Short Introduction to Docker

What Is Docker?

When we answer this question, we need to clarify the word “docker,” because Docker has become synonymous with containers.

Docker the Company

Docker Inc. is the company behind Docker. Docker Inc. was founded as dotCloud Inc. in 2010 by Solomon Hykes. dotCloud engineers built abstraction and tooling for Linux Containers and used the Linux Kernel features cgroups and namespaces with the intention of reducing complexity around using Linux containers. dotCloud made their tooling open source and changed the focus from the PaaS business to focus on containerization. Docker Inc. sold dotCloud to cloudControl, which eventually filed for bankruptcy.

Docker the Software Technology

Docker is the technology that provides for operating system level virtualization known as containers. It is important to note that this is not the same as hardware virtualization. Docker uses the resource isolation features of the Linux kernel such as cgroups, kernel namespaces, and OverlayFS, all within the same physical or virtual machine. OverlayFS is a union-capable filesystem that combines several files and directories into one in order to run multiple applications that are isolated and contained from one other, all within the same physical or virtual machine.

Understanding Problems that Docker Solves

For the longest period, setting up a developer’s workstation was a highly troublesome task for sysadmins. Even with complete automation of the installation of developer tools, when you have a mix of different operating systems, different versions of operating systems, and different versions of libraries and programming languages, setting up a workspace that is consistent and provides a uniform experience is nearly impossible. Docker solves much of this problem by reducing the moving parts. Instead of targeting operating systems and programming versions, the target is now the Docker engine and the runtime. The Docker engine provides a uniform abstraction from the underlying system, making it very easy for developers to test their code. Things get even more complicated on the production landscape.

Assume that we have a Python web application that is running on Python 2.7 on Amazon Web Services EC2 instance. In an effort to modernize the codebase, the application had some major upgrades, including a change in Python version from 2.7 to version 3.5. Assume that this version of Python is not available in the packages offered by the Linux distribution currently running the existing codebases. Now to deploy this new application, we have the choice of either of the following:

- Replace the existing instance
- Set up the Python Interpreter by
  - Changing the Linux distribution version to one that includes the newer Python packages
  - Adding a third-party channel that offers a packaged version of the newer Python version

- Doing an in-place upgrade, keeping the existing version of the Linux distribution
- Compiling Python 3.5 from sources, which brings in additional dependencies
- Or using something like virtualenv, which has its own set of tradeoffs

Whichever way you look at it, a new version deployment for application code brings about lots of uncertainty. As an operations engineer, limiting the changes to the configuration is critical. Factoring in an operating system change, a Python version change, and a change in application code results in a lot of uncertainty. Docker solves this issue by dramatically reducing the surface area of the uncertainty. Your application is being modernized? No problem. Build a new container with the new application code and dependencies and ship it. The existing infrastructure remains the same. If the application doesn't behave as expected, then rolling back is as simple as redeploying the older container—it is not uncommon to have all the generated Docker images stored in a Docker registry. Having an easy way to roll back without messing with the current infrastructure dramatically reduces the time required to respond to failures.

Knowing the Difference Between Containers and Virtual Machines

Many people assume that since containers isolate the applications, they are the same as virtual machines. At first glance it looks like it, but the fundamental difference is that containers share the same kernel as the host.
Docker only isolates a single process (or a group of processes, depending on how the image is built) and all the containers run on the same host system. Since the isolation is applied at the kernel level, running containers does not impose a heavy overhead on the host as compared to virtual machines. When a container is spun up, the selected process or group of processes still runs on the same host, without the need to virtualize or emulate anything. Figure 1 shows the three apps running on three different containers on a single physical host.

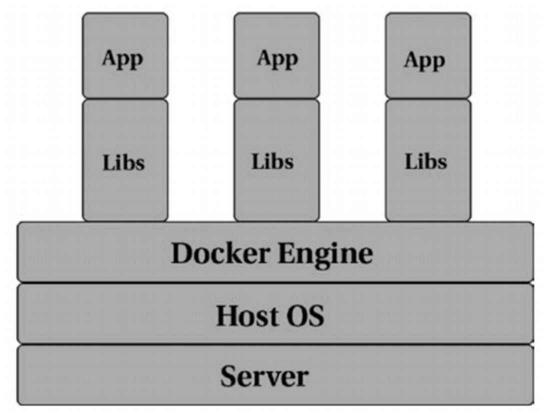

Figure 1. Representation of three apps running on three different containers

In contrast, when a virtual machine is spun up, the hypervisor virtualizes an entire system—from the CPU to RAM to storage. To support this virtualized system, an entire operating system needs to be installed. For all practical purposes, the virtualized system is an entire computer running in a computer. Now if you can imagine how much overhead it takes to run a single operating system, imagine how it'd be if you ran a nested operating system! Figure 2 shows a representation of the three apps running on three different virtual machines on a single physical host.

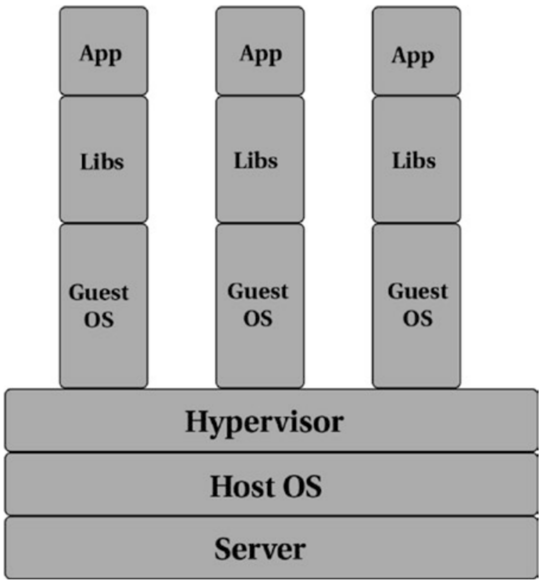

Figure 2. Representation of three apps running on three different virtual machines

Figures 1 and 2 give an indication of three different applications running on a single host. In the case of a VM, not only do we need the application's dependent libraries, we also need an operating system to run the application. In comparison, with containers, the sharing of the host OS's kernel with the application means that the overhead of an additional OS is removed. Not only does this greatly improve the performance, it also lets us improve the resource utilization and minimize wasted compute power.

Docker key words

Layers

A layer is a modification applied to a Docker image as represented by an instruction in a Dockerfile. Typically, a layer is created when a base image is changed—for example, consider a Dockerfile that looks like this:

```
FROM ubuntu
Run mkdir /tmp/logs
RUN apt-get install vim
RUN apt-get install htop
```

Now in this case, Docker will consider Ubuntu image as the base image and add three layers:

- One layer for creating /tmp/logs
- One other layer that installs vim
- A third layer that installs htop

When Docker builds the image, each layer is stacked on the next and merged into a single layer using the union filesystem. Layers are uniquely identified using sha256 hashes. This makes it easy to reuse and cache them. When Docker scans a base image, it scans for the IDs of all the layers that constitute the image and begins to download the layers. If a layer exists in the local cache, it skips downloading the cached image.

Docker Image

Docker image is a read-only template that forms the foundation of your application. It is very much similar to, say, a shell script that prepares a system with the desired state. In simpler terms, it's the equivalent of a cooking recipe that has step-by-step instructions for making the final dish.
A Docker image starts with a base image—typically the one selected is that of an operating system are most familiar with, such as Ubuntu. On top of this image, we can add build our application stack adding the packages as and when required.
There are many pre-built images for some of the most common application stacks, such as Ruby on Rails, Django, PHP-FPM with nginx, and so on. On the advanced scale, to keep the image size as low as possible, we can also start with slim packages, such as Alpine or even Scratch, which is Docker's reserved, minimal starting image for building other images.
Docker images are created using a series of commands, known as instructions, in the Dockerfile. The presence of a Dockerfile in the root of a project repository is a good indicator that the program is container-friendly. We can build our own images from the associated Dockerfile and the built image is then published to a registry. For now, consider the Docker image as the final executable package that contains everything to run an application. This includes the source code, the required libraries, and any dependencies.

Docker Container

A Docker image, when it's run in a host computer, spawns a process with its own namespace, known as a Docker container. The main difference between a Docker image and a container is the presence of a thin read/write layer known as the container layer. Any changes to the filesystem of a container, such as writing new files or modifying existing files, are done to this writable container layer than the lower layers.
An important aspect to grasp is that when a container is running, the changes are applied to the container layer and when the container is stopped/killed, the container layer is not saved. Hence, all changes are lost. This aspect of containers is not understood very well and for this reason, stateful applications and those requiring persistent data were initially not recommended as containerized applications. However, with Docker Volumes, there are ways to get around this limitation.

Bind Mounts and Volumes

When a container is running, any changes to the container are present in the container layer of the filesystem. When a container is killed, the changes are lost and the data is no longer accessible. Even when a container is running, getting data out of it is not very straightforward. In addition, writing into the container's writable layer requires a storage driver to manage the filesystem. The storage driver provides an abstraction on the filesystem available to persist the changes and this abstraction often reduces performance.
For these reasons, Docker provides different ways to mount data into a container from the Docker host: volumes, bind mounts, and tmpfs volumes. While tmpfs volumes are stored in the host system's memory only, bind mounts and volumes are stored in the host filesystem.

Docker Registry

You can leverage existing images of common application stacks—have you ever wondered where these are and how you can use them in building your application? A Docker Registry is a place where you can store Docker images so that they can be used as the basis for an application stack. Some common examples of Docker registries include the following:

- Docker Hub
- Google Container Registry

- Amazon Elastic Container Registry
- JFrog Artifactory

Most of these registries also allow for the visibility level of the images that you have pushed to be set as public/private. Private registries will prevent your Docker images from being accessible to the public, allowing you to set up access control so that only authorized users can use your Docker image.

Dockerfile

A Dockerfile is a set of instructions that tells Docker how to build an image. A typical Dockerfile is made up of the following:

- A FROM instruction that tells Docker what the base image is
- An ENV instruction to pass an environment variable
- A RUN instruction to run some shell commands (for example, install-dependent programs not available in the base image)
- A CMD or an ENTRYPOINT instruction that tells Docker which executable to run when a container is started

As you can see, the Dockerfile instruction set has clear and simple
syntax, which makes it easy to understand.

Docker Engine

Docker Engine is the core part of Docker. Docker Engine is a client-server application that provides the platform, the runtime, and the tooling for building and managing Docker images, Docker containers, and more. Docker Engine provides the following:

- Docker daemon
- Docker CLI
- Docker API

Docker Daemon

The Docker daemon is a service that runs in the background of the host computer and handles the heavy lifting of most of the Docker commands. The daemon listens for API requests for creating and managing Docker objects, such as containers, networks, and volumes. Docker daemon can also talk to other daemons for managing and monitoring Docker containers. Some examples of inter-daemon communication include communication Datadog for container metrics monitoring and Aqua for container security monitoring.

Docker CLI

Docker CLI is the primary way that you will interact with Docker. Docker CLI exposes a set of commands that you can provide. The Docker CLI forwards the request to Docker daemon, which then performs the necessary work.
While the Docker CLI includes a huge variety of commands and sub-commands, the most common commands that we will work are :

docker build
docker pull
docker run
docker exec

for more informations : https://docs.docker.com/engine/reference/commandline/cli/

Type docker help <command> to have information on command an associated parameters.

Docker API

Docker also provides an API for interacting with the Docker Engine. This is particularly useful if there's a need to create or manage containers from within applications. Almost every operation supported by the Docker CLI can be done via the API.

Docker Compose

Docker Compose is a tool for defining and running multi-container applications. Much like how Docker allows you to build an image for your application and run it in your container, Compose use the same images in combination with a definition file (known as the compose file) to build, launch, and run multi-container applications, including dependent and linked containers.
The most common use case for Docker Compose is to run applications and their dependent services (such as databases and caching providers) in the same simple, streamlined manner as running a single container application.

Docker Machine

Docker Machine is a tool for installing Docker Engines on multiple virtual hosts and then managing the hosts. Docker Machine allows you to create Docker hosts on local as well remote systems, including on cloud platforms like Amazon Web Services, DigitalOcean, and Microsoft Azure.