# vagrant@sevenos~]$sudo yum -y install docker-engine

# [vagrant@sevenos~]$sudo yum -y remove docker-engine-1.7.1-1.el6.x86\_64

# Docker Overview

Docker is an open platform for developing, shipping, and running applications. Docker is designed to deliver your applications faster. With Docker you can separate your applications from your infrastructure and treat your infrastructure like a managed application. Docker helps you ship code faster, test faster, deploy faster, and shorten the cycle between writing code and running code.

Docker does this by combining kernel containerization features with workflows and tooling that help you manage and deploy your applications.

# What is the Docker platform?

At its core, Docker provides a way to run almost any application securely isolated in a container. The isolation and security allow you to run many containers simultaneously on your host. The lightweight nature of containers, which run without the extra load of a hypervisor, means you can get more out of your hardware.

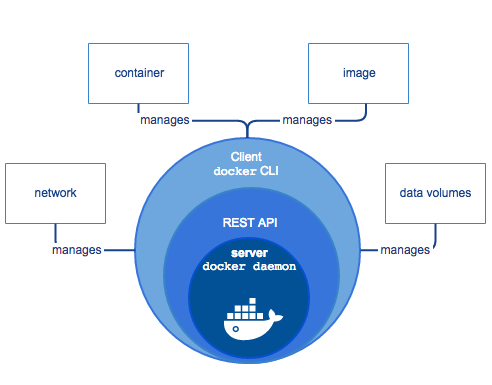
Surrounding the container is tooling and a platform which can help you in several ways:

* Get your applications (and supporting components) into Docker containers
* Distribute and ship those containers to your teams for further development and testing.
* Deploy those applications to your production environment, whether it is in a local data center or the Cloud

# What is 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.
* 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 CLI makes use of the Docker REST API to control or interact with the Docker daemon through scripting or direct CLI commands. Many other Docker applications make use of the underlying API and CLI.

The daemon creates and manages Docker objects. Docker objects include images, containers, networks, data volumes, and so forth.

**Note:** Docker is licensed under the open source Apache 2.0 license.

# What can I use Docker for?

Faster delivery of your applications

Docker is perfect for helping you with the development lifecycle. Docker allows your developers to develop on local containers that contain your applications and services. It can then integrate into a continuous integration and deployment workflow.

For example, your developers write code locally and share their development stack via Docker with their colleagues. When they are ready, they push their code and the stack they are developing onto a test environment and execute any required tests. From the testing environment, you can then push the Docker images into production and deploy your code.

Deploying and scaling more easily

Docker’s container-based platform allows for highly portable workloads. Docker containers can run on a developer’s local host, on physical or virtual machines in a data center, or in the Cloud.

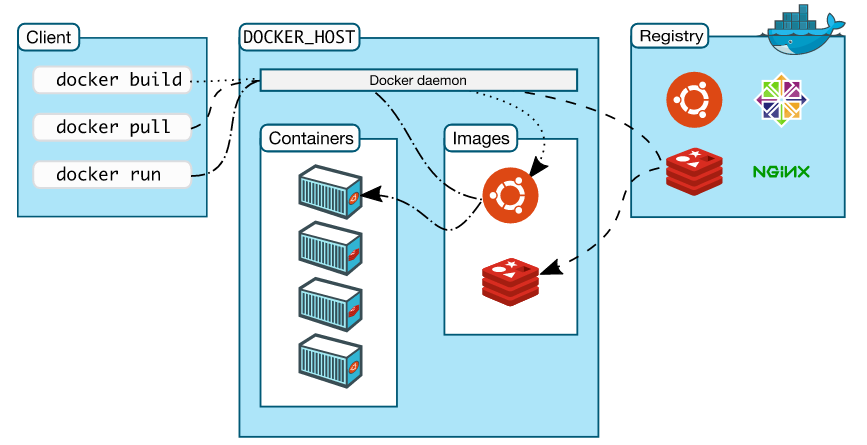
Docker’s portability and lightweight nature also make dynamically managing workloads easy. You can use Docker to quickly scale up or tear down applications and services. Docker’s speed means that scaling can be near real time.

Achieving higher density and running more workloads

Docker is lightweight and fast. It provides a viable, cost-effective alternative to hypervisor-based virtual machines. This is especially useful in high density environments: for example, building your own Cloud or Platform-as-a-Service. But it is also useful for small and medium deployments where you want to get more out of the resources you have.

# What is Docker’s 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. Both the Docker client and the daemon canrun on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate via sockets or through a RESTful API.



# The Docker daemon

As shown in the diagram above, the Docker daemon runs on a host machine. The user does not directly interact with the daemon, but instead through the Docker client.

# The Docker client

The Docker client, in the form of the docker binary, is the primary user interface to Docker. It accepts commands from the user and communicates back and forth with a Docker daemon.

# Inside Docker

To understand Docker’s internals, you need to know about three resources:

* Docker images
* Docker registries
* Docker containers

# Docker images

A Docker image is a read-only template. For example, an image could contain an Ubuntu operating system with Apache and your web application installed. Images are used to create Docker containers. Docker provides a simple way to build new images or update existing images, or you can download Docker images that other people have already created. Docker images are the **build** component of Docker.

# Docker registries

Docker registries hold images. These are public or private stores from which you upload or download images. The public Docker registry is provided with the [Docker Hub](http://hub.docker.com/). It serves a huge collection of existing images for your use. These can be images you create yourself or you can use images that others have previously created. Docker registries are the **distribution** component of Docker. For more information, go to [Docker Registry](https://docs.docker.com/registry/overview/) and[Docker Trusted Registry](https://docs.docker.com/docker-trusted-registry/overview/).

# Docker containers

Docker containers are similar to a directory. A Docker container holds everything that is needed for an application to run. Each container is created from a Docker image. Docker containers can be run, started, stopped, moved, and deleted. Each container is an isolated and secure application platform. Docker containers are the **run**component of Docker.

# How does a Docker image work?

We’ve already seen that Docker images are read-only templates from which Docker containers are launched. Each image consists of a series of layers. Docker makes use of [union file systems](http://en.wikipedia.org/wiki/UnionFS) to combine these layers into a single image. Union file systems allow files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system.

One of the reasons Docker is so lightweight is because of these layers. When you change a Docker image—for example, update an application to a new version— a new layer gets built. Thus, rather than replacing the whole image or entirely rebuilding, as you may do with a virtual machine, only that layer is added or updated. Now you don’t need to distribute a whole new image, just the update, making distributing Docker images faster and simpler.

Every image starts from a base image, for example Ubuntu, a base Ubuntu image, or fedora, a base Fedora image. You can also use images of your own as the basis for a new image, for example if you have a base Apache image you could use this as the base of all your web application images.

**Note:** [Docker Hub](https://hub.docker.com/) is a public registry and stores images.

Docker images are then built from these base images using a simple, descriptive set of steps we call instructions. Each instruction creates a new layer in our image. Instructions include actions like:

* Run a command
* Add a file or directory
* Create an environment variable
* What process to run when launching a container from this image

These instructions are stored in a file called a Dockerfile. A Dockerfile is a text based script that contains instructions and commands for building the image from the base image. Docker reads this Dockerfile when you request a build of an image, executes the instructions, and returns a final image.

# How does a Docker registry work?

The Docker registry is the store for your Docker images. Once you build a Docker image you can push it to a public registry such as [Docker Hub](https://hub.docker.com/) or to your own registry running behind your firewall.

Using the Docker client, you can search for already published images and then pull them down to your Docker host to build containers from them.

[Docker Hub](https://hub.docker.com/) provides both public and private storage for images. Public storage is searchable and can be downloaded by anyone. Private storage is excluded from search results and only you and your users can pull images down and use them to build containers. You can [sign up for a storage plan here](https://www.docker.com/pricing).

# How does a container work?

A container consists of an operating system, user-added files, and meta-data. As we’ve seen, each container is built from an image. That image tells Docker what the container holds, what process to run when the container is launched, and a variety of other configuration data. The Docker image is read-only. When Docker runs a container from an image, it adds a read-write layer on top of the image (using a union file system as we saw earlier) in which your application can then run.

# What happens when you run a container?

Either by using the docker binary or via the API, the Docker client tells the Docker daemon to run a container.

$ docker**run** -i -t ubuntu /bin/bash

The Docker Engine client is launched using the docker binary with the run option running a new container. The bare minimum the Docker client needs to tell the Docker daemon to run the container is:

* What Docker image to build the container from, for example, ubuntu
* The command you want to run inside the container when it is launched, for example,/bin/bash

So what happens under the hood when we run this command?

In order, Docker Engine does the following:

* **Pulls the** ubuntu **image:** Docker Engine checks for the presence of the ubuntu image. If the image already exists, then Docker Engine uses it for the new container. If it doesn’t exist locally on the host, then Docker Engine pulls it from [Docker Hub](https://hub.docker.com/).
* **Creates a new container:** Once Docker Engine has the image, it uses it to create a container.
* **Allocates a filesystem and mounts a read-write** layer**:** The container is created in the file system and a read-write layer is added to the image.
* **Allocates a network / bridge interface:** Creates a network interface that allows the Docker container to talk to the local host.
* **Sets up an IP address:** Finds and attaches an available IP address from a pool.
* **Executes a process that you specify:** Runs your application, and;
* **Captures and provides application output:** Connects and logs standard input, outputs and errors for you to see how your application is running.

You now have a running container! Now you can manage your container, interact with your application and then, when finished, stop and remove your container.

# The underlying technology

Docker is written in Go and makes use of several kernel features to deliver the functionality we’ve seen.

# Namespaces

Docker takes advantage of a technology called namespaces to provide the isolated workspace we call the container. When you run a container, Docker creates a set of namespaces for that container.

This provides a layer of isolation: each aspect of a container runs in its own namespace and does not have access outside of it.

Some of the namespaces that Docker Engine uses on Linux are:

* **The** pid **namespace:** Process isolation (PID: Process ID).
* **The** net **namespace:** Managing network interfaces (NET: Networking).
* **The** ipc **namespace:** Managing access to IPC resources (IPC: InterProcess Communication).
* **The** mnt **namespace:** Managing mount-points (MNT: Mount).
* **The** uts **namespace:** Isolating kernel and version identifiers. (UTS: Unix Timesharing System).

# Control groups

Docker Engine on Linux also makes use of another technology called cgroups or control groups. A key to running applications in isolation is to have them only use the resources you want. This ensures containers are good multi-tenant citizens on a host. Control groups allow Docker Engine to share available hardware resources to containers and, if required, set up limits and constraints. For example, limiting the memory available to a specific container.

# Union file systems

Union file systems, or UnionFS, are file systems that operate by creating layers, making them very lightweight and fast. Docker Engine uses union file systems to provide the building blocks for containers. Docker Engine can make use of several union file system variants including: AUFS, btrfs, vfs, and DeviceMapper.

# Container format

Docker Engine combines these components into a wrapper we call a container format. The default container format is called libcontainer. In the future, Docker may support other container formats, for example, by integrating with BSD Jails or Solaris Zones.