# What is a container?

A **container** is a lightweight, stand-alone, executable package of *a piece of software* that includes everything needed to run it: code, runtime, system tools, system libraries and settings. An **image** is a binary file, essentially a snapshot of a container. Like a blueprint, images are used to create containers.

* “To use a programming metaphor, if an image is a class, then a container is an instance of a class—a runtime object.”

<https://www.docker.com/what-container>

<http://blog.codesupport.info/docker-images-vs-containers/>

# Why containers?

Containerized software will always run the same, regardless of the environment. For example, you can develop containerized applications on Windows or Mac for development, and deploy to Linux servers for production execution. This reduces friction when developers have varied environments as well as minimizes issues when deploying to different staging or production environments.

Containers run on a single machine that share the machine's operating system kernel; they start instantly and use less compute and RAM. Images are constructed from filesystem layers and share common files. This minimizes disk usage and image downloads are much faster. In these ways, containers help to optimize the use of hardware resources.

Containers also isolate applications from one another as well as the underlying hardware. This allows other containers to continue to function even if one container has an issue, assuming runtime constraints are set for each container.

<https://www.cio.com/article/2924995/software/what-are-containers-and-why-do-you-need-them.html>

# What is Docker?

Docker is a software technology providing containers, promoted by the company [Docker, Inc.](https://www.docker.com/) “Docker” is often used as a synonym for containers\*.

<https://coreos.com/rkt/>

<https://hub.docker.com/>

\* In this document, all references to containers refer to containers created with the Docker format.

## Docker vs. Container Orchestration

Docker is focused on the individual container\*. However, deployments of multiple containers typically rely on some type of container orchestration that 1) coordinates/schedules deployment of containers and 2) enables cross-container communication.

<https://dzone.com/articles/docker-orchestration-what-it>

Some container orchestration options are: Kubernetes, Docker Cloud/Docker Swarm, Mesos and Amazon ECS.

\* Docker installations include Docker Engine which is the foundational container execution technology enabling basic container deployment and management.

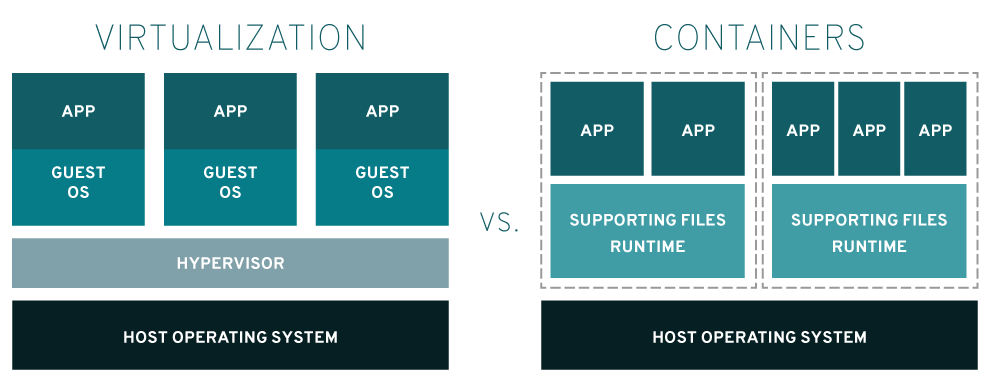
<https://platform9.com/blog/compare-kubernetes-vs-ecs/>

<https://mesosphere.com/blog/docker-vs-kubernetes-vs-apache-mesos/>

# Containers vs. Virtualization

With virtualization technology, the package that can be passed around is a virtual machine, and it includes an entire operating system as well as the application. A physical server running three virtual machines would have a hypervisor and **three separate operating systems running on top of it**.

By contrast a server running three containerized applications with Docker runs **a single operating system**, and each container shares the operating system kernel with the other containers. Shared parts of the operating system are read only, while each container has its own mount (i.e., a way to access the container) for writing. That means the containers are much more lightweight and use far fewer resources than virtual machines.



<https://www.redhat.com/en/topics/virtualization>

# Demo

Windows 7/8 - <https://docs.docker.com/toolbox/toolbox_install_windows/>

Windows 10 - <https://docs.docker.com/docker-for-windows/install>

Mac - <https://docs.docker.com/docker-for-mac/install/>

<https://docker-curriculum.com/>

<https://github.com/prakhar1989/docker-curriculum/blob/master/static-site/Dockerfile>

# Docker How-To

# Defines the base Docker image. Alpine-Java is a simple Java 8 runtime distribution.  
**FROM** anapsix**/**alpine-java  
  
# Run the (Java) server as the daemon user.  
**USER** daemon  
  
# Copy the locally built code, jars and shell scripts to the image (from sbt).  
**ADD** target**/**pack**/**bin **/**app**/**bin  
**ADD** target**/**pack**/**lib **/**app**/**lib  
  
# Run the server when a container based on this image is being run.  
**ENTRYPOINT** ["/app/bin/server"]  
  
# Allow network traffic to the application on port 8080.  
**EXPOSE** 8080

## Base Images

Most images start with a parent image (or base image).

<https://hub.docker.com/r/anapsix/alpine-java/~/dockerfile/>

<https://docs.docker.com/engine/userguide/eng-image/baseimages/>

## Registry

The Registry is a stateless, highly scalable server side application that stores and lets you distribute Docker images.

Use docker “pull” and “push” commands to move images between registries.

<https://docs.docker.com/registry/>

<https://hub.docker.com/>

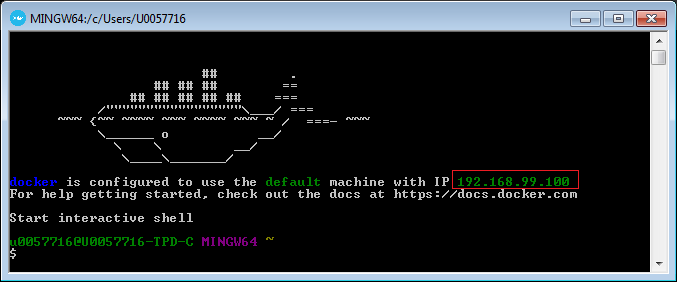
## Accessing Docker Containers

By default, when a container is created, the container is isolated from interaction. You can run commands within a container through a shell.

<http://phase2.github.io/devtools/common-tasks/ssh-into-a-container/>

To access a running service, that service must have exposed a port for communication.

* EXPOSE 8080
* docker run -p 8889:8080 <image name>



<https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/>

<https://github.com/wsargent/docker-cheat-sheet>

## Versioning

Images can be versioned to allow for updates to occur without affecting previous versions. Images without a version specified get the special version label “:latest”.

docker tag <image ID or name> my-private-registry.docker.net:5050/answers/answers-service:1.3