# DOCKER

Introduction

What is Docker?

💡  *Wikipedia defines Docker as*  *an open-source project that automates the deployment of software applications inside containers by providing an additional layer of abstraction and automation of OS-level virtualization on Linux.*

"DOCKER" refers to several things. This includes an open-source community project which started in 2013; tools from the open-source project; Docker Inc., the company that is the primary supporter of that project; and the tools that the company formally supports.

**💡Tips: From Wikipedia**

Virtualization, in computing, refers to the act of creating a virtual (rather than actual) version of something, including but not limited to a virtual computer hardware platform, operating system (OS), storage device, or computer network resources.



* Docker is a tool that allows developers, sys-admins etc. to easily deploy their applications in a sandbox (called containers) to run on the host operating system i.e. Linux.
* The key benefit of Docker is that it allows users to package an application with all of its dependencies into a standardized unit for software development. Unlike virtual machines, containers do not have high overhead and hence enable more efficient usage of the underlying system and resources.

***Here's a quick explanation:***

* The IT software "Docker” is containerization technology that enables the creation and use of Linux® containers.
* The open source Docker community works to improve these technologies to benefit all users
* The company, Docker Inc., builds on the work of the Docker community, makes it more secure, and shares those advancements back to the greater community.

**Tips:**

With DOCKER, you can treat containers like extremely lightweight, modular virtual machines. And you get flexibility with those containers—you can create, deploy, copy, and move them from environment to environment, which helps optimize your apps for the cloud.

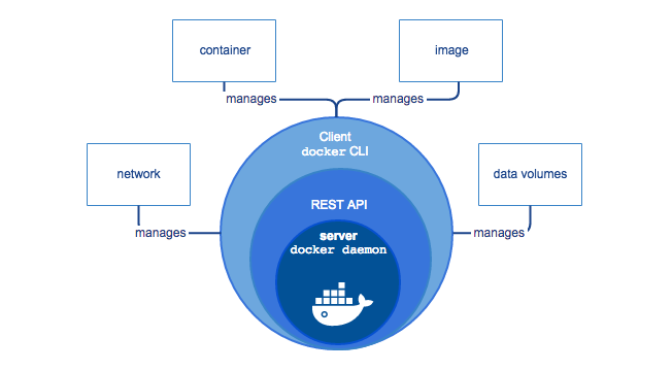


Q: What is a Docker?  
A: **Docker** is defined as the platform for containerizing the applications to isolate it from each other in order to ensure high availability and more efficiency irrespective of the environments such as **Development, Testing or Production**. All the application related dependencies such as libraries, jar files, server related configurations, infrastructure-related elements will be packaged and formed as a container called containerized applications which does not need any dependency and works independently. It ensures the application to be run irrespective of the external factors. Containers in Docker have support from Docker Engine and Host Operating System to support all the operational or infrastructural related dependencies.

Docker Engine

Docker Engine is a client-server application with these major components:

* A server which is a type of long-running program called a daemon process (the dockerd command).
* A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
* A command-line interface (CLI) client (the docker command).
* The CLI uses the Docker REST API to control or interact with the Docker daemon through scripting or direct CLI commands.



Docker Architecture

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your [Docker containers](https://lms.clarusway.com/mod/lesson/view.php?id=2158). The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets, or a network interface.



*The core components that compose Docker:*

* + - *The Docker client and server also called the Docker Engine.*
    - [*Docker Images*](https://lms.clarusway.com/mod/lesson/view.php?id=2159)
    - *Registries*
    - [*Docker Containers*](https://lms.clarusway.com/mod/lesson/view.php?id=2158)

**Docker Client and Server**

Docker is a client-server application. The Docker client talks to the Docker server or daemon, which, in turn, does all the work. You’ll also sometimes see the Docker daemon called the Docker Engine.

[**Docker images**](https://lms.clarusway.com/mod/lesson/view.php?id=2159)

Images are the building blocks of the Docker world. Containers are launched from images. [Docker images](https://lms.clarusway.com/mod/lesson/view.php?id=2159) are light-weight, portable, reproduce-able, and declarative.

**Registries**

Docker stores the images you build in registries. There are two types of registries: public and private. Docker, Inc., operates the public registry for images, called the Docker Hub. You can create an account on the Docker Hub and use it to share and store your images. You can also re-use the registry repository capability for free at your premises to store your images.

**Containers**

Docker helps you build and deploy containers that encapsulate your packages, applications, infrastructures, and services. [Docker containers](https://lms.clarusway.com/mod/lesson/view.php?id=2158) try to maximize resource sharing between containers and allow us to isolate those parts of the applications that are different and need their own space.

Because of **high resource sharing**, we can build [Docker containers](https://lms.clarusway.com/mod/lesson/view.php?id=2158) with a **small footprint** that makes application **distribution easier** and it makes container **startup times faster**.

Q: What are the components of Docker Architecture and explain?  
A: The Docker works on a client-server architecture. The Docker client establishes communication with the Docker Daemon. The Docker client and Daemon can run on the same system. A Docker client can also be connected to a remote Docker Daemon.

* **Docker Client:** This performs Docker build pull and run operations to establish communication with the Docker Host. The Docker command uses Docker API to call the queries to be run.
* **Docker Host:** This component contains Docker Daemon, Containers and its images. The images will be the kind of metadata for the applications which are containerized in the containers. The Docker Daemon establishes a connection with Registry.
* **Registry:**This component will be storing the [Docker images](https://lms.clarusway.com/mod/lesson/view.php?id=2159). The public registries are Docker Hub and Docker Cloud which can be used by anyone.

### **Docker vs. VM's**

#### **What are VMs?**

A virtual machine (VM) is an emulation of a computer system. Put simply, it makes it possible to run what appears to be many separate computers on hardware that is actually one computer.

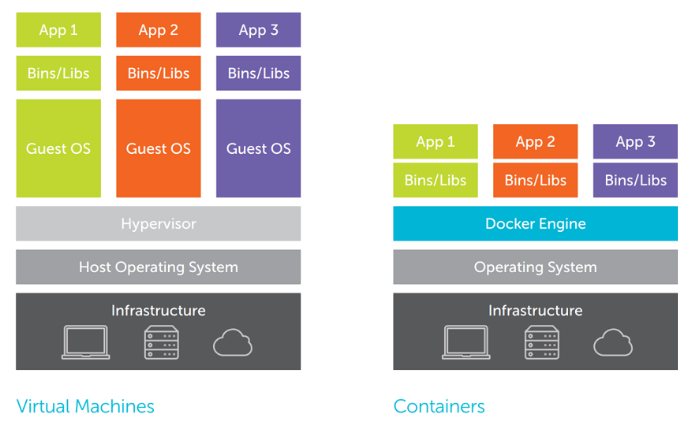
#### **Difference Between VM and Docker**

Virtual machines have a full OS with its own memory management installed with the associated overhead of virtual device drivers. In a virtual machine, valuable resources are emulated for the guest OS and hypervisor, which makes it possible to run many instances of one or more operating systems in parallel on a single machine (or host). Every guest OS runs as an individual entity from the host system. Hence, we can look at it an independent full-fledged house where we don't share any resources as shown below:



[Docker containers](https://lms.clarusway.com/mod/lesson/view.php?id=2158) are executed with the Docker engine rather than the hypervisor. Containers are therefore smaller than Virtual Machines and enable **faster startup** with **better performance**, **less isolation** and **greater compatibility** possible due to **sharing** of the host’s kernel. Hence, it looks very similar to the residential flats system where we share resources of the building.

**💡Tips:**  
  *From Wikipedia:* A hypervisor (or virtual machine monitor, VMM) is computer software, firmware or hardware that creates and runs virtual machines. A computer on which a hypervisor runs one or more virtual machines is called a host machine, and each virtual machine is called a guest machine. The hypervisor presents the guest operating systems with a virtual operating platform and manages the execution of the guest operating systems. Multiple instances of a variety of operating systems may share the virtualized hardware resources.



Virtual Machines are built over the physical hardware, there is a layer of Hypervisor which sits between physical hardware and operating systems. In a broader view, Hypervisor is used to virtualize the hardware which is then configured with the way a user wants it to. Unlike virtual machines where hypervisor divides physical hardware into parts, Containers are like normal operating system processes.

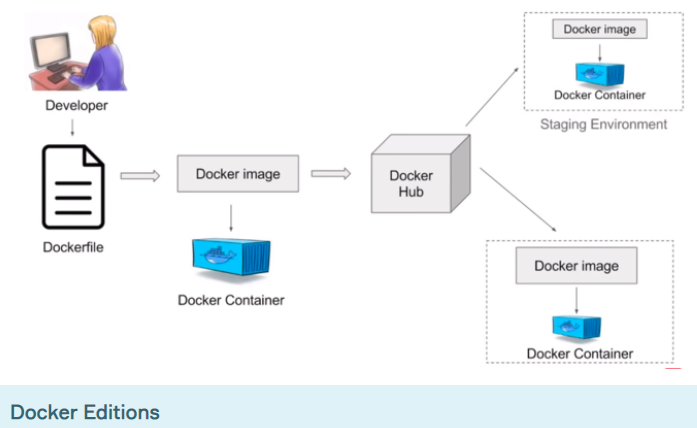
| [Docker Containers](https://lms.clarusway.com/mod/lesson/view.php?id=2158) versus Virtual Machines (VM) | |
| --- | --- |
| **Docker** | **Virtual Machines** |
| All containers share the same kernel of the host | Each VM runs its own OS |
| Containers initiated in seconds | Boot-up time is in minutes |
| Images are built incrementally on top of another like layers. Lots of images/snapshots | VMs snapshots are used sparingly |
| Images can be diffed and can be version controlled. Dockerhub is like GitHub | Not effective diffs. Not version controlled |
| Can run many [Docker containers](https://lms.clarusway.com/mod/lesson/view.php?id=2158) on a laptop. | Cannot run more than a couple of VMS on an average laptop |
| Multiple [Docker containers](https://lms.clarusway.com/mod/lesson/view.php?id=2158) can be started from one Docker image | Only one VM can be started from one set of VMX and VMDK files |

Q: What is the advantage of Docker over hypervisors?  
A: Docker is light weight and more efficient in terms of resource uses because it uses the host underlying kernel rather than creating its own hypervisor.

 Interview Q&A

Terminology

In this course, we will use a lot of Docker-specific jargon which might be confusing to some. So before we go further, we need to clarify some terminology that is used frequently in the Docker ecosystem.



* **Docker Community Edition (CE)** is ideal for Developers who are looking for experimenting docker and creating container-based applications. It’s free.
* **Docker Enterprise Edition (EE)** is a Containers-as-a-Service (CaaS) platform. Enterprise Edition Subscription packages include an integrated Docker platform and tooling for container management and security.

**Docker ID**

* Your free Docker ID grants you access to Docker services such as the Docker Store, Docker Cloud, Docker Hub repositories, and some beta programs. Your Docker ID becomes repository namespace used by hosted services such as Docker Hub and Docker Cloud. All you need is an email address.

**Registry**

* A Docker registry stores [Docker images](https://lms.clarusway.com/mod/lesson/view.php?id=2159).
* **Docker Hub (Like GitHub)** is a cloud-based registry service that allows you to link to code repositories, build your images and test them, stores manually pushed images, and links to Docker Cloud so you can deploy images to your hosts.
* **Docker Cloud** uses the hosted Docker Cloud Registry, which allows you to publish Dockerized images on the internet either publicly or privately. Docker Cloud can also store pre-built images, or link to your source code so it can build the code into [Docker images](https://lms.clarusway.com/mod/lesson/view.php?id=2159), and optionally test the resulting images before pushing them to a repository.

**Docker Client**

* The command-line tool that allows the user to interact with the daemon. It is the primary user interface to Docker. Accepts commands from the user and communicates back and forth with a Docker daemon.

**Docker Daemon**

* The background service running on the host that manages the building, running and distributing [Docker containers](https://lms.clarusway.com/mod/lesson/view.php?id=2158). Runs on a host machine.

**Dockerfile**

* Dockerfile is a text document that contains all commands a user could call on the command line to create an image.

**Docker Image**

* Docker image is a read-only template with instructions for creating a Docker container.

**Docker Container**

* Created from [Docker images](https://lms.clarusway.com/mod/lesson/view.php?id=2159) and run the actual application. It is a runnable instance of an image.

**Docker Compose**

* Compose is a tool for defining and running multi-container Docker applications.

Q: What are Docker Image and Docker Hub?  
A: The Docker Image is a set of files and a combination of parameters that will allow creating the instances to run in separate containers as an isolated process. The Docker hub is a kind of repository to the images where these images can be stored and this access is public. The Docker run command can be used to create the instance called container which can be run using the Docker image. Docker hub is the largest public repository of the image containers which is being maintained by the community of developers and individual contributors.

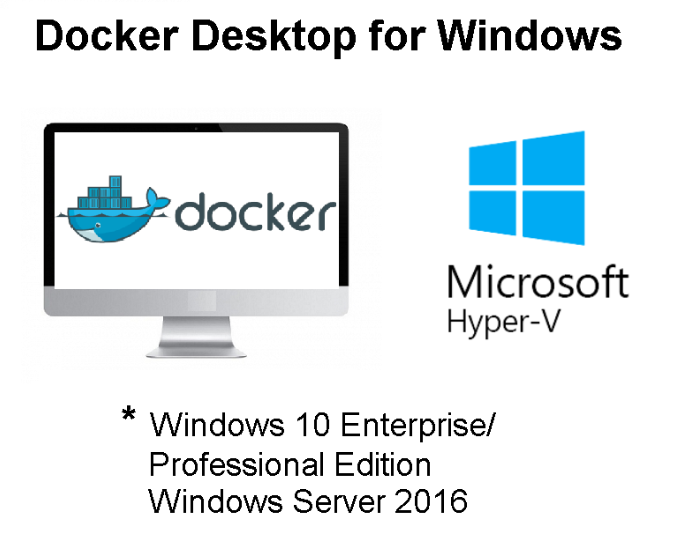
## **Docker Desktop for Windows**

### **Docker on Windows**

* Docker Desktop for Windows is the Community version of Docker for Microsoft Windows. You can download Docker Desktop for Windows from Docker Hub.

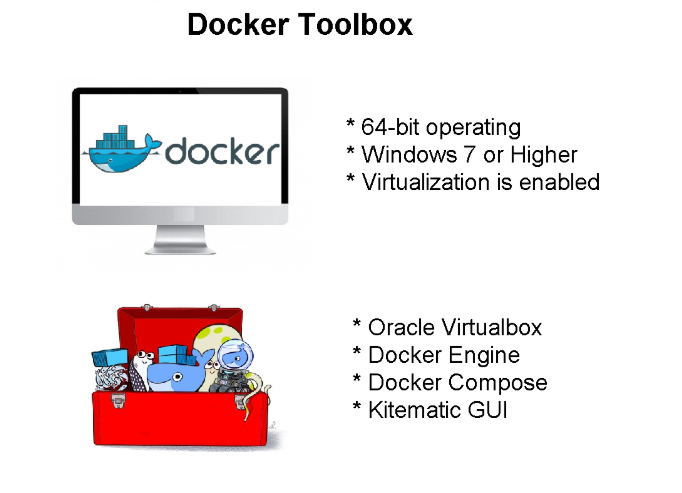
[Download from Docker Hub](https://hub.docker.com/editions/community/docker-ce-desktop-windows/)

The installation provides Docker Engine, Docker CLI client, Docker Compose, Docker Machine, and Kitematic. Containers and images created with Docker Desktop for Windows are shared between all user accounts on machines where it is installed. This is because all Windows accounts use the same VM to build and run containers.



**Avoid ! :What to know before you install**  
**System Requirements:**

* Windows 10 64-bit: Pro, Enterprise, or Education (Build 15063 or later).
* Hyper-V and Containers Windows features must be enabled.
* The following hardware prerequisites are required to successfully run Client Hyper-V on Windows 10:
  + 64 bit processor with Second Level Address Translation (SLAT)
  + 4GB system RAM
  + BIOS-level hardware virtualization support must be enabled in the BIOS settings.
* **Tips:**
* Docker Desktop for Windows Requires Microsoft Windows 10 Professional or Enterprise 64-bit. For previous versions get Docker Toolbox.
* [Download Docker Toolbox](https://docs.docker.com/toolbox/toolbox_install_windows/)



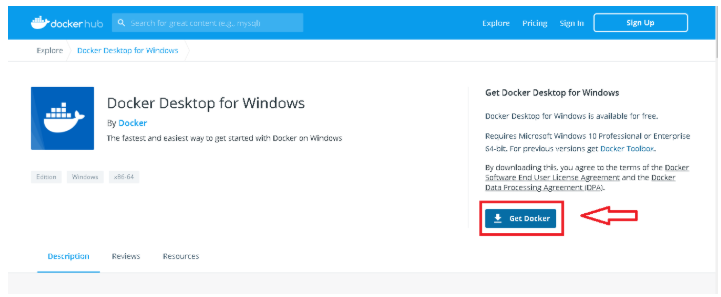
Docker Desktop for Windows

Install Docker Desktop on Windows

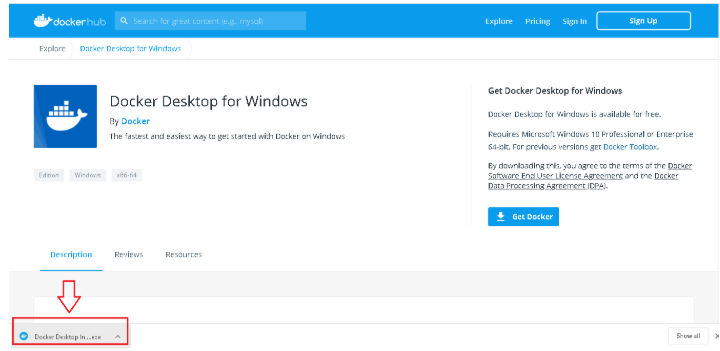
1. Download the Docker.

**💡Tips:**  
If you haven’t already downloaded the installer **(Docker Desktop Installer.exe)**, you can get it from;

[Download from Docker Hub](https://hub.docker.com/editions/community/docker-ce-desktop-windows/)



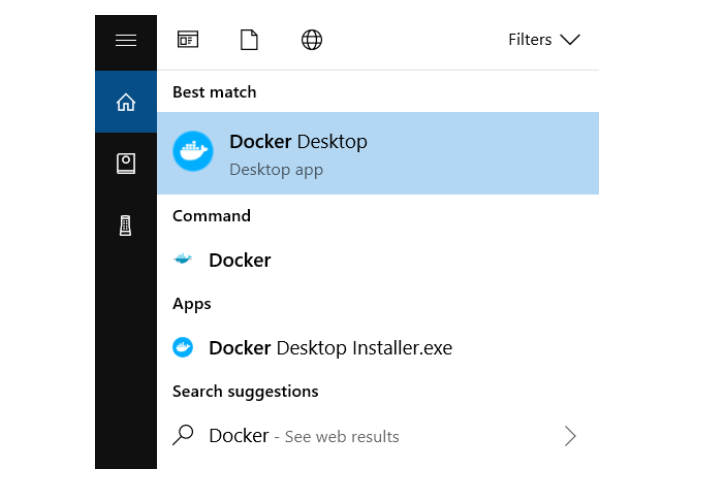
2. Double-click Docker **Desktop Installer.exe** to run the installer.



3. Follow the instructions on the installation wizard to accept the license, authorize the installer, and proceed with the install.

4. **Click Finish** on the setup complete dialog and launch the Docker Desktop application.

5. Docker Desktop does not start automatically after installation. To start Docker Desktop, search for Docker, and select Docker Desktop in the search results.

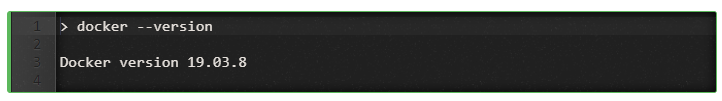


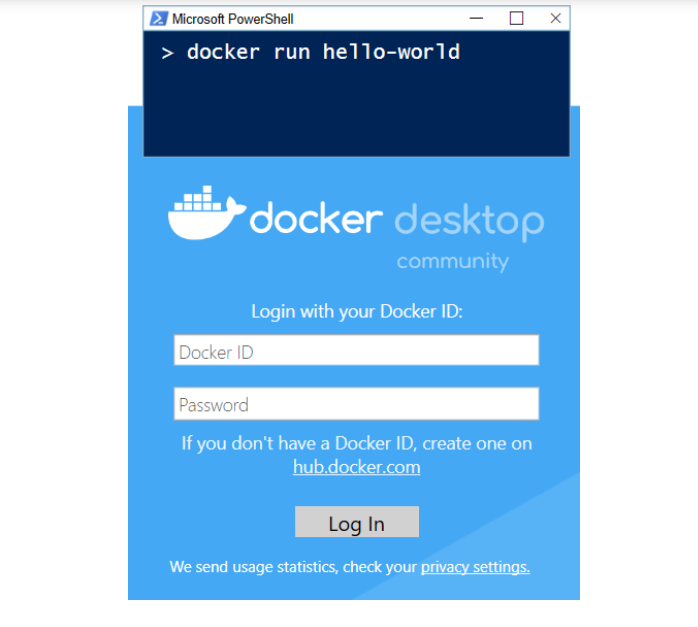
6. When the whale icon in the status bar stays steady, Docker Desktop is up-and-running and is accessible from any terminal window.



7. From now on, you can use Docker.

**💡Tips:**  
Run **docker version** to check the version.





### **Install Docker Toolbox on Windows**

**💡Tips:**

Docker Desktop for Windows Requires Microsoft Windows 10 Professional or Enterprise 64-bit. For previous versions get Docker Toolbox.

Docker Toolbox includes the following Docker tools:

* Docker CLI client for running Docker Engine to create images and containers
* Docker Machine so you can run Docker Engine commands from Windows terminals
* Docker Compose for running the docker-compose command
* Kitematic, the Docker GUI
* The Docker QuickStart shell preconfigured for a Docker command-line environment
* Oracle VM VirtualBox

**Avoid ! :**You can’t run Docker Engine natively on Windows, Because the Docker Engine daemon uses Linux-specific kernel features. Instead, you must use the Docker Machine command, docker-machine, to create and attach to a small Linux VM on your machine. This VM hosts Docker Engine for you on your Windows system.

1. Check your version.

**💡Tips:**

* **For Windows 10:** Run Speccy, and look at the CPU information.
* **For Windows 8 or 8.1** Choose **Start > Task Manager** and navigate to the Performance tab. Under **CPU** you should see.
* **For Windows 7 :**Run a tool like the Microsoft® Hardware-Assisted Virtualization Detection Tool or Speccy, and follow the on-screen instructions.

**⚠️Avoid ! :**Verify your Windows OS is 64-bit (x64).

2. Install Docker Toolbox.

3. To download the latest version of Docker Toolbox

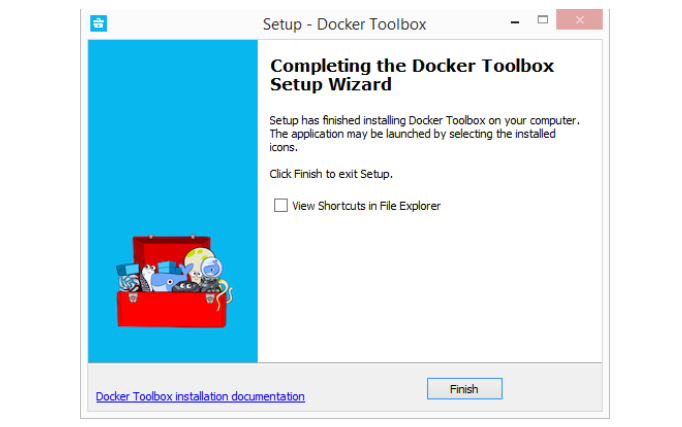
[Download Docker Toolbox](https://docs.docker.com/toolbox/toolbox_install_windows/)

4.Install Docker Toolbox by double-clicking the installer.

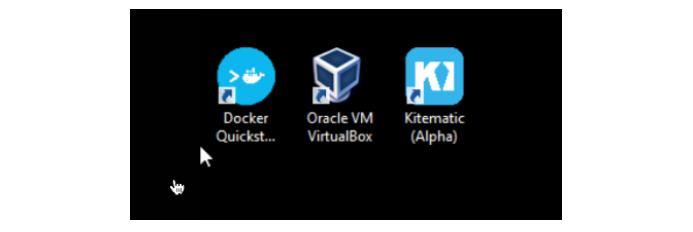
5.Press Next to accept all the defaults and then Install.



6. When it completes, the installer reports it was successful:



7. On your Desktop, find the Docker QuickStart Terminal icon.



8. Click the Docker QuickStart icon to launch a pre-configured Docker Toolbox terminal.

9. Click the Docker QuickStart icon to launch a pre-configured Docker Toolbox terminal.

10. Type the docker run hello-world command and press RETURN.

## **Docker Desktop for Mac**

### **Docker on Mac**

* Docker Desktop for Mac is the Community version of Docker for Mac. You can download Docker Desktop for Mac from Docker Hub.

[Download from Docker Hub](https://hub.docker.com/editions/community/docker-ce-desktop-mac/)

**⚠️Avoid ! :What to know before you install**  
System Requirements:

* Mac hardware must be a 2010 or a newer model
* macOS must be version 10.13 or newer.
* At least 4 GB of RAM.

**💡Tips:**

Docker Toolbox provides a way to use Docker on older Macs that do not meet minimal system requirements for Docker Desktop for Mac.

[Download Docker Toolbox](https://github.com/docker/toolbox/releases)

### **Install Docker Desktop on Mac**

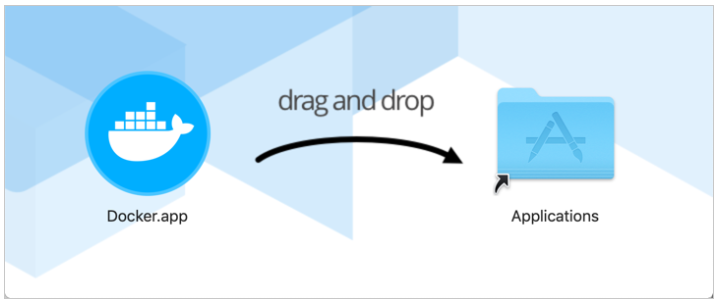
**1. Download the Docker.**

**💡Tips:**  
If you haven’t already downloaded the installer **(Docker Desktop Installer.exe)**, you can get it from;

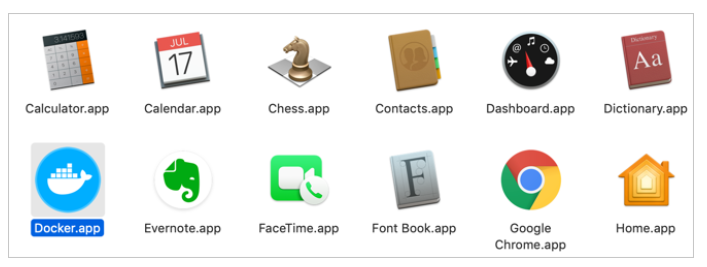
[Download from Docker Hub](https://hub.docker.com/editions/community/docker-ce-desktop-mac/)



**2. Double-click**Docker.dmg to open the installer, then drag the Docker icon to the Applications folder.



**3. Double-click Docker.app in the Applications folder to start Docker. (In the example below, the Applications folder is in “grid” view mode.)**



**4. The Docker menu in the top status bar indicates that Docker Desktop is running, and accessible from a terminal.**



**5. From now on, you can use Docker.**



### **Install Docker Toolbox on Mac**

**💡Tips:**

Docker Toolbox provides a way to use Docker on older Macs that do not meet minimal system requirements for Docker Desktop for Mac.

Docker Toolbox includes the following Docker tools:

* Docker CLI client for running Docker Engine to create images and containers
* Docker Machine so you can run Docker Engine commands from Windows terminals
* Docker Compose for running the docker-compose command
* Kitematic, the Docker GUI
* The Docker QuickStart shell preconfigured for a Docker command-line environment
* Oracle VM VirtualBox

**⚠️Avoid ! :**You can’t run Docker Engine natively on macOS with Docker Toolbox, because the Docker Engine daemon uses Linux-specific kernel features. Instead, you must use the Docker Machine command, docker-machine, to create and attach to a small Linux VM on your machine. This VM hosts Docker Engine for you on your Mac.

1. Check your version.

**💡Tips:**Your Mac must be running macOS 10.8 “Mountain Lion” or newer to run Docker software. To find out what version of the OS you have:

* Choose About this Mac from the Apple menu.
* If you have the correct version, go to the next step.

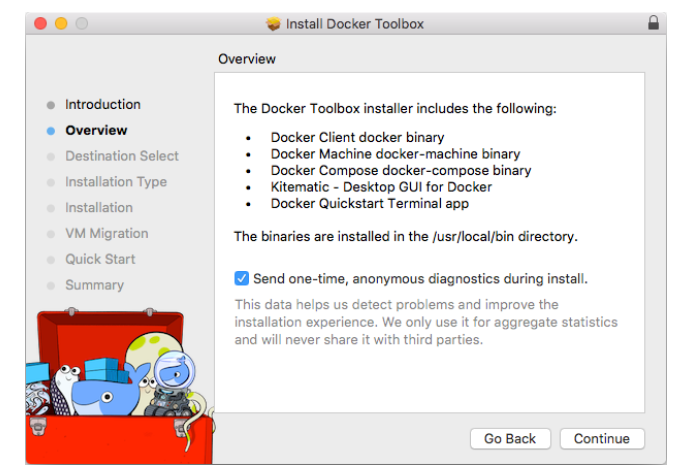
**⚠️Avoid ! :**If you aren’t using a supported version, you could consider upgrading your operating system.

2. Install Docker Toolbox.

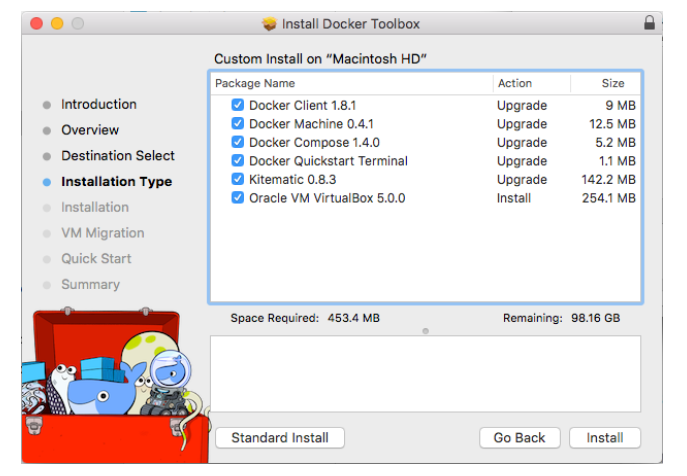
3. To download the latest version of Docker Toolbox, go to Toolbox Releases and download the latest .pkg file.

[Download Docker Toolbox.](https://github.com/docker/toolbox/releases)

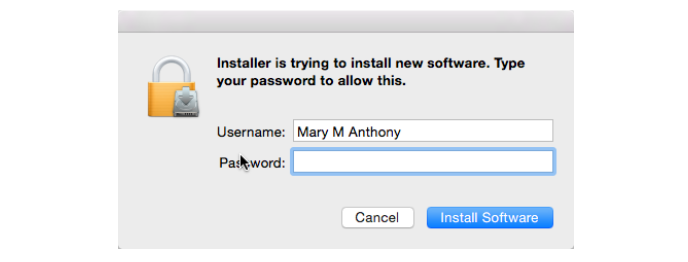
4. Install Docker Toolbox by double-clicking the package or by right-clicking and choosing “Open” from the pop-up menu.



5. Press Continue to install the toolbox.

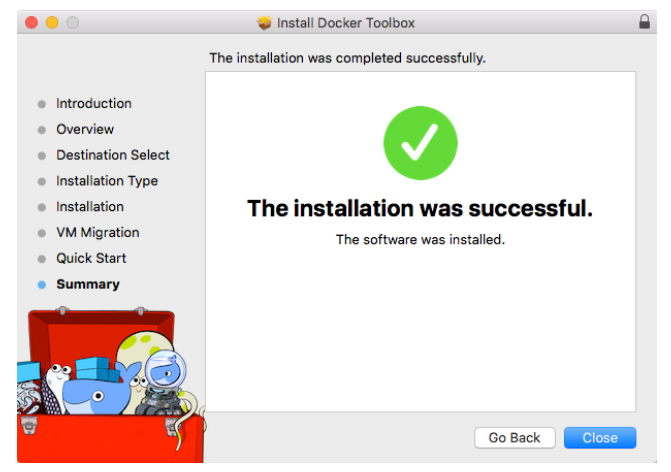


6. Press Install to perform the standard installation.



7. When it completes, the installer provides you with some shortcuts. You can ignore this for now and click Continue.

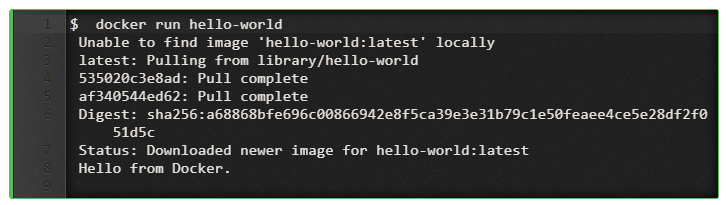




7. Click the icon to launch a Docker Quickstart Terminal window.



8. Click your mouse in the terminal window to make it active.



## **Docker for Linux**

### **Install Docker to Linux**

There are of course millions ways to install Docker on like a million different flavors of Linux, and we are just showing you installation of Amazon Linux 2 and Ubuntu Linux.

No matter your distribution of choice, you’ll need a 64-bit installation and a kernel at 3.10 or newer. Kernels older than 3.10 do not have the necessary features Docker requires to run containers; data loss and kernel panics occur frequently under certain conditions.

**💡Tips:**

Check your current Linux version with **uname -r**. You should see something like 3.10.[alphanumeric string].x86\_64.

### **Installing Docker on Amazon Linux 2 (ec2)**

1. Launch an instance with the Amazon Linux 2 AMI

2. Connect to your instance.

3. Update the installed packages and package cache on your instance.

[ec2-user@clarusway ~]$ sudo yum update –y

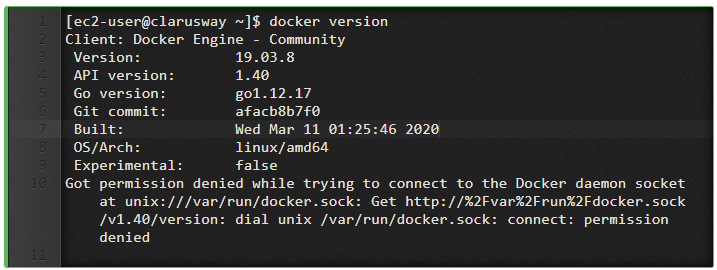
4.Install the most recent Docker Community Edition package.

[ec2-user@clarusway ~]$ sudo amazon-linux-extras install docker

5. Start the Docker service.

[ec2-user@clarusway ~]$ sudo service docker start

After Installation



6. Add the ec2-user to the docker group so you can execute Docker commands without using sudo.

[ec2-user@clarusway ~]$ sudo usermod -a -G docker ec2-user

**💡Tips:**

After adding your user name to the docker group,

* Run $ docker version
* You will see on screen Got this permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get http://%2Fvar%2Frun%2Fdocker.sock/v1.40/version: dial unix /var/run/docker.sock: connect: permission denied
* Solution: as a non-root user run $ newgrp docker commands.

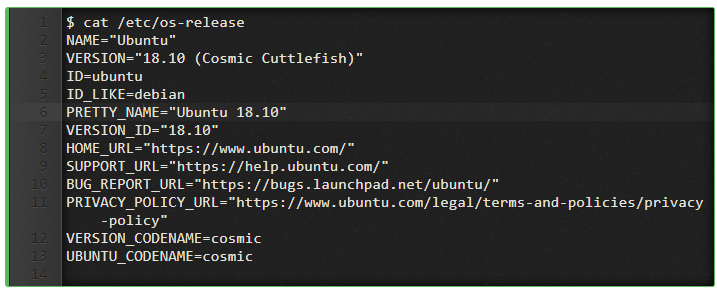
7. Verify that the ec2-user can run Docker commands without sudo.

[ec2-user@clarusway ~]$ docker info

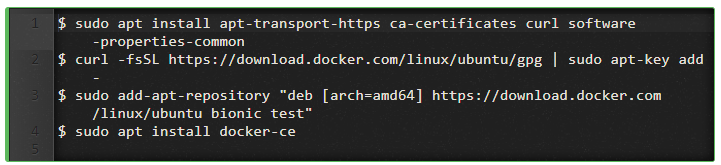
### **Installing Docker on Ubuntu**

### **Installing Docker 18.09 CE on Ubuntu 18.10**

Verifying Ubuntu 18.10 release



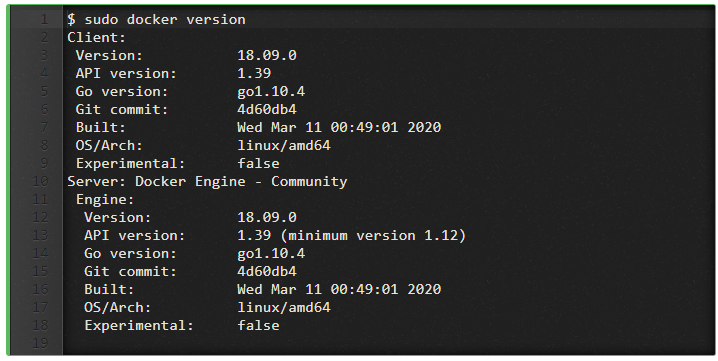
Installing Docker 18.09 Release



**Tips:**

If you want to run docker as non-root user then you need to add it to the docker group.

* $ sudo groupadd docker #Create the docker group if it does not exist
* $ sudo usermod -aG docker $USER #Add your user to the docker group
* $ newgrp docker #Run the following command or Logout and login again and run



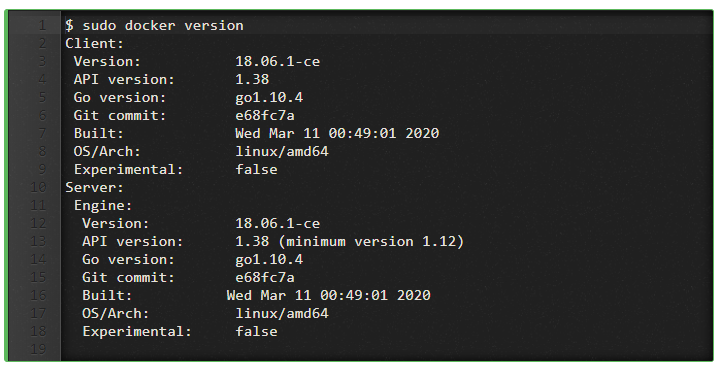
### **Installing Docker 18.06 CE on Ubuntu 18.11**

Updating Ubuntu 18.11 Repository



Using Snap to install Docker CE 18.06





## **Docker Containers**

### **What are containers?**

The industry standard today is to use Virtual Machines (VMs) to run software applications. VMs run applications inside a guest Operating System, which runs on virtual hardware powered by the server’s host OS.

VMs are great at providing full process isolation for applications: there are very few ways a problem in the host operating system can affect the software running in the guest operating system, and vice-versa. But this isolation comes at great cost — the computational overhead spent virtualizing hardware for a guest OS to use is substantial.

Containers take a different approach: by leveraging the low-level mechanics of the host operating system, containers provide most of the **isolation** of virtual machines at a **fraction of the computing power**.

Containers offer a **logical packaging** mechanism in which applications can be **abstracted** from the environment in which they actually run. This **decoupling** allows container-based applications to be deployed easily and consistently, regardless of whether the target environment is a private data center, the public cloud, or even a developer’s personal laptop. This gives developers the ability to create predictable environments that are isolated from the rest of the applications and can be run anywhere.

Q: What is Docker Container  
A: A Docker Container is a form of encapsulation to the application which holds all the dependencies which share the kernel with other containers in the duration of running the isolated processes on the host operating system. A Docker container can be created by creating a Docker image. These [Docker images](https://lms.clarusway.com/mod/lesson/view.php?id=2159) can be run after that using Docker commands. Docker containers are the instances of the [Docker images](https://lms.clarusway.com/mod/lesson/view.php?id=2159) at the runtime. [Docker images](https://lms.clarusway.com/mod/lesson/view.php?id=2159) can be stored in any public hosts or private hosts like Docker hub. Docker Image is a set of files which can be run in an isolated process.

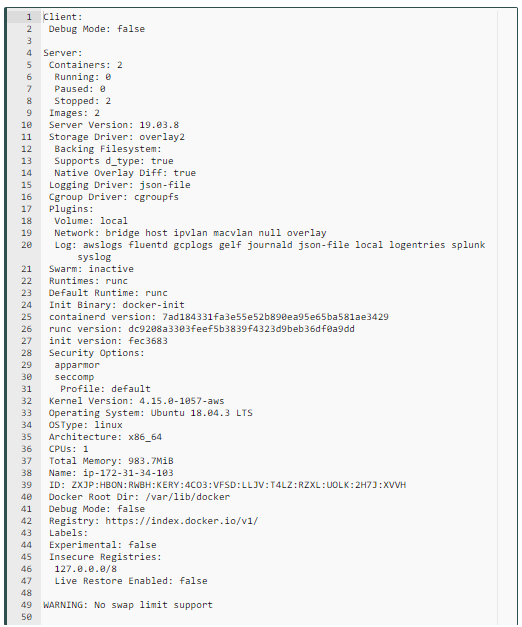
Run your first container

We’re going to start with checking that Docker is working correctly, and then we’re going to take a look at the basic Docker workflow: creating and managing containers. We’ll take a container through its typical lifecycle from creation to a managed state and then stop and remove it.

Firstly, let’s check that the docker **binary**exists and is functional:  
input :



Output:



Here, we’ve passed the info command to the docker binary, which returns a list of any containers, any images (the building blocks Docker uses to build containers), the execution and storage drivers. Docker is using, and its basic configuration.

**Your Container**

Now let’s try and launch our first container with Docker. We’re going to use the docker run command to create a container. The docker run command provides all of the ”launch” capabilities for Docker. We’ll be using it a lot to create new containers.

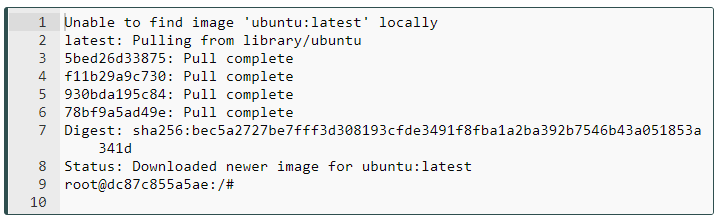
**💡Tips:**

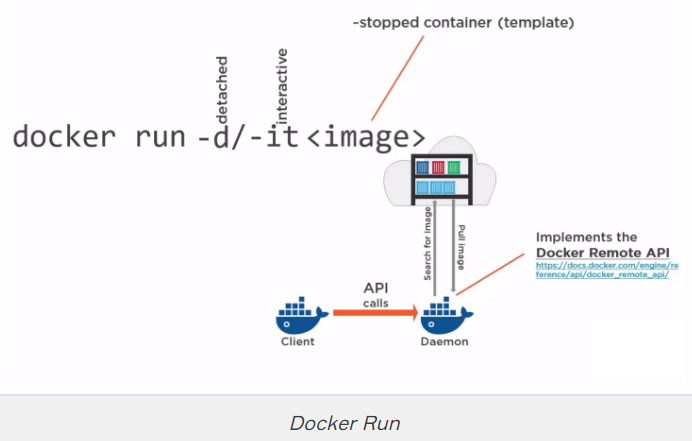
* You can find a full list of the available Docker commands [here](https://docs.docker.com/engine/reference/commandline/cli/)or by typing **docker help**. You can also use the Docker man pages (e.g., man docker-run).

input :



Output:





First, we told Docker to run a command using docker run. We passed it two command line flags: -i and -t.

The -i flag keeps STDIN open from the container, even if we’re not attached to it. This persistent standard input is one half of what we need for an interactive shell.

The -t flag is the other half and tells Docker to assign a pseudo -tty to the container we’re about to create. This provides us with an interactive shell in the new container. This line is the base configuration needed to create a container with which we plan to interact on the command line rather than run as a daemonized service.

**💡Tips:**

* You can find a full list of the available Docker run flags by typing **docker run help**. You can also use the Docker man pages (e.g., man docker-run).

We told Docker which image to use to create a container, in this case, the ubuntu image. The ubuntu image is a stock image, also known as a ”base” image, provided by Docker, Inc., on the Docker Hub registry.

So what was happening in the background here? Firstly, Docker checked locally for the ubuntu image. If it can’t find the image on our local Docker host, it will reach out to the Docker Hub registry run by Docker, Inc., and look for it there. Once Docker had found the image, it downloaded the image and stored it on the local host.

Docker then used this image to create a new container inside a filesystem. The container has a network, IP address, and a bridge interface to talk to the local host. Finally, we told Docker which command to run in our new container, in this case launching a Bash shell with the /bin/bash command.

## **Working with Container**

We are now logged into a new container, with the catchy ID of 1946f83d53a0, as the root user. This is a fully-fledged Ubuntu host, and we can do anything we like in it. Let’s start by asking for its hostname.



### **Installing a package in container**



You can keep playing with the container for as long as you like. When you’re done, type exit, and you’ll return to the command prompt of your Ubuntu host.



It has now stopped running. The container only runs for as long as the command we specified, /bin/bash, is running. Once we exited the container, that command ended, and the container was stopped.

## **docker ps -a command**

The container still exists; we can show a list of current containers using the docker ps -a command.

ubuntu@ip-172-31-34-103:~$ docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

1946f83d53a0 ubuntu "/bin/bash" 14 hours ago Exited (0) 13 hours ago tender\_visvesvaraya

By default, when we run just docker ps, we will only see the running containers. When we specify the -a flag, the docker ps -a command will show us all containers, both stopped and running.

We learn quite a bit of information about our container: its ID, the image used to create it, the command it last ran, when it was created, and its exit status (in our case, 0, because it was exited normally using the exit command). We can also see that each container has a name.

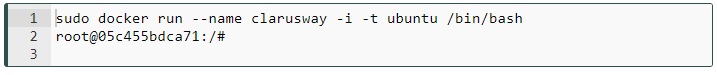
**💡Tips:**

There are three ways containers can be identified:

* a short UUID (1946f83d53a0),
* a longer UUID (like 1946f83d53a021548190053c4053f93fda09c3e5fa6bd670acbc7697a96b9961), and
* a name (tender\_visvesvaraya).

### **Container naming**

Docker will automatically generate a name at random for each container we create. We see that the container we’ve created is called tender\_visvesvaraya. If we want to specify a particular container name in place of the automatically generated name, we can do so using the --name flag.



You can check it in another terminal and verify the following output.

ubuntu@ip-172-31-34-103:~$ sudo docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

2f5a96f3ccde ubuntu "/bin/bash" 5 hours ago Exited (0) 5 hours ago clarusway

ubuntu@ip-172-31-34-103:~$

This would create a new container called clarusway. A valid container name can contain the following characters: a to z, A to Z, the digits 0 to 9, the underscore, period, and dash (or, expressed as a regular expression: [a-zA-Z0-9\_.-]).

We can use the container name in place of the container ID in most Docker commands. It is also much easier to remember a specific container name than a container ID or even a random name.

Names are unique. If we try to create two containers with the same name, the command will fail. We need to delete the previous container with the same name before we can create a new one. We can do so with the docker rm command.

### **Starting a stopped container**

ubuntu@ip-172-31-34-103:~$ sudo docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

2f5a96f3ccde ubuntu "/bin/bash" 5 hours ago Exited (0) 5 hours ago clarusway

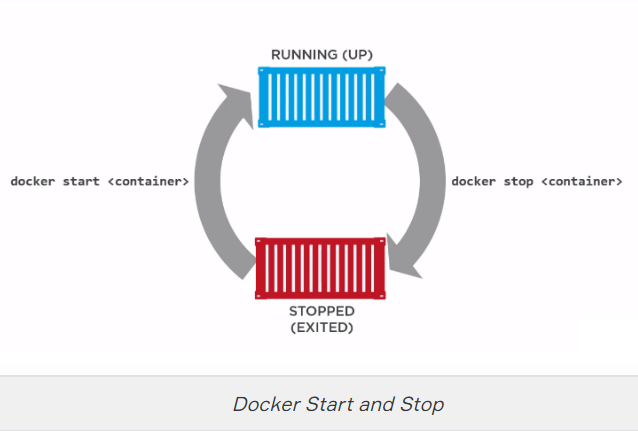
ubuntu@ip-172-31-34-103:~$

As we see above, we stopped clarusway container. If we want, we can restart a stopped container like so:



We could also refer to the container by its container ID instead.





If we run the docker ps command without the -a flag, we’ll see our running container.

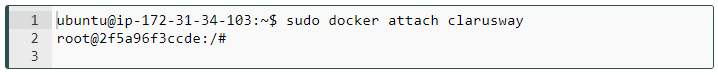
ubuntu@ip-172-31-34-103:~$ sudo docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

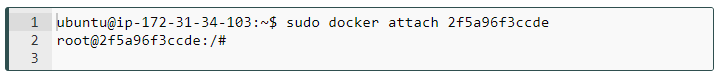
2f5a96f3ccde ubuntu "/bin/bash" 21 hours ago Up 29 seconds clarusway

### **Attaching to a container**

Our container will restart with the same options we’d specified when we launched it with the docker run command. So there is an interactive session waiting on our running container. We can reattach to that session using the docker attach command. So, we’ll be brought back to our container’s Bash prompt.



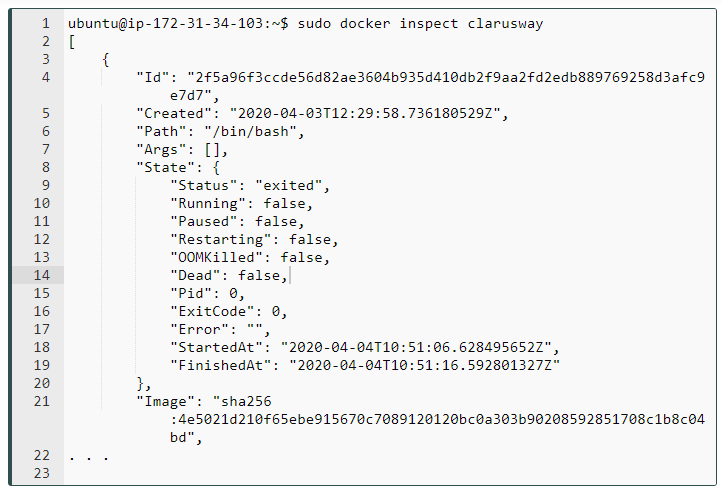
We could also do it via ID.



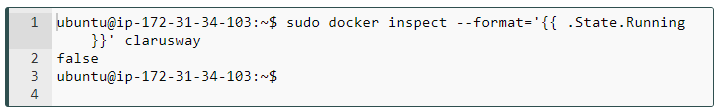
**Note:** Instead of typing a lengthy characters of container id, a few character that is enough to differentiate it from the other containers can be used. For example sudo docker attach 2f5 is the same with sudo docker attach 2f5a96f3ccde.

### **Finding out more about our container**

In addition to the information we retrieved about our container using the docker ps command, we can get a whole lot more information using the docker inspect command.



The docker inspect command will interrogate our container and return its configuration information, including names, commands, networking configuration, and a wide variety of other useful data. We can also selectively query the inspect results hash using the -f or --format flag.



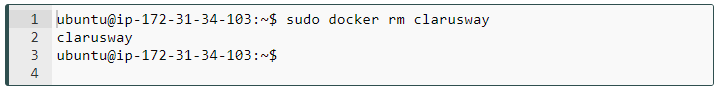
This will return the running state of the container, which in our case is false.

**💡Tips:**

* In addition to inspecting containers, you can see a bit more about how Docker works by exploring the /var/lib/docker directory. This directory holds your images, containers, and container configuration. You’ll find all your containers in the /var/lib/docker/containers directory.

### **Deleting a container**

If we are finished with a container, we can delete it using the docker rm command.



## **Docker Volumes**

### **Manage data in Docker**

By default, all files created inside a container are stored on a writable container layer. This means that the data doesn’t persist when that container no longer exists.

Docker volumes, which are special directories in a container, store files in the host machine so that the files are persisted even after the container stops.

Volumes are created and managed by Docker. You can create a volume explicitly using the docker volume create command.

[ec2-user@clarusway ~]$ docker volume create firstvolume

firstvolume

When you create a volume, it is stored within a directory on the Docker host. When you mount the volume into a container, this directory is what is mounted into the container. Look at the Mountpoint.

[ec2-user@clarusway ~]$ docker volume inspect firstvolume

[

{

"CreatedAt": "2020-07-12T13:19:27Z",

"Driver": "local",

"Labels": {},

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

"Name": "firstvolume",

"Options": {},

"Scope": "local"

}

]

### **Declaration of volumes**

Volumes can be declared on the command-line, with the --volume or -v flag for docker run. Let's create an alpine container.

[ec2-user@clarusway ~]$ docker container run -it -v firstvolume:/sample alpine sh

Unable to find image 'alpine:latest' locally

df20fa9351a1: Pull complete Digest: sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f60a771a2dfe321 32.77kB/2.798MB

Status: Downloaded newer image for alpine:latest

/ #

**Tip:**

-v or --volume: Consists of three fields, separated by colon characters (:). The fields must be in the correct order.

* the first field is the name of the volume, and is unique on a given host machine. In this example volume name is firstvolume.
* The second field is the path where the file or directory are mounted in the container. In this example folder in container is /sample.
* The third field is optional, and is a comma-separated list of options, such as ro (read only).

**Alpine:**

* Alpine Linux is an independent, non-commercial, general purpose Linux distribution designed for power users who appreciate security, simplicity and resource efficiency.
* Because of its small size, it is commonly used in containers providing quick boot-up times.

When we type ls command in alpine terminal, we can see the sample folder.

[ec2-user@clarusway ~]$ docker container run -it -v firstvolume:/sample alpine sh

Unable to find image 'alpine:latest' locally

df20fa9351a1: Pull complete Digest: sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f60a771a2dfe321 32.77kB/2.798MB

Status: Downloaded newer image for alpine:latest

/ #ls

bin etc lib mnt proc run sbin sys usr

dev home media opt root sample srv tmp var

/ #

We create a file in the sample folder and exit.

/ #ls

bin dev etc home lib media mnt opt proc root run sample sbin srv sys tmp usr var

/ # cd sample

/sample # touch file1.txt

/sample # echo "this is added in first container" >> file1.txt

/sample # exit

We remove the alpine container.

[ec2-user@clarusway ~]$ docker container ls -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

2e77f7472339 alpine "sh" 23 seconds ago Exited (0) 17 seconds ago intelligent\_ellis

[ec2-user@clarusway ~]$ docker container rm intelligent\_ellis

intelligent\_ellis

Let's check the file1.txt.

[ec2-user@clarusway ~]$ docker volume inspect firstvolume

[

{

"CreatedAt": "2020-07-12T13:36:52Z",

"Driver": "local",

"Labels": {},

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

"Name": "firstvolume",

"Options": {},

"Scope": "local"

}

]

[ec2-user@clarusway ~]$ sudo su

[root@clarusway]# cd /var/lib/docker/volumes/firstvolume/\_data

[root@clarusway \_data]# cat file1.txt

this is added in first container

As we see above, file1.txt is still there even if we remove the container.

### **Usage volume with different containers**

Let's run an alpine image and this time we will create try1 folder instead of sample folder.

[ec2-user@clarusway ~]$ docker container run -it -v firstvolume:/try1 alpine sh

Unable to find image 'alpine:latest' locally

latest: Pulling from library/alpine

df20fa9351a1: Pull complete Digest: sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f60a771a2dfe321

Status: Downloaded newer image for alpine:latest

/ # ls

bin dev etc home lib media mnt opt proc root run sbin srv sys tmp try1 usr var

/ # cd try1

/try1 # ls

file1.txt

/try1 # cat file1.txt

this is added in first container

As we see, we can reach file1.txt via a new container.

We can add a new file to the try1 folder.

/try1 # touch file2.txt

/try1 # echo "this is added in second container" >> file2.txt

/try1 # cat file2.txt

this is added in second container

/try1 #

We create an ubuntu image.

[ec2-user@clarusway ~]$ docker container run -it -v firstvolume:/try2 ubuntu sh

Unable to find image 'ubuntu:latest' locally

latest: Pulling from library/ubuntu

692c352adcf2: Pull complete

97058a342707: Pull complete

2821b8e766f4: Pull complete

4e643cc37772: Pull complete

Digest: sha256:55cd38b70425947db71112eb5dddfa3aa3e3ce307754a3df2269069d2278ce47

Status: Downloaded newer image for ubuntu:latest

# ls

bin boot dev etc home lib lib32 lib64 libx32 media mnt opt proc root run sbin srv sys tmp try2 usr var

# cd try2

# ls

file1.txt file2.txt

We can use the same volumes with different containers.

### **Use a read-only volume**

For some development applications, the container needs to write into the bind mount so that changes are propagated back to the Docker host. At other times, the container only needs read access to the data. Remember that multiple containers can mount the same volume, and it can be mounted read-write for some of them and read-only for others, at the same time.

[ec2-user@clarusway ~]$ docker container run -it -v firstvolume:/try3:ro centos sh

Unable to find image 'centos:latest' locally

latest: Pulling from library/centos

6910e5a164f7: Pull complete Digest: sha256:4062bbdd1bb0801b0aa38e0f83dece70fb7a5e9bce223423a68de2d8b784b43b

Status: Downloaded newer image for centos:latest

sh-4.4# ls

bin dev etc home lib lib64 lost+found media mnt opt proc root run sbin srv sys tmp try3 usr var

sh-4.4# cd try3

sh-4.4# ls

file1.txt file2.txt

Let's try to add a file to the volume.

sh-4.4# touch file3

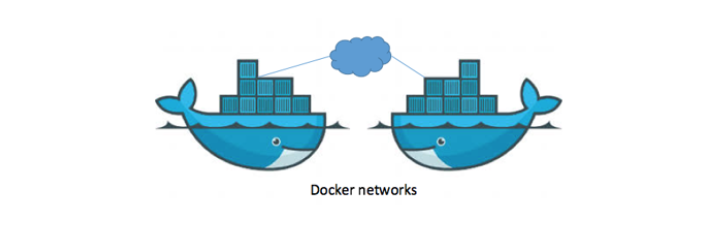
touch: cannot touch 'file3': Read-only file system

sh-4.4#

## **Docker Networking**

### **Networking overview**

One of the reasons [Docker containers](https://lms.clarusway.com/mod/lesson/view.php?id=2158) and services are so powerful is that you can connect them together, or connect them to non-Docker workloads. [Docker containers](https://lms.clarusway.com/mod/lesson/view.php?id=2158) and services do not even need to be aware that they are deployed on Docker, or whether their peers are also Docker workloads or not. Whether your Docker hosts run Linux, Windows, or a mix of the two, you can use Docker to manage them in a platform-agnostic way.



Network drivers

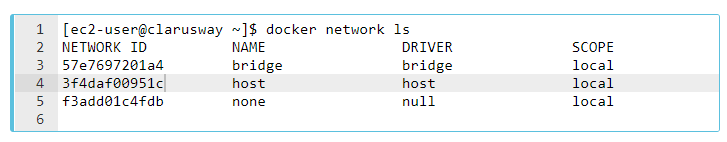
Docker’s networking subsystem is pluggable, using drivers. Several drivers exist by default, and provide core networking functionality:

* **bridge:** The default network driver. If you don’t specify a driver, this is the type of network you are creating. Bridge networks are usually used when your applications run in standalone containers that need to communicate.
* **host:** For standalone containers, remove network isolation between the container and the Docker host, and use the host’s networking directly.
* **none:** For this container, disable all networking. Usually used in conjunction with a custom network driver.
* **overlay:** Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons. This strategy removes the need to do OS-level routing between these containers.
* **macvlan:** Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host’s network stack.
* **Network plugins:** You can install and use third-party network plugins with Docker. These plugins are available from Docker Hub or from third-party vendors.

### **Use the default bridge network**

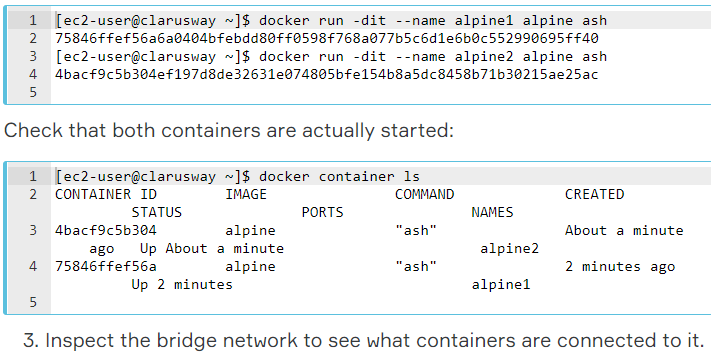
In this example, you start two different alpine containers on the same Docker host and do some tests to understand how they communicate with each other. You need to have Docker installed and running.

1. Open a terminal window. List current networks before you do anything else. Here’s what you should see if you’ve never added a network.



The default bridge network is listed, along with host and none. The latter two are not fully-fledged networks but are used to start a container connected directly to the Docker daemon host’s networking stack or to start a container with no network devices.

1. Start two alpine containers running ash, which is Alpine’s default shell rather than bash. The -dit flags mean to start the container detached (in the background), interactive (with the ability to type into it), and with a TTY (so you can see the input and output). Since you are starting it detached, you won’t be connected to the container right away. Instead, the container’s ID will be printed. Because you have not specified any --network flags, the containers connect to the default bridge network.



[ec2-user@clarusway ~]$ docker network inspect bridge

[

{

"Name": "bridge",

"Id": "57e7697201a40c4d6acb28f6ecaafac18d2ca282fd5eb3d8ced1e0ea5fc7b199",

"Created": "2020-07-13T10:10:39.886577521Z",

"Scope": "local",

"Driver": "bridge",

"EnableIPv6": false,

"IPAM": {

"Driver": "default",

"Options": null,

"Config": [

{

"Subnet": "172.17.0.0/16",

"Gateway": "172.17.0.1"

}

]

},

"Internal": false,

"Attachable": false,

"Ingress": false,

"ConfigFrom": {

"Network": ""

},

"ConfigOnly": false,

"Containers": {

"4bacf9c5b304ef197d8de32631e074805bfe154b8a5dc8458b71b30215ae25ac": {

"Name": "alpine2",

"EndpointID": "9bc6784ddf823bd7e7c6ac37ceddf87cfea63292f23ca3d5b178a367474a1299",

"MacAddress": "02:42:ac:11:00:03",

"IPv4Address": "172.17.0.3/16",

"IPv6Address": ""

},

"75846ffef56a6a0404bfebdd80ff0598f768a077b5c6d1e6b0c552990695ff40": {

"Name": "alpine1",

"EndpointID": "8426e9df26bf373f00dd185e23683e9c41ea45aee3aefed68ec1b28a397306fb",

"MacAddress": "02:42:ac:11:00:02",

"IPv4Address": "172.17.0.2/16",

"IPv6Address": ""

}

},

"Options": {

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

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

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

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

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

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

},

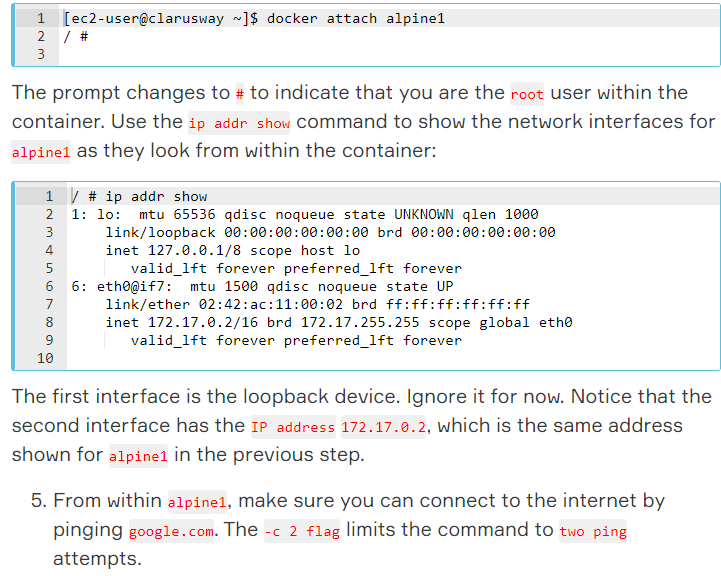
"Labels": {}

}

]

Near the top, information about the bridge network is listed, including the IP address of the gateway between the Docker host and the bridge network (172.17.0.1). Under the Containers key, each connected container is listed, along with information about its IP address (172.17.0.2 for alpine1 and 172.17.0.3 for alpine2).

1. The containers are running in the background. Use the docker attach command to connect to alpine1.

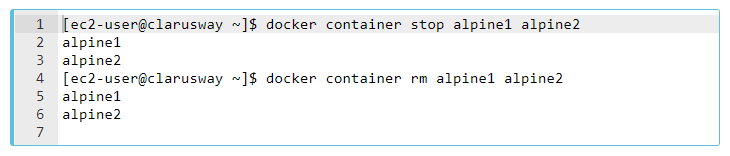


The first interface is the loopback device. Ignore it for now. Notice that the second interface has the IP address 172.17.0.2, which is the same address shown for alpine1 in the previous step.

1. From within alpine1, make sure you can connect to the internet by pinging google.com. The -c 2 flag limits the command to two ping attempts.



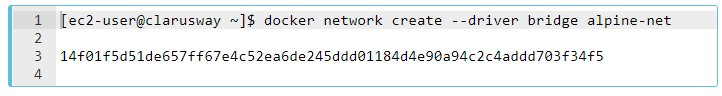
1. Detach from alpine1 without stopping it by using the detach sequence, CTRL + p + q (hold down CTRL and type p followed by q). If you wish, attach to alpine2 and repeat steps 4, 5, and 6 there, substituting alpine1 for alpine2.
2. Stop and remove both containers.



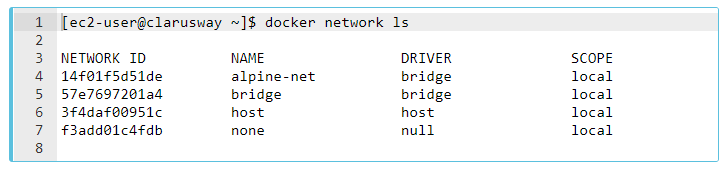
### **Use user-defined bridge networks**

In this example, we again start two alpine containers but attach them to a user-defined network called alpine-net which we have already created. These containers are not connected to the default bridge network at all. We then start a third alpine container which is connected to the bridge network but not connected to alpine-net and a fourth alpine container which is connected to both networks.

1. Create the alpine-net network. You do not need the --driver bridge flag since it’s the default, but this example shows how to specify it.



1. List Docker’s networks:

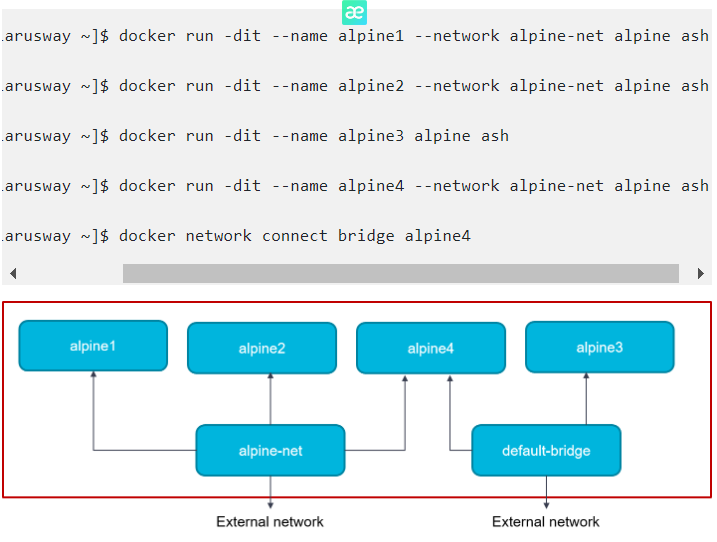


Inspect the alpine-net network. This shows you its IP address and the fact that no containers are connected to it:

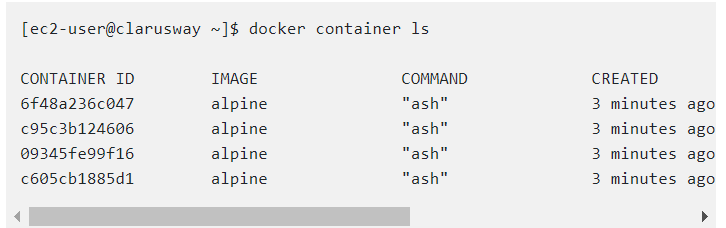


Notice that this network’s gateway is 172.18.0.1, as opposed to the default bridge network, whose gateway is 172.17.0.1. The exact IP address may be different on your system.

1. Create your four containers. Notice the --network flags. You can only connect to one network during the docker run command, so you need to use docker network connect afterward to connect alpine4 to the bridge network as well.



Verify that all containers are running:



1. Inspect the bridge network and the alpine-net network again:

[ec2-user@clarusway ~]$ docker network inspect alpine-net

[

{

"Name": "alpine-net",

"Id": "14f01f5d51de657ff67e4c52ea6de245ddd01184d4e90a94c2c4addd703f34f5",

"Created": "2020-07-13T10:54:23.575226558Z",

"Scope": "local",

"Driver": "bridge",

"EnableIPv6": false,

"IPAM": {

"Driver": "default",

"Options": {},

"Config": [

{

"Subnet": "172.18.0.0/16",

"Gateway": "172.18.0.1"

}

]

},

"Internal": false,

"Attachable": false,

"Ingress": false,

"ConfigFrom": {

"Network": ""

},

"ConfigOnly": false,

"Containers": {

"0891d7a4a8e3650ccb1ea169cd3e250226c734184f30757a60bd0513e2953924": {

"Name": "alpine4",

"EndpointID": "c691b71fb8ff6c0e1b6405feab28031acdaff5d764f0f0cd2901605e1c26a718",

"MacAddress": "02:42:ac:12:00:04",

"IPv4Address": "172.18.0.4/16",

"IPv6Address": ""

},

"ba2425af655ab6af6f74e90dcf51bc8639c663eb978b991d43940a45116ecffd": {

"Name": "alpine2",

"EndpointID": "6a6d0d3014a5e8192316bda4fb5bc19f63adb11f4de3acbc5fc30300efd88e54",

"MacAddress": "02:42:ac:12:00:03",

"IPv4Address": "172.18.0.3/16",

"IPv6Address": ""

},

"f1b109ada2ac11d005f9519846815d2ecb15b57b606d5e1aba418eb4b8f360d8": {

"Name": "alpine1",

"EndpointID": "7dab942e3fc98084552d9e80716eedd036808cdeda71c1068ec1a5b67c373e9d",

"MacAddress": "02:42:ac:12:00:02",

"IPv4Address": "172.18.0.2/16",

"IPv6Address": ""

}

},

"Options": {},

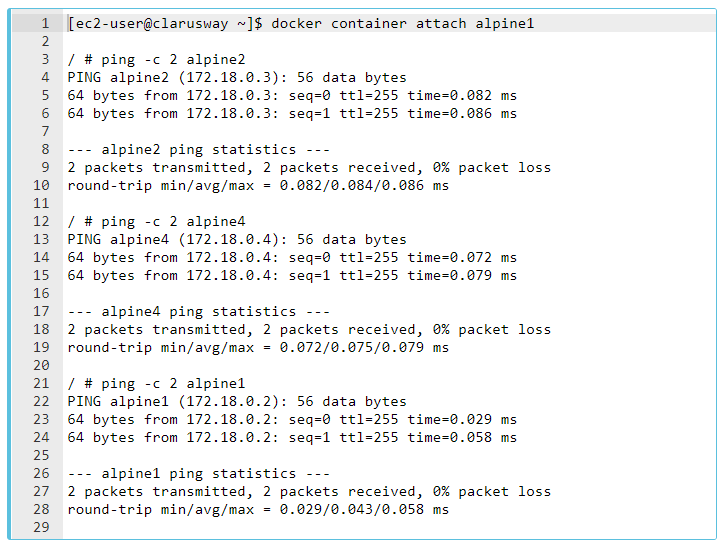
"Labels": {}

}

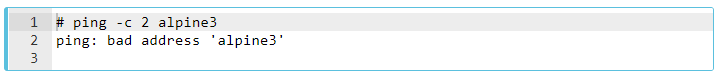
]

Containers alpine1, alpine2, and alpine4 are connected to the alpine-net network.

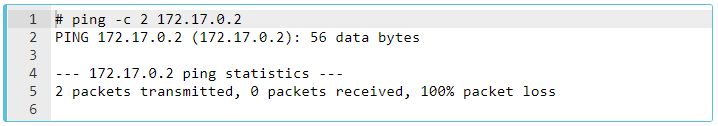
1. On user-defined networks like alpine-net, containers can not only communicate by IP address but can also resolve a container name to an IP address. This capability is called automatic service discovery. Let’s connect to alpine1 and test this out. alpine1 should be able to resolve alpine2 and alpine4 (and alpine1, itself) to IP addresses.



1. From alpine1, you should not be able to connect to alpine3 at all, since it is not on the alpine-net network.

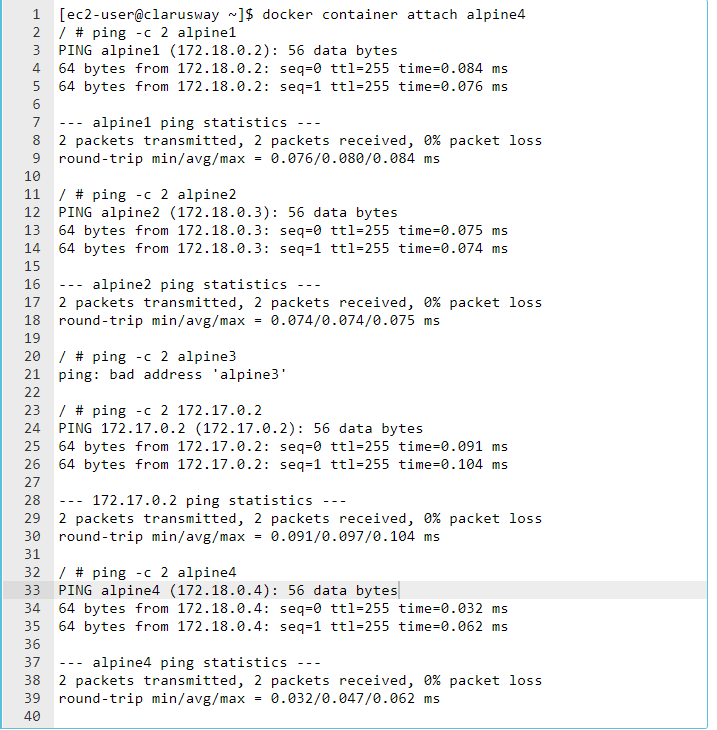


Not only that, but you can’t connect to alpine3 from alpine1 by its IP address either. Look back at the docker network inspect output for the bridge network and find alpine3’s IP address: 172.17.0.2 Try to ping it.



Detach from alpine1 using detach sequence, CTRL + p + q (hold down CTRL and type p followed by q).

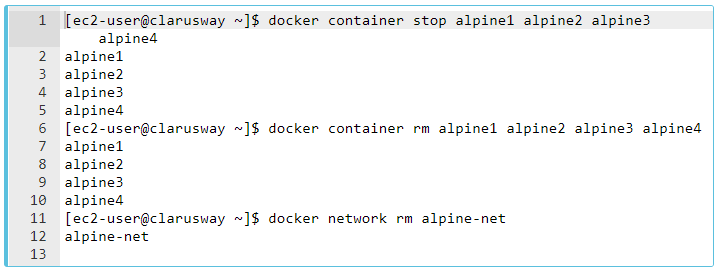
1. Remember that alpine4 is connected to both the default bridge network and alpine-net. It should be able to reach all of the other containers. However, you will need to address alpine3 by its IP address. Attach to it and run the tests.



8. As a final test, make sure your containers can all connect to the internet by pinging clarusway.com. You are already attached to alpine4 so start by trying from there. Next, detach from alpine4 and connect to alpine3 (which is only attached to the bridge network) and try again. Finally, connect to alpine1 (which is only connected to the alpine-net network) and try again.



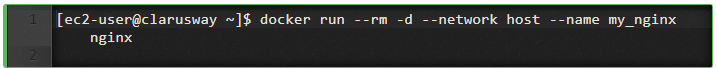
9. Stop and remove all containers and the alpine-net network.



### **Using the host network**

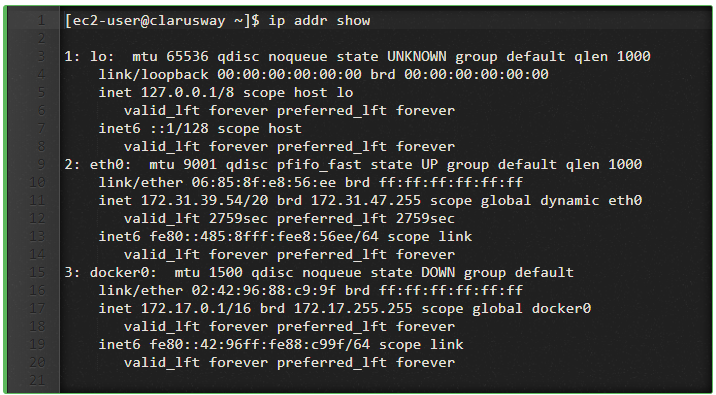
Let's start a nginx container which binds directly to port 80 on the Docker host. From a networking point of view, this is the same level of isolation as if the nginx process were running directly on the Docker host and not in a container. However, in all other ways, such as storage, process namespace, and user namespace, the nginx process is isolated from the host.

1. Create and start the container as a detached process. The --rm option means to remove the container once it exits/stops. The -d flag means to start the container detached (in the background)

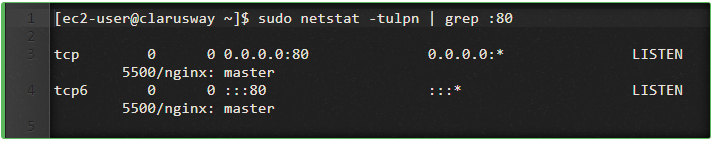


1. Access Nginx by browsing to http://localhost:80/. (< ip number of ec2 instance >:80/)
2. Examine your network stack using the following commands:

* Examine all network interfaces and verify that a new one was not created.



Verify which process is bound to port 80, using the netstat command. You need to use sudo because the process is owned by the Docker daemon user and you otherwise won’t be able to see its name or PID.



4. Stop the container. It will be removed automatically as it was started using the --rm option.

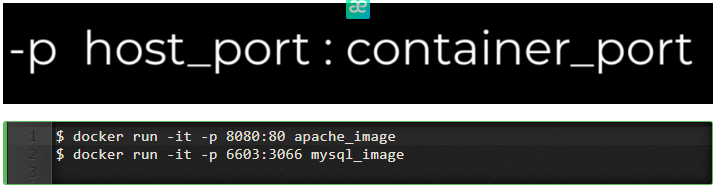


### **Container networking**

The type of network a container uses, whether it is a bridge, an overlay, a macvlan network, or a custom network plugin, is transparent from within the container. From the container’s point of view, it has a network interface with an IP address, a gateway, a routing table, DNS services, and other networking details (assuming the container is not using the none network driver). This lesson is about networking concerns from the point of view of the container.

### **Published ports**

By default, when you create a container, it does not publish any of its ports to the outside world. To make a port available to services outside of Docker, or to [Docker containers](https://lms.clarusway.com/mod/lesson/view.php?id=2158) which are not connected to the container’s network, use the --publish or -p flag. This creates a firewall rule which maps a container port to a port on the Docker host.



Here are some examples.

| **Flag value** | **Description** |
| --- | --- |
| -p 8080:80 | Map TCP port 80 in the container to port 8080 on the Docker host. |
| -p 192.168.1.100:8080:80 | Map TCP port 80 in the container to port 8080 on the Docker host for connections to host IP 192.168.1.100. |
| -p 8080:80/udp | Map UDP port 80 in the container to port 8080 on the Docker host. |
| -p 8080:80/tcp -p 8080:80/udp | Map TCP port 80 in the container to TCP port 8080 on the Docker host, and map UDP port 80 in the container to UDP port 8080 on the Docker host. |

### **IP address and hostname**

By default, the container is assigned an IP address for every Docker network it connects to. The IP address is assigned from the pool assigned to the network, so the Docker daemon effectively acts as a DHCP server for each container. Each network also has a default subnet mask and gateway.

When the container starts, it can only be connected to a single network, using --network. However, you can connect a running container to multiple networks using docker network connect. When you start a container using the --network flag, you can specify the IP address assigned to the container on that network using the --ip or --ip6 flags.

When you connect an existing container to a different network using docker network connect, you can use the --ip or --ip6 flags on that command to specify the container’s IP address on the additional network.

In the same way, a container’s hostname defaults to be the container’s ID in Docker. You can override the hostname using--hostname. When connecting to an existing network using docker network connect, you can use the --alias flag to specify an additional network alias for the container on that network.

### **DNS services**

By default, a container inherits the DNS settings of the host, as defined in the /etc/resolv.conf configuration file. Containers that use the default bridge network get a copy of this file, whereas containers that use a custom network use Docker’s embedded DNS server, which forwards external DNS lookups to the DNS servers configured on the host.

Custom hosts defined in /etc/hosts are not inherited. To pass additional hosts into your container, refer to add entries to container hosts file in the docker run reference documentation. You can override these settings on a per-container basis. settings on a per-container basis.

| **Flag value** | **Description** |
| --- | --- |
| --dns | The IP address of a DNS server. To specify multiple DNS servers, use multiple --dns flags. If the container cannot reach any of the IP addresses you specify, Google’s public DNS server 8.8.8.8 is added, so that your container can resolve internet domains. |
| --dns-search | A DNS search domain to search non-fully-qualified hostnames. To specify multiple DNS search prefixes, use multiple --dns-search flags. |
| --dns-opt | A key-value pair representing a DNS option and its value. See your operating system’s documentation for resolv.conf for valid options. |
| --hostname | The hostname a container uses for itself. Defaults to the container’s ID if not specified. |

## **Docker Images**

### **What is a Docker image?**

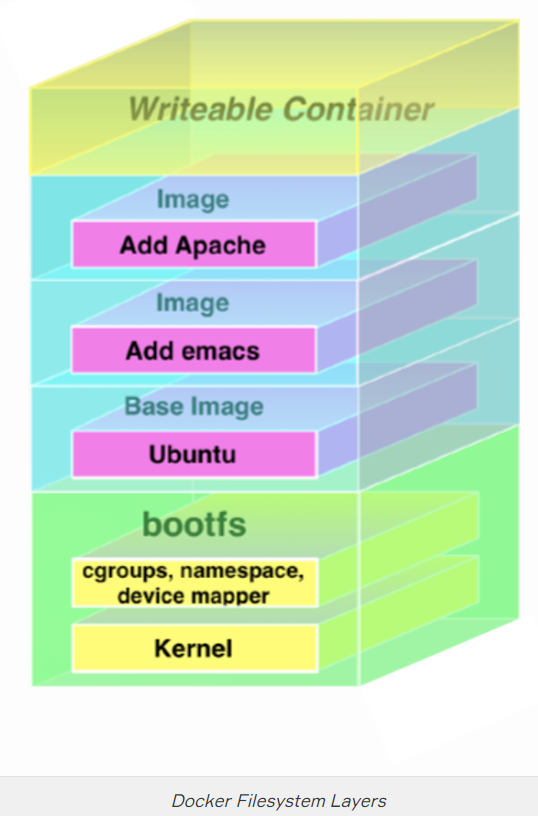
A Docker image is made up of filesystems layered over each other. At the base is a boot filesystem, bootfs, which resembles the typical Linux/Unix boot filesystem. A Docker user will probably never interact with the boot filesystem. Indeed, when a container has booted, it is moved into memory, and the boot filesystem is unmounted to free up the RAM used by the initrd disk image.

So far this looks pretty much like a typical Linux virtualization stack. Indeed, Docker next layers a root filesystem, rootfs, on top of the boot filesystem. This rootfs can be one or more operating systems (e.g., a Debian or Ubuntu filesystem).

In a more traditional Linux boot, the root filesystem is mounted read-only and then switched to read-write after boot and an integrity check is conducted. In the Docker world, however, the root filesystem stays in read-only mode, and Docker takes advantage of a union mount to add more read-only filesystems onto the root filesystem. A union mount is a mount that allows several filesystems to be mounted at one time but appears to be one filesystem. The union mount overlays the filesystems on top of one another so that the resulting filesystem may contain files and subdirectories from any or all of the underlying filesystems.

Docker calls each of these filesystems images. Images can be layered on top of one another. The image below is called the parent image and you can traverse each layer until you reach the bottom of the image stack where the final image is called the base image. Finally, when a container is launched from an image, Docker mounts a read-write filesystem on top of any layers below. This is where whatever processes we want our Docker container to run will execute.

This sounds confusing, let's represent it by a diagram.

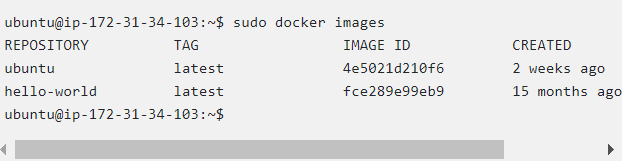


When Docker first starts a container, the initial read-write layer is empty. As changes occur, they are applied to this layer; for example, if you want to change a file, then that file will be copied from the read-only layer below into the readwrite layer. The read-only version of the file will still exist but is now hidden underneath the copy.

This pattern is traditionally called ”copy on write” and is one of the features that makes Docker so powerful. Each read-only image layer is read-only; this image never changes. When a container is created, Docker builds from the stack of images and then adds the read-write layer on top. That layer, combined with the knowledge of the image layers below it and some configuration data, form the container. As we learned, containers can be changed, they have state, and they can be started and stopped. This, and the image-layering framework, allows us to quickly build images and run containers with our applications and services.

### **Listing Docker images**

We can list the images using the docker images command.



We see that we’ve got images, from a repository called ubuntu and hello-world. When we ran the docker run command, that part of the process was downloading these images.

**Tips:**

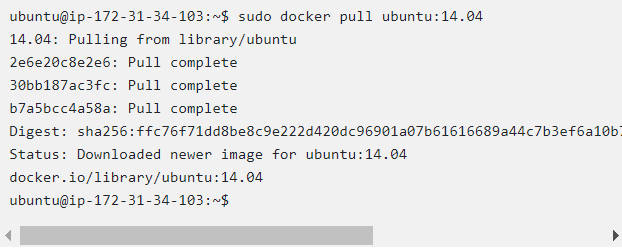
* Local images live on our local Docker host in the /var/lib/docker directory. Each image will be inside a directory named for your storage driver; for example, aufs or devicemapper. You’ll also find all your containers in the /var/lib/docker/containers directory.

Images were downloaded from a repository. Images live inside repositories, and repositories live on registries. The default registry is the public registry managed by Docker, Inc., Docker Hub.

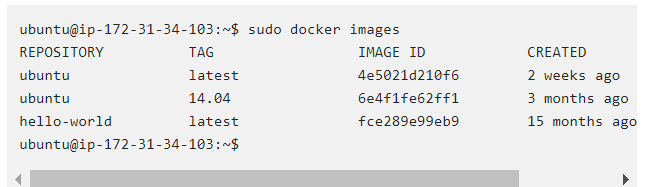
**Tips:**

* The Docker registry code is open source. You can also run your own registry, as we’ll see later in this chapter. The Docker Hub product is also available as a commercial ”behind the firewall” product called Docker Trusted Registry, formerly Docker Enterprise Hub.

Inside [Docker Hub](https://hub.docker.com/) (or on a Docker registry you run yourself), images are stored in repositories. You can think of an image repository as being much like a Git repository. It contains images, layers, and metadata about those images. Each repository can contain multiple images (e.g., the ubuntu repository contains images for Ubuntu 18.04, 19.10, 20.04, 14.04, and 16.04). Let’s get another image from the ubuntu repository now.



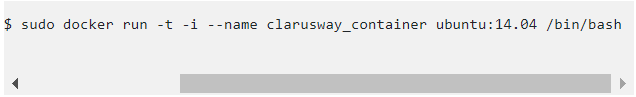
Here we’ve used the docker pull command to pull down the Ubuntu 14.04 image from the ubuntu repository. Let’s see what our docker images command reveals now.



We can see we’ve now got the latest Ubuntu image and the 14.04 image. This shows us that the ubuntu image is actually a series of images collected under a single repository.

**💡Tips:**

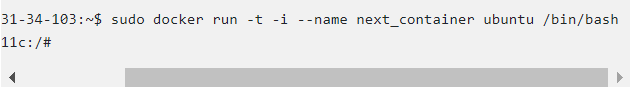
* We call it the Ubuntu operating system, but really it is not the full operating system. It’s a cut-down version with the bare runtime required to run the distribution.
* We identify each image inside that repository by what Docker calls tags. Each image is being listed by the tags applied to it, so, for example, 18.04, 19.10, 14.04, or xenial and so on. Each tag marks together with a series of image layers that represent a specific image (e.g., the 14.04 tag collects together all the layers of the Ubuntu 14.04 image). This allows us to store more than one image inside a repository.
* We can refer to a specific image inside a repository by suffixing the repository name with a colon and a tag name, for example:



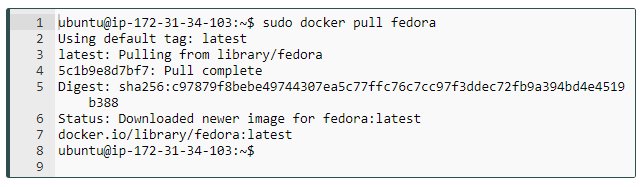
This launches a container from the ubuntu:14.04 image, which is an Ubuntu 14.04 operating system. It’s always a good idea to build a container from specific tags. That way we’ll know exactly what the source of our container is. There are differences, for example, between Ubuntu 14.04 and 18.04, so it would be useful to specifically state that we’re using ubuntu:14.04 so we know exactly what we’re getting.

### **Pulling images**

When we run a container from images with the docker run command if the image isn’t present locally already then Docker will download it from the Docker Hub. By default, if you don’t specify a specific tag, Docker will download the latest tag, for example: this will download the ubuntu:latest image if it isn’t already present on the host.

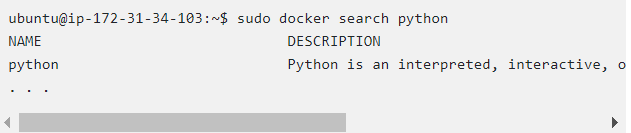


Alternatively, we can use the docker pull command to pull images down ourselves preemptively. Using docker pull saves us some time launching a container from a new image. Let’s see that now by pulling down the fedora base image.



### **Searching for images**

We can search all of the publicly available images on Docker Hub using the docker search command:

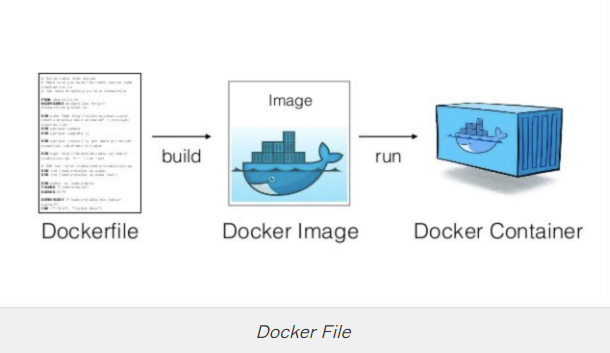


Here, we’ve searched the Docker Hub for the term python. It’ll search for images and return:

* Repository names
* Image descriptions
* Stars - these measure the popularity of an image
* Official - an image managed by the upstream developer (e.g., the fedora image managed by the Fedora team)
* Automated - an image built by the Docker Hub’s Automated Build process

### **Dockerfile**

💡  *A Dockerfile*  *is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build, users can create an automated build that executes several command-line instructions in succession. Docker can build images automatically by reading the instructions from a****Dockerfile****.*



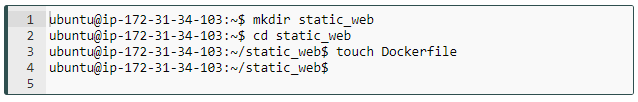
Some instructions in the Dockerfile described below:

* **FROM**: The FROM instruction initializes a new build stage and sets the Base Image for subsequent instructions. As such, a valid Dockerfile must start with a FROM instruction.
* **RUN**: The RUN instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile.
* **CMD**: The main purpose of a CMD is to provide defaults for an executing container.
* **ADD**: The ADD instruction copies new files, directories or remote file URLs from <src> and adds them to the filesystem of the image at the path <dest>.
* **COPY**: The COPY instruction copies new files or directories from <src> and adds them to the filesystem of the container at the path <dest>.
* **WORKDIR**: The WORKDIR instruction sets the working directory for any RUN, CMD, COPY and ADD instructions that follow it in the Dockerfile. If the WORKDIR doesn’t exist, it will be created even if it’s not used in any subsequent Dockerfile instruction.

**Note:**  
When ADD’ing files Docker uses the ending character of the destination to determine what the source is. If the destination ends in a /, then it considers the source a directory. If it doesn’t end in a /, it considers the source a file.  
  
COPY only supports the basic copying of local files into the container, while ADD has some features (like local-only tar extraction and remote URL support) that are not immediately obvious. Consequently, the best use for ADD is local tar file auto-extraction into the image, as in ADD rootfs.tar.xz /.

## **Our first Dockerfile**

Let’s create a directory and an initial Dockerfile. We’re going to build a Docker image that contains a simple web server.



We’ve created a directory called static\_web to hold our Dockerfile. This directory is our build environment, which is what Docker calls a context or build context. Docker will upload the build context, as well as any files and directories contained in it, to our Docker daemon when the build is run. This provides the Docker daemon with direct access to any code, files or other data you might want to include in the image.

We’ve also created an empty Dockerfile file to get started. Now let’s look at an example of a Dockerfile to create a Docker image that will act as a Web server.



The Dockerfile contains a series of instructions paired with arguments. Each instruction, for example FROM, should be in upper-case and be followed by an argument: FROM ubuntu:18.04. Instructions in the Dockerfile are processed from the top down, so you should order them accordingly.

Each instruction adds a new layer to the image and then commits the image. Docker executing instructions roughly follow a workflow:

* Docker runs a container from the image.
* An instruction executes and makes a change to the container.
* Docker runs the equivalent of docker commit to commit a new layer.
* Docker then runs a new container from this new image.
* The next instruction in the file is executed, and the process repeats until all instructions have been executed.

This means that if your Dockerfile stops for some reason (for example, if an instruction fails to complete), you will be left with an image you can use. This is highly useful for debugging: you can run a container from this image interactively and then debug why your instruction failed using the last image created.

## **Dockerfile Format**

The first instruction in a Dockerfile must be FROM. The FROM instruction specifies an existing image that the following instructions will operate on; this image is called the base image.

In our sample Dockerfile we’ve specified the ubuntu:18.04 image as our base image. This specification will build an image on top of an Ubuntu 18.04 base operating system. As with running a container, you should always be specific about exactly from which base image you are building.

Next, we’ve specified the MAINTAINER instruction, which tells Docker who the author of the image is and what their email address is. This is useful for specifying an owner and contact for an image.

We’ve followed these instructions with two RUN instructions. The RUN instruction executes commands on the current image. The commands in our example: updating the installed APT repositories and installing the nginx package and then creating the /usr/share/nginx/html/index.html file containing some example text. As we’ve discovered, each of these instructions will create a new layer and, if successful, will commit that layer and then execute the next instruction.

By default, the RUN instruction executes inside a shell using the command wrapper /bin/sh -c. If you are running the instruction on a platform without a shell or you wish to execute without a shell (for example, to avoid shell string munging), you can specify the instruction in exec format:



We use this format to specify an array containing the command to be executed and then each parameter to pass to the command.

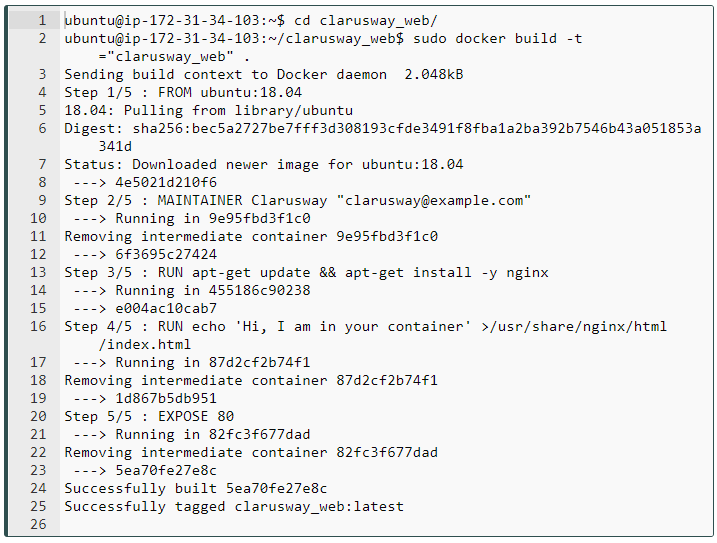
Next, we’ve specified the EXPOSE instruction, which tells Docker that the application in this container will use this specific port on the container. That doesn’t mean you can automatically access whatever service is running on that port (here, port 80) on the container. For security reasons, Docker doesn’t open the port automatically but waits for you to do it when you run the container using the docker run command.

Q: Explain Dockerfile  
A: Dockerfile contain a set of instructions that specify what environment to use and which commands to run. It enable you to create your own images. A Dockerfile describes the software that makes up an image.

 - Interview Q&A

### **Building our own images**

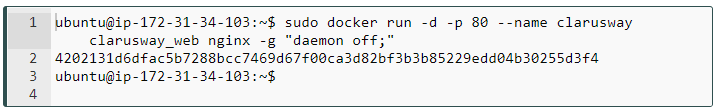
All of the instructions will be executed and committed and a new image returned when we run the docker build command. Let’s try that now:



We’ve used the docker build command to build our new image.

## **Launching a container from our new image**

Let’s launch a new container using our new image.



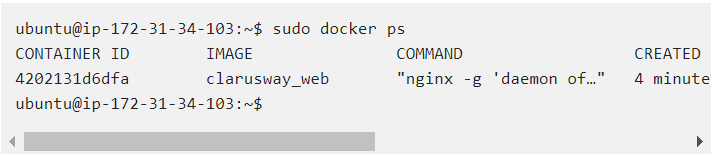
Here we’ve launched a new container called clarusway using the docker run command and the name of the image we’ve just created. We’ve specified the -d option, which tells Docker to *run detached* in the background. This allows us to run long-running processes like the Nginx daemon. We’ve also specified a command for the container to run: nginx -g "daemon off;". This will launch Nginx in the *foreground* to run our web server.

We’ve also specified a new flag, -p. The -p flag manages which network ports Docker publishes at runtime. When you run a container, Docker has two methods of assigning ports on the Docker host:

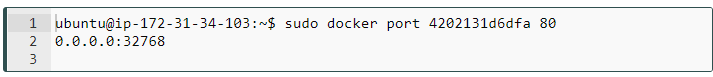
* Docker can randomly assign a high port from the range 32768 to 61000 on the Docker host that maps to port 80 on the container.
* You can specify a specific port on the Docker host that maps to port 80 on the container.

The docker run command will open a random port on the Docker host that will connect to port 80 on the Docker container.

Let’s look at what port has been assigned using the docker ps command.



We see that port 32768 is mapped to the container port of 80. We can get the same information with the docker port command.



### **Pushing images to the Docker Hub**

Once we’ve got an image, we can upload it to the Docker Hub. This allows us to make it available for others to use. For example, we could share it with others in our organization or make it publicly available.

### **Set up your Docker Hub account**

If you don’t have a Docker ID, follow these steps to create one. A Docker ID allows you to share images on Docker Hub.

1. Visit the [Docker Hub sign up](https://hub.docker.com/signup) page.
2. Fill out the form and submit to create your Docker ID.
3. Verify your email address to complete the registration process.
4. Click on the Docker icon in your toolbar or system tray, and click Sign in / Create Docker ID.
5. Fill in your new Docker ID and password. After you have successfully authenticated, your Docker ID appears in the Docker Desktop menu in place of the ‘Sign in’ option you just used.

You can also sign into Docker Hub from the command line by typing docker login.

### **Create a Docker Hub repository and push your image**

1. Click on the Docker icon in your menu bar, and navigate to **Repositories>Create.** You’ll be redirected to the **Create Repository** page on Docker Hub.
2. Type the repository name and click Create at the bottom of the page. Do not fill any other details for now.
3. You are now ready to share your image on Docker Hub, however, there’s one thing you must do first: images must be namespaced correctly to share on Docker Hub. Specifically, you must name images like /:.

**💡Tips:**

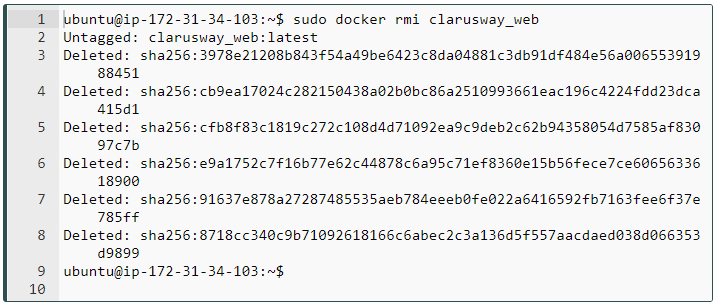
* The Docker Hub also has the option of private repositories. These are a paid-for feature that allows you to store an image in a private repository that is only available to you or anyone with whom you share it. This allows you to have private images containing proprietary information or code you might not want to share publicly.

We push images to the Docker Hub using the docker push command.



### **Deleting an image**

We can also delete images with docker rmi (docker image rm) command.

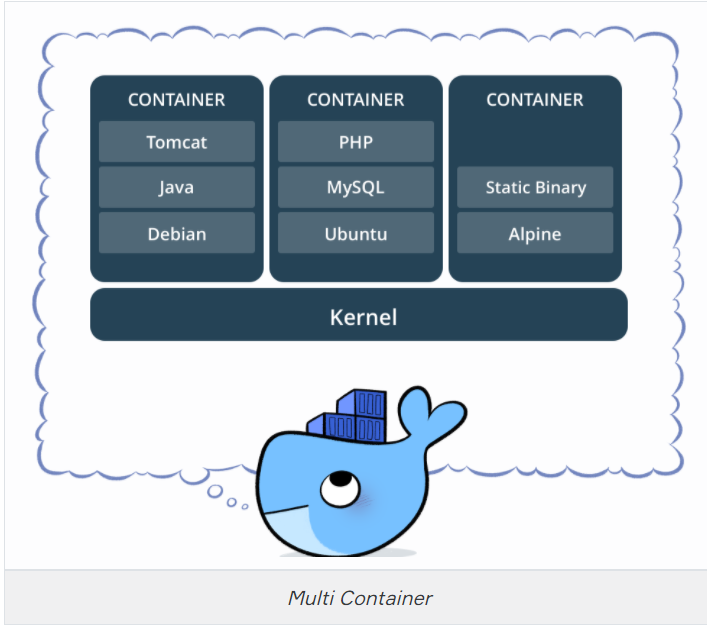


## **Running multi-containers**

### **Why multi-containers?**

In particular, we are going to see how we can run and manage multi-container docker environments.

*Why multi-container you might ask?*  *Well, one of the key points of Docker is the way it provides*isolation*. The idea of bundling a process with its dependencies in a sandbox (called containers) is what makes this so powerful.*



* Just like it's a good strategy to decouple your application tiers, it is wise to keep containers for each of the **services** separate. Each tier is likely to have different **resource needs** and those needs might grow at different rates.
* By separating the tiers into different containers, we can **compose each tier** using the most appropriate instance type based on different resource needs. This also plays in very well with the whole microservices (software applications as suites of independently deployable services) movement which is one of the main reasons why Docker (or any other container technology) is at the forefront of modern microservices architectures.

### **Docker Compose**

Docker Compose is a tool for defining and running multi-container Docker applications.

Compose is a tool that is used for defining and running multi-container Docker apps in an easy way. It provides a configuration file called docker-compose.yml that can be used to bring up an application and the **suite of services** it depends on with just one command. Compose works in all environments: production, staging, development, testing, as well as CI workflows, although Compose is ideal for development and testing environments.

Using Compose is basically a three-step process:

1. Define your app’s environment with a Dockerfile so it can be reproduced anywhere.
2. Define the services that make up your app in **docker-compose.yml** so they can be run together in an isolated environment.
3. Run docker-compose up and Compose starts and runs your entire app.

### **Common Use Cases**

Compose can be used in many different ways. Some common use cases are outlined below.

* **Development environments** When you’re developing software, the ability to run an application in an isolated environment and interact with it is crucial. The Compose command line tool can be used to create the environment and interact with it.

The Compose file provides a way to document and configure all of the application’s service dependencies (databases, queues, caches, web service APIs, etc). Using the Compose command line tool you can create and start one or more containers for each dependency with a single command (docker-compose up).

* **Automated testing environments**

An important part of any Continuous Deployment or Continuous Integration process is the **automated test suite**. Automated end-to-end testing requires an environment in which to run tests. Compose provides a convenient way to create and destroy isolated testing environments for your test suite. By defining the full environment in a Compose file, you can create and destroy these environments in just a few commands:

$ docker-compose up -d

$ ./run\_tests

$ docker-compose down

* **Single Host Deployments**

Compose has traditionally been focused on *development* and *testing* workflows. You can use Compose to deploy to a remote Docker Engine. The Docker Engine may be a single instance provisioned with **Docker Machine**

Q: What are the docker compose features?  
A: The features of Compose that make it effective are:

* Multiple isolated environments on a single host
* Preserve volume data when containers are created
* Only recreate containers that have changed
* Variables and moving a composition between environments

### **Docker Compose File**

### **Installation**

If you're running Windows or Mac, Docker Compose is already installed as it comes in the Docker Toolbox. Linux users can easily get their hands on Docker Compose by pasting this lines to your terminal: (First line downloads the current stable release of Docker Compose and the second line applies executable permissions to the binary)

sudo curl -L "https://github.com/docker/compose/releases/download/1.25.4/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose

**Note:** If the command docker-compose fails after installation, check your path. You can also create a symbolic link to **/usr/bin** or any other directory in your path.

Since Compose is written in Python, you can also simply do pip install docker-compose. Test your installation with

$ docker-compose --version

docker-compose version 1.25.4, build 1110ad01

### **Docker Compose File (docker-compose.yml)**

As you understand from .yml extension its YAML file. The syntax for YAML is quite simple. As an example, we will look one of the docker-compose.yml:

version: "3"

services:

es:

image: docker.elastic.co/elasticsearch/elasticsearch:6.3.2

container\_name: es

environment:

- discovery.type=single-node

ports:

- 9200:9200

volumes:

- esdata1:/usr/share/elasticsearch/data

web:

image: prakhar1989/foodtrucks-web

command: python app.py

depends\_on:

- es

ports:

- 5000:5000

volumes:

- ./flask-app:/opt/flask-app

volumes:

esdata1:

driver: local

Let's breakdown what the file above means. At the parent level, the names of services are defined - es and web. For each service that Docker needs to run, we can add additional parameters out of which image is required.

Via other parameters such as command and ports we can reach more information about the container. The volumes parameter specifies a mount point in web container where the code will reside. This is purely optional and is useful if you need access to logs etc. There is also volumes for the es container so that the data we load persists between restarts. Also depends\_on is specified , which tells docker to start the es container before web.

**Note:** You must be inside the directory with the docker-compose.yml file in order to execute most Compose commands.

## **Dockerize your Python Application**

### **Dockerfile**

First, start with a fresh empty directory. As an example, you can call this my\_new\_docker\_project – but feel free to use whatever name you like. This directory defines the context of your build, meaning it contains all of the things you need to do.

Create a new text file in my\_new\_docker\_project called **Dockerfile** (note no extension; on Windows, you may need to save the file as “All types” and put the filename in quotes to avoid automatically appending an extension); use whatever text file editor you already know (you might use Sublime, Notepad++, emacs, nano, or even vim).

In our example, we use the basic Python 3 image as our launching point. Add the following line to your Dockerfile:

FROM python:3

We want to run a basic Python script which we’ll call app. First, we need to add the script to the Dockerfile:

ADD . /app

WORKDIR /app

Our script depends on the some requirements (in our example flask and redis), so we need to make sure we install that before we run app.py. Add this line to your Dockerfile to install:

RUN pip3 install -r requirements.txt

Add this line to your Dockerfile to execute the script:

CMD python3 app.py

Your Dockerfile should look like this:

FROM python:3

ADD . /app

WORKDIR /app

RUN pip3 install -r requirements.txt

CMD python3 app.py

### **requirements.txt**

After creating your Dockerfile, You should create requirements.txt file at the same path. As we mentioned above there are two required python libraries (flask and redis). So inside of the requirements.txt should be like that:

flask

redis

### **Docker-compose.yml**

Create a docker compose.yml at your project directory. In our example, we create it at the my\_new\_docker\_project. We will use version 3 and two services. We still use port 5000 at localhost. We all declared inside of the docker-compose.yml file.

version: '3'

services:

app:

build: .

ports:

- "5000:5000"

volumes:

- .:/app

depends\_on:

- redis

redis:

image: redis

### **app.py File**

Do not have to understand every line of code but this code renders a web page. When you change URL path. It gives you to Bonus and web page changes according to your path's name.

# compose\_flask/app.py

# import Flask and Redis libraries

from flask import Flask

from redis import Redis

import random

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

redis = Redis(host='redis', port=6379)

# declare main route

@app.route('/')

def main():

return 'Hi. In order to earn bonus points enter your name in the url. eg: /John'

# declare a route which gets the visitor's name. Every name has its own bonus points.

@app.route('/')

def greet(name):

bonus = random.randrange(1, 100)

redis.incrby(name, bonus)

return 'Hello %s. You have earned %d bonus points. Your total point is %s.' % (name, bonus, redis.get(name))

# run the flask application on the local development server.

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

app.run(host="0.0.0.0", debug=True)

### **docker-compose up**

💡  *Now time to set on fire our first docker application.*

As you understand from the title you just need to write docker-compose up at the terminal. The project directory should contain:

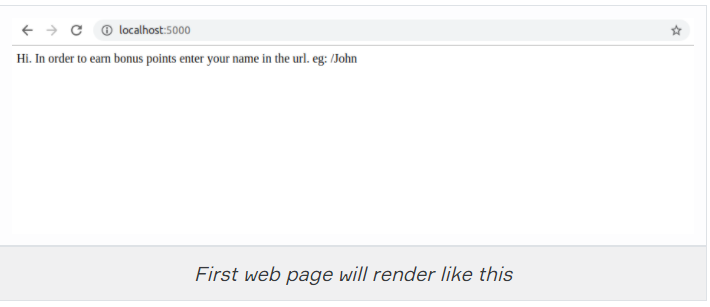
* docker-compose.yml
* Dockerfile
* requirements.txt
* app.py (Your python codes name)

Be sure to this command is running at the project directory. Otherwise, it doesn't work.

clarus-linux@professor:~/my\_new\_docker\_app$ docker-compose up

Your output should seem like this output

Then open your web browser and type localhost/5000.



Please write your name and see the results. For example when i type localhost:5000/clarusway:

