Understanding Docker Data Storage and Persistence

docker

Introduction

Applications are generally built using three types of infrastructure: compute, networking, and storage. Docker has provided a means to abstract all of these components, greatly increasing the agility and control organizations have around how they build, ship, and run their applications.

In this paper we're going to take a look at how Docker handles storage. We'll examine how images and containers are built, and what makes them so efficient. This paper also covers how to handle the issue of data persistence with Docker, and wraps up with a discussion around things to consider around storage as organizations look to deploy Docker.

Image and Container Architecture

Docker images form the basis for Docker containers. Every container is launched from an image. But, how do containers relate to images architecturally? And, how do images related to other images?

It all comes down to layers.

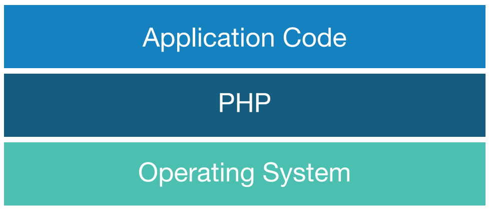
It might be best to start with an example. If a developer was building a PHP-based web application, they would start with a base operating system. On top of that they would install PHP, and finally their application code. This general process is the same whether it's being done with Docker, on a physical box, or in a VM.

However, in a case where Docker is not being used, the resulting files and subdirectories from each installation step would all be intermingled. All of the installed components are written to a single monolithic file system.

By contrast, with Docker each step results in a new, isolated, read-only layer being added to our image. For instance, every Docker image starts with some base operating system layer. As components are added to this image, new layers are created. Each layer separates the newly added components from any components that were previously installed. The underlying layers remain untouched.

If a newly added layer includes a file or directory that exists in a lower layer, the top most instance of that file or directory takes precedence. This is one of the fundamental ways Docker helps address scenarios like library mismatches.

Conceptually a Docker image implementing the PHP example cited above would look like the following diagram:



Note that while we represent the operating system, PHP, and application code as single layers in our diagram, the reality is that each one of them is actually made up of multiple layers.

This becomes evident when you review the output of a Docker pull command.

If we examine the output for a pull of the official `Debian:Jessie` image we can see that it's actually made up of two different layers `(03e1855d4f31 and a3ed95caeb02):`

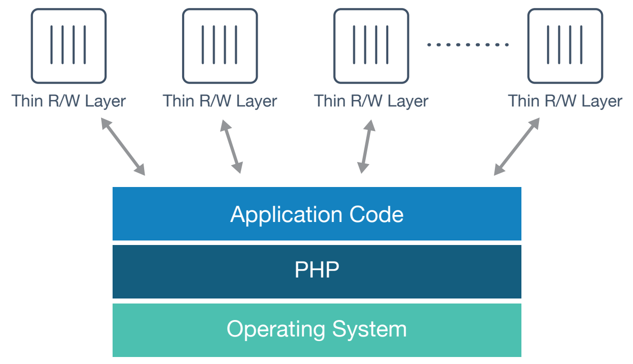
```
$ docker pull debian:jessie
jessie: Pulling from library/debian

03e1855d4f31: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:7f82488af7fe57134b058b925928b9ce-
3b96384229a566d5236bd09ec2f82ef5
Status: Downloaded newer image for debian:Jessie
```

As mentioned before, image layers are created as packages and code are installed onto an existing image. Since the Debian image is just an operating system (and a stripped down one at that), we would expect it to have only a couple of layers. In the case of the official PHP image (which we'll look at later in this document) it's made up of eight distinct layers.

Containers vs. Images

What differentiates a container from an image is the addition of a "container layer. When a container is instantiated, a read/write layer is added on top of the underlying read-only image layers. Any changes made to the container while it is running, for instance a log file being written, are reflected in the container layer; the underlying image layers are never affected. As soon as a change is initiated to an underlying layer, a new read-write layer is added. This is referred to as copy on write.



Using a container layer provides several benefits, the two most prominent being: speed and minimizing the storage space needed.

When a new container is started, Docker does not clone the read-only base image, it simply creates a new container layer and links that to the existing base image. This can be done extremely quickly. In a published test Docker found that even with 50,000 nodes running on a 1,000 node Swarm cluster, a new container can be spun up in less than 400ms.

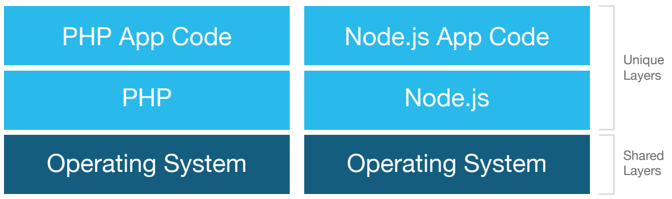
Docker customers have been able to capitalize on the near-instantaneous instantiation times to drastically cut their application testing cycles.

Consider a case where a company wants to test an application at scale and are using full-clone virtual machines. Full-clone VMs in the best scenario take several minutes to boot, and most virtual machine management platforms can only boot a handful machines simultaneously. Based on these factors standing up 1,000 full-clone virtual machines could hours if not days. Meaning the test cycle itself could take days if not weeks.

By contrast, the same application running inside of a Docker container can be started in less than half a second. Standing up 1,000 containers becomes trivial. Test cycle times can be slashed from days to hours. This can translate into measurable savings as well as increased agility.

It's important to note that Docker will not only share the base image between containers, but it will also share the same layers between different images.

For instance, if an organization uses a specific operating system layer across all images, then that base OS layer will be shared between all containers on a given Docker host. For instance, if we extend our original example to include a second application built with Node.js, but built on the same OS layer as our PHP application, the only new layers that would need to be pulled onto our Docker host would be for Node.js and our application code. The base operating system layer would already be in place on the Docker host since it was used for our PHP application.



If we examine the output of another Docker pull command, we can see this in practice.

The official PHP image on Docker hub is actually based off the `Debian:Jessie` image we examined earlier. When we pull the PHP image, we'll note that the first layer pulled `(03e1855d4f31)` is already present because it's shared with the `Debian:Jessie` image.

```
$ docker pull php
Using default tag: latest
latest: Pulling from library/php
03e1855d4f31: Already exists
a3ed95caeb02: Pull complete
18f8f35c7f98: Pull complete
252f4816c8e7: Pull complete
f8ac546f9ad6: Pull complete
bd844e35a25f: Pull complete
41ab119aa8ec: Pull complete
bc8a8b159c5b: Pull complete
Digest: sha256:0f596fe8533d8fa0ebc59897f48e03261244c-
67c5aff3d02e7b2354c57d2b3f5
Status: Downloaded newer image for php:latest
```

By only instantiating a single read-only image regardless of the number of containers created, as well as only storing layers one time, regardless of the number of different images they are used in, Docker is able to drastically reduce the amount of storage it uses. Each additional container will only require a small read-write layer vs. having to clone the entire image with each new instantiation. This translates into a direct savings on storage infrastructure when compared to other technologies like full-clone virtual machines.

Docker Storage Drivers

One of the core principles that guides Docker development is "Batteries included, but replaceable." This is the idea that the core platform will ship with certain capabilities, but will be architected in a way that key components can be swapped out so users can choose the right technologies for their environments.

Docker storage is architected to comply with this approach though two different mechanisms: Docker storage drivers and Docker volume drivers. Docker volume drivers will be discussed later in this document, for now we'll focus on Docker storage drivers.

The Docker storage driver is the piece of code that facilitates and manages the layering discussed earlier in this paper. The storage driver is based on a Linux file system or volume manager.

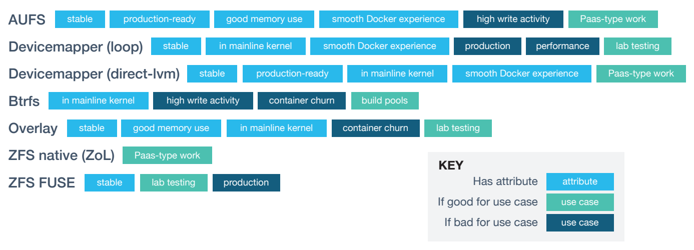
Docker ships with a default Docker storage driver enabled by default. The specific driver is distribution dependent, for instance for Ubuntu it's AUFS whereas for Red Hat it's device mapper.

In addition to the two drivers previously mentioned (AUFS and device mapper) Docker also supports the following Docker storage drivers:

- OverlayFS
- Btrfs
- VFS
- ZFS

Which device driver to choose is ultimately based on the needs of each individual organization. It is important to note that a given Docker host can only run one storage driver. You can however, have different hosts running different storage drivers. And, while it's true that some storage drivers (specifically Btrfs and ZFS) require a matching backing file system, IT admins can choose the underlying storage hardware that best meets their needs including storage array networks, network attached storage, or even a locally-mounted disk.

Each storage driver has its own sets of pros and cons, and the Docker documentation provides excellent guidance on how to choose the right one. However, the graphic below provides a good summary of some things to consider when making a decision on a storage driver.


AUFS stable production-ready good memory use smooth Docker experience high write activity Paas-type work
Devicemapper (loop) stable in mainline kernel smooth Docker experience production performance lab testing
Devicemapper (direct-lvm) stable production-ready in mainline kernel smooth Docker experience Paas-type work
Btrfs in mainline kernel high write activity container churn build pools
Overlay stable good memory use in mainline kernel container churn lab testing
ZFS native (ZoL) Paas-type work
ZFS FUSE stable lab testing production
KEY
Has attribute attribute
If good for use case use case
If bad for use case use case

Data Persistence with Docker Volumes

There are some well-accepted best practices when it comes to developing containerized services. For instance, containers should optimally provide a single service. Containers should also be ephemeral – if a container dies, a new one should be able to be brought up in its place with no additional configuration.

The ephemeral nature of containers offers a myriad of possibilities when it comes to scaling and disaster recovery, but it also has some caveats. One of those caveats is that while the container is ephemeral, so is the data inside the container. If a container dies, anything in the read-write layer is lost.

The issue is that applications need permanent data storage, whether it's linking to a database, or saving off audit logs. With Docker data persistence is achieved through volumes.

Image and containers are, as previously discussed, managed by the Docker storage driver. The storage driver stores all the layers under a designated subdirectory on the Docker host. For instance, with AUFS this directory is typically /var/lib/docker. Anything written to a container is stored inside of this directory structure, and is managed by the Docker storage driver.

A volume is simply a directory on the Docker host that exists outside of the directory structure managed by the storage driver. The directory is mapped from the Docker host to a directory inside the container. Anything written to or read from that directory bypasses the storage driver, and operates at host speeds. When a container is removed, the volume persists and the data is still available.

Another interesting point is that multiple containers can mount to the same volume. For instance, logs can be written to a Docker volume, and then containers that run a report processor can access that volume.

It is important to note that Docker does not do anything to control simultaneous access to the volume. The best way to think about it is that Docker is just providing a sub directory into which data can be read or written. It's up to the application or service accessing that data to ensure its integrity.

Storage Volume Plug-ins

Just like there are plug-ins to allow Docker to leverage different files systems, there are also plugins to allow volumes to be managed regardless of the underlying storage system. Volume plug translate Docker volume commands into the appropriate API calls for the given storage hardware. This allows IT organizations the flexibility to choose the right storage platform, yet still use Docker in a consistent way regardless of the platform chose.

Additional Considerations

When looking at implementing a Docker-based containers as a service (CaaS) platform in a production environment it's important to consider some of the following points around storage as

Optimizing Store Space Usage

As mentioned earlier Docker does an excellent job of efficiently utilizing storage resources. However, there are still steps that IT personnel can take to ensure they are using the least amount of storage possible.

Since Docker will only store a given image layer once, one best practice is to settle on a defined set of core images. A core component of any CaaS platform is the registry. Whether it's private repositories on Docker Cloud, or an internal registry provided by Docker Datacenter, IT operations staff should work to provide their development staff with a consistent set of vetted base images.

By doing this they will not only minimize the storage impact of future images and containers based on those images, they'll also be able to ensure that the images that are being deployed are optimized for their organization.

Backup and Restore

While containers should be designed to be ephemeral, there still needs to be a mechanism to back up the Docker data volumes.

Fortunately, this is a pretty straight forward process since Docker data volumes are merely subdirectories on a Docker host. As such, volumes can be backed up and restored using standard processes and tools.

Data Concurrency and Security

We briefly touched on this earlier in the document, but it bears repeating. Because Docker volumes are simply subdirectories on the Docker host, they are not offered any special protections by the Docker daemon. These directories, and the data in them, need to be treated in the same manner as they would be in a legacy environment.

It's up to the applications and services access the data to ensure that appropriate safeguards are in place to prevent data corruption. Additionally, while it's possible to directly access the data in the Docker host subdirectory, this is not advisable as there are no safeguards in place to prevent data corruption.

Volume Cleanup

Unless a volume is explicitly deleted when a container is removed, they will remain on the host. In some cases this is exactly the behavior desired. For instance, a volume may contain a database that is used for testing. New versions of an app container can be started, and pointed at the volume for testing, and then removed after the testing is completed without removing the volume.

However, if tight operational procedures aren't followed, "orphaned" volumes can become problematic. The best way to deal with this is through defined processes built around named volumes. By using intelligent names on containers, operations staff can quickly decipher the purpose of the volume, and decide whether or not to delete it.

Conclusion

Docker is revolutionizing the way applications are built, shipped, and run. Docker's storage capabilities allow organizations a high-level of flexibility – whether it's providing persistent storage, or allowing seamless integration with existing infrastructure. This means that developers can model the full application stack – compute, network, storage and code, in software and know that it will work wherever it's deployed. Operations staff can rest assured that their existing technology investments are protected, as well as know that they can easily transition to new infrastructure as their business needs evolve.

www.docker.com

March 24, 2016