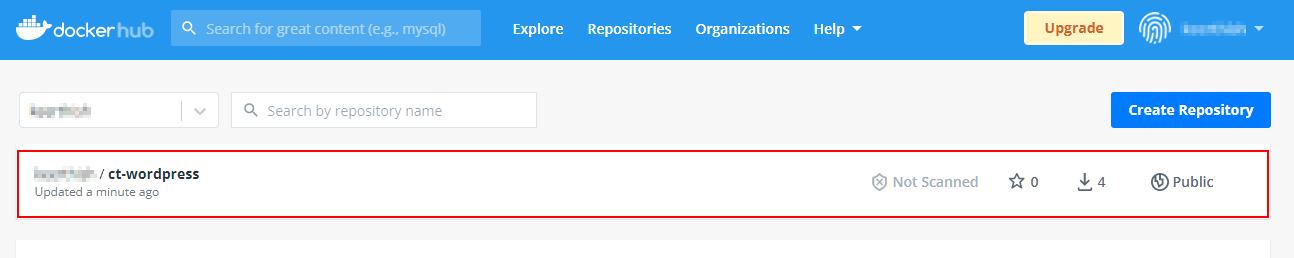


1. Push the Docker image to DockerHub by using **docker push** command, you will see only the committed layer getting pushed while the layers which are already present in DockerHub are only getting mounted

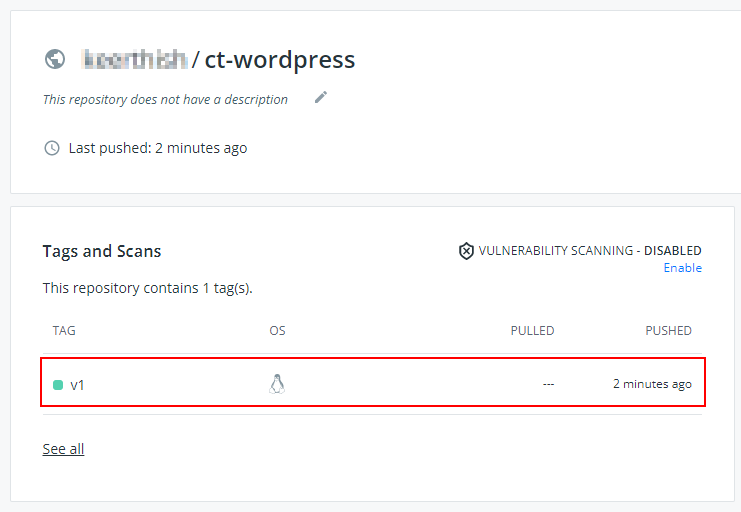
# docker push <dockerhub account>/ct-wordpress:v1



1. Go to your DockerHub account in the browser and refresh the repositories page to verify the creation of a new repository **ct-wordpress**



1. Verify the tag of the pushed image as well, which, as per the steps followed above, is v1



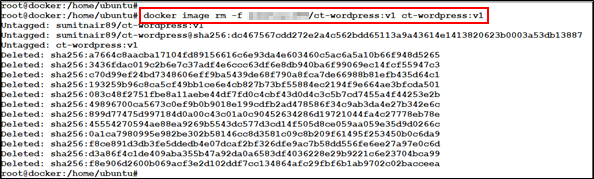
1. Make a note of the Docker images in host machine by using below command

# docker image ls



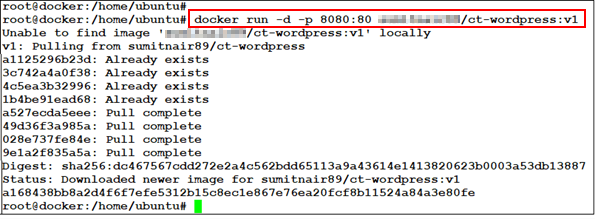
1. Remove all tagged images except for the MySQL and Ubuntu image. Use the **docker image rm** command

# docker image rm -f <dockerhub account>/ct-wordpress:v1 ct-wordpress:v1



1. Now use your image which is pushed to your Docker Hub to create a new container in **detached** mode, you will see the pull is happening only for the missing layer.

# docker run -d -p 8080:80 <dockerhub account>/ct-wordpress:v1



1. Open the browser and access your WordPress website using **Public\_IP:8080**

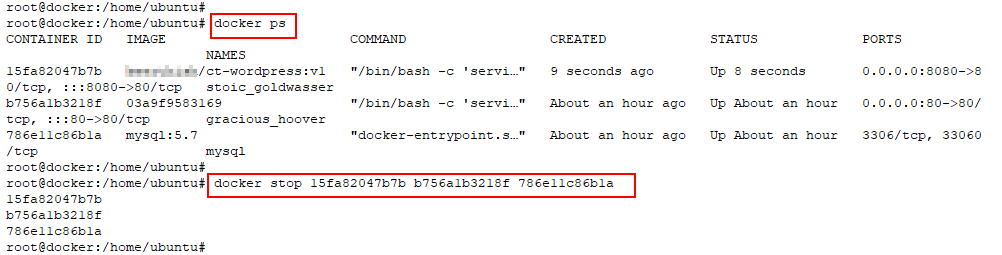
A screenshot of a social media post

Description automatically generated

1. Stop the running containers by using below command

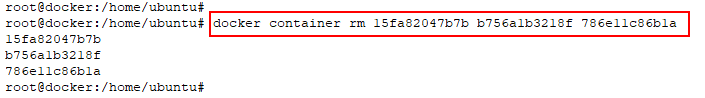
# docker ps

# docker stop < container id/name > < container id/name >



1. Delete the stopped containers completely

# docker container rm < container id/name > < container id/name >



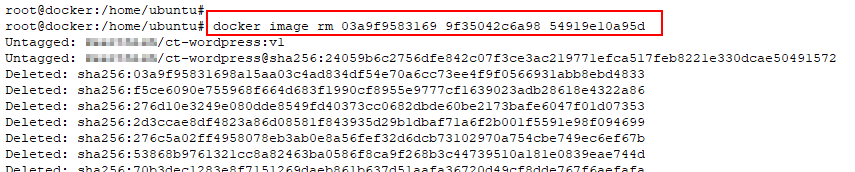
1. List all the images present in Docker host and make note of image IDs

# docker image ls



1. Delete the docker images by using below command

# docker image rm < image id/name > < image id/name >



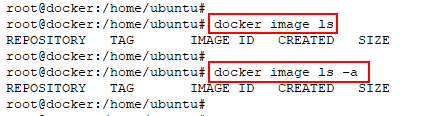
A picture containing text, newspaper

Description automatically generated

1. Execute the below command to make sure all the images are deleted

# docker image ls

# docker image ls -a



Lab 9: Building Efficient Docker Images

Task 1: Create a golang Docker Image to print Hello World

1. Make sure you are in home directory (/home/ubuntu)

$ pwd

Graphical user interface, application

Description automatically generated

1. Create 2 directories – large and optimal, and change current directory to large using below command

$ mkdir large optimal && cd large

Shape

Description automatically generated with medium confidence

1. Create a helloworld.go file which prints **hello world**

$ vi helloworld.go

A picture containing graphical user interface

Description automatically generated

package main

import "fmt"

func main() {

fmt.Println("hello world")

}

Logo

Description automatically generated with medium confidence

1. Create a Docker file using below

$ vi Dockerfile

Schematic

Description automatically generated with medium confidence

FROM golang:1.12.5

WORKDIR /app

COPY helloworld.go .

RUN GOOS=linux go build -a -installsuffix cgo -o helloworld .

CMD ["./helloworld"]

A picture containing text

Description automatically generated

1. Build the Dockerfile to create an image using docker build as below.

$ docker build -t hello-go:v1 .

Text

Description automatically generated

1. Run the docker image as container and verify that it shows “hello world” as output

$ docker run hello-go:v1

Graphical user interface

Description automatically generated with medium confidence

1. Perform docker images to check the build image and the respective image sizes

$ docker image ls

Table

Description automatically generated

Task 2: Create an optimal Docker Image using Multi-stage Dockerfile

1. Change the directory to optimal and create another Dockerfile

$ cd ../optimal

A picture containing shape

Description automatically generated

1. Create a multi-stage docker file as below

$ vi Dockerfile

A picture containing schematic

Description automatically generated

FROM golang:1.12.5 AS stage1

WORKDIR /app

COPY helloworld.go .

RUN GOOS=linux go build -a -installsuffix cgo -o helloworld .

FROM alpine:3.9.3

WORKDIR /root

COPY --from=stage1 /app/helloworld .

CMD ["./helloworld"]

Graphical user interface, text, email

Description automatically generated

1. Copy the helloworld.go which we created earlier to the current directory

$ cp ../large/helloworld.go .

A picture containing shape

Description automatically generated

1. Build the Docker image with above Dockerfile using below command

$ docker build -t hello-go:v2 .

Text

Description automatically generated

1. Run the image as a container and verify that you get “hello world” as output

$ docker run hello-go:v2

Graphical user interface, application

Description automatically generated with medium confidence

1. Verify the docker image built from multi-stage docker file and image size as below.

$ docker image ls

As you can see, the image size is just over 7 MB as compared to the earlier image which is 784 MB

Table

Description automatically generated

1. Check the layers of both images hello-go:v1 and hello-go:v2 using below command

$ docker history hello-go:v1

Text

Description automatically generated

$ docker history hello-go:v2

Text

Description automatically generated

# Chapter Six: Docker Compose

**Topics covered in this unit:**

* Docker Compose
* Working of Docker Compose
* Features of Docker Compose

## Docker Compose Introduction

Docker Compose can be used for defining and running applications that require multiple containers. Docker compose file helps to configure your application services.

By using a single command, you can start multiple containers as services in isolated environments from the mentioned configuration in the docker-compose.yml file.

Individual applications running in individual containers can be launched and managed separately using the normal container concepts. Managing an application that requires multiple containers to be linked and managed is a cumbersome task. This state prevailed until Docker came up with the Orchestration tool called Docker Compose. The orchestration is the key factor which is required for managing multiple containers. Orchestration can be defined as "The automated arrangement, coordination, and management of multiple containers in a Docker environment".

The main concept of Docker compose is creating containers and linking them which is managing the micro-service architecture.

## Working of Docker Compose

There are mainly three steps for Docker compose:

* Define required commands to build the container image in **Dockerfile**
* Define the services and relationship(linking) between containers in **docker-compose.yml** file
* Use **docker-compose up** command to deploy your environment

Docker compose can help in managing the services which are running in a Docker-compose environment like start, stop or rebuilding the services. You are also able to list the status of running containers and get the outputs of the running services.

## Features of Docker Compose

* Having multiple separate environments on a single machine by giving project name to particular environment
* When Docker-compose up runs, this copies the old container's data to a new container. This helps not to lose your data in volumes

# Lab 10: Docker Compose

## Task 1: Docker Compose Installation

1. Run the following commands to install Docker compose

$ sudo su

# wget https://bootstrap.pypa.io/pip/2.7/get-pip.py

# apt install python -y

# python get-pip.py

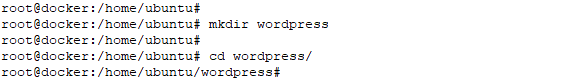
# pip install docker-compose

## Task 2: Create a docker-compose.yaml for Wordpress application

1. Create a directory named **wordpress** and change directory to **wordpress**

# mkdir wordpress

# cd wordpress



1. Create **docker-compose.yml** with below given contents

# vi docker-compose.yaml

version: '3.3'

services:

db:

image: mysql:5.7

volumes:

- db\_data:/var/lib/mysql

restart: always

environment:

MYSQL\_ROOT\_PASSWORD: somewordpress

MYSQL\_DATABASE: wordpress

MYSQL\_USER: wordpress

MYSQL\_PASSWORD: wordpress

wordpress:

depends\_on:

- db

image: wordpress:latest

ports:

- "80:80"

restart: always

environment:

WORDPRESS\_DB\_HOST: db:3306

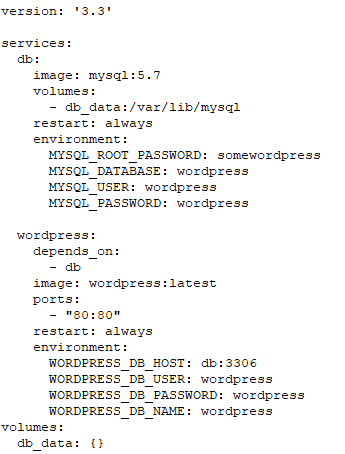
WORDPRESS\_DB\_USER: wordpress

WORDPRESS\_DB\_PASSWORD: wordpress

WORDPRESS\_DB\_NAME: wordpress

volumes:

db\_data: {}



1. Use **docker-compose up -d command** to bring up your entire microservice application stack

# docker-compose up -d

A screenshot of a social media post

Description automatically generated

1. Perform **docker-compose ps command** to see the service status

# docker-compose ps

A screenshot of a cell phone

Description automatically generated

1. Perform **docker image ls command** to see the **images** pulled by docker compose **wordpress:latest** and **mysql:5.7**

# docker image ls

A screenshot of a cell phone

Description automatically generated

1. Perform **docker container ls command** to see the running containers **wordpress\_wordpress\_1** and **wordpress\_db\_1**

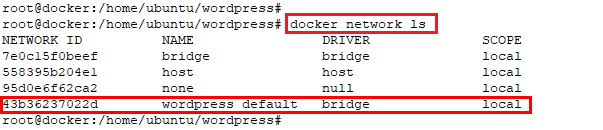
# docker container ls

A screenshot of a social media post

Description automatically generated

1. Perform **docker network ls command** to see the newly created isolated bridge network named **wordpress\_default**

# docker network ls



1. Perform **docker volume ls command** to see the newly created docker volume **wordpress\_db\_data** which is mounted with **wordpress\_db\_1** container and an unnamed volume mounted to **wordpress\_wordpress\_1**

# docker volume ls

A screenshot of a cell phone

Description automatically generated

1. Perform **docker inspect command** on various resources to see how connections worked

# docker <resource type > inspect < resource name/id >

1. Browse to **http://< Docker Host PublicIP >** from your web browser, and see your **wordpress** web site is ready, click on **Continue** to proceed with wordpress installation

A screenshot of a cell phone

Description automatically generated

1. Provide the required details and click on **Install Wordpress**

A screenshot of a cell phone

Description automatically generated

1. Click **Log In** in the Success message page

A screenshot of a cell phone

Description automatically generated

1. Provide the username and password given during installation, which was **admin** and **mypassword** in above screen shot then click on **Log In**

A screenshot of a cell phone

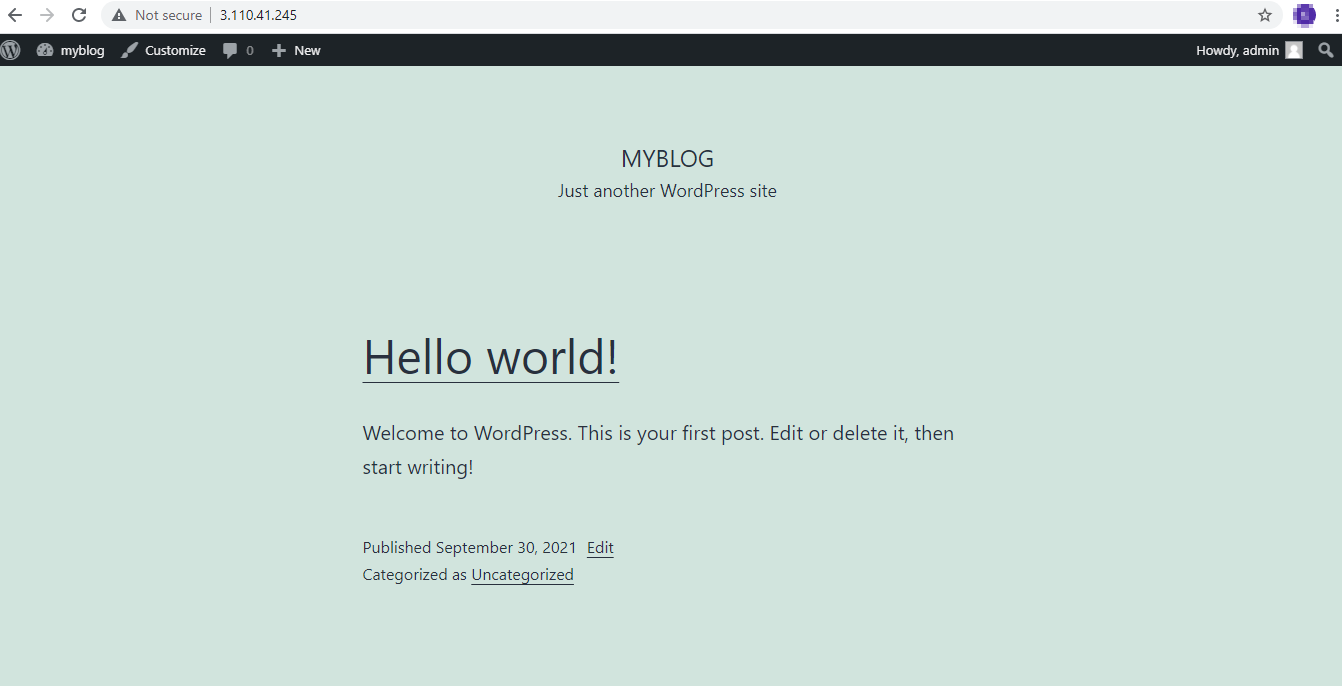
Description automatically generated

1. In wordpress admin page, expand **MyBlog** tab and click on **Visit Site**

A screenshot of a cell phone

Description automatically generated

1. See your wordpress site got created with in minutes



1. Terminate the entire application stack using **docker-compose down command**

# docker-compose down

A screenshot of a cell phone

Description automatically generated