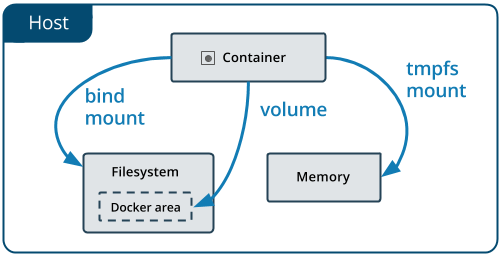
**Docker Storage / Managing Data in Docker**

**Container Data:**

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, and it can be difficult to get the data out of the container if another process needs it.
* A container’s writable layer is tightly coupled to the host machine where the container is running. You can’t easily move the data somewhere else.
* Writing into a container’s writable layer requires a storage driver to manage the filesystem. The storage driver provides a union filesystem, using the Linux kernel.
* Docker has two options (volumes and bind mounts (temporary mounts: tmpfs in Linux / named pipes in Windows)) for containers to store files in the host machine, so that the files are persisted even after the container stops.
* No matter which type of mount you choose to use, the data looks the same from within the container.



1. **Volumes**

* Parts of the host file system which is managed by Docker (/var/lib/docker/volumes/ on Linux).
* Created and completely managed by Docker.
* Volumes are the preferred mechanism for persisting data generated by and used by Docker containers.
* Volumes use **rprivate bind propagation**, and bind propagation is not configurable for volumes.
* Volume does not increase the size of the containers using it, and the volume’s contents exist outside the lifecycle of a given container.
* Multiple containers can mount the same volume simultaneously, and it can be mounted **read-write** for some of them and **read-only** for others, at the same time.
* If you mount an empty volume into a directory in the container in which files or directories exist, these files or directories are propagated (copied) into the volume.
* if you start a container and specify a volume which does not already exist, an empty volume is created and this is a good way to pre-populate data that another container needs.
* -v or --volume flag was used for standalone containers and –mount flag used for swarm services, however Docker 17.06 onwards --mount flag can be used for standalones containers.



**Advantages over bind mounts:**

* Volumes are easier to back up or migrate than bind mounts.
* You can manage volumes using Docker CLI or API.
* Volumes work on both Linux and Windows containers.
* Volumes can be more safely shared among multiple containers.
* Volume drivers help to store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
* New volumes can have their content pre-populated by a container.

**Use cases for volumes:**

* When the Docker host is not guaranteed to have a given directory or file structure. Volumes helps to decouple the configuration of the Docker host from the container runtime.
* When you want to store your container’s data on a remote host or a cloud provider, rather than locally.
* When you need to back up, restore, or migrate data from one Docker host to another, volumes are a better choice. You can stop containers using the volume, then back up the volume’s directory.

(/*var/lib/docker/volumes/<volume-name>)*

**Share data among machines:**

To build fault-tolerant applications, we required to configure multiple service replicas to have the access on same files.

Ways to achieve this:

1. Add logic to application to store files/objects in cloud storage
2. Create a volume with driver that supports writing files to an external storage (ex: Amazon S3, NFS ..etc)

Volume drivers: allows to abstract the underlying storage system from the application logic.

****

**-v, --volume and –mount flags:**

**-v**or**--volume:** Consists of three fields, separated by colon characters (:). The fields must be in the correct order, and the meaning of each field is not immediately obvious.

* **First field:** name of the volume, and is unique on a given host machine
* **Second field:** path where the file or directory are mounted in the container.
* **Third field:** optional, and is a comma-separated list of options

**--mount:** Consists of multiple key-value pairs, separated by commas and each consisting of a <key>=<value> tuple. The --mount syntax is more verbose than -v or –volume.

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Key** | **Values** | **Description** |
| 1 | type | **volume**, bind and tmpfs | Choose the mount type |
| 2 | source / src | volume | Named volume |
| 3 | destination / dst / target | Container path | path where the file or directory is mounted in the container |
| 4 | readonly |  | if present, causes the bind mount to be mounted into the container as read-only. |
| 5 | volume-driver | local | Volume driver |
| 6 | volume-opt | type=nfs  device=<nfs server: nfs path> | option, which can be specified more than once, takes a key-value pair consisting of the option name and its value. |

**Create and manage volumes**

**Usage: docker volume COMMAND**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Command** | **Options** | **Description** |
| 1 | create | -d, --driver  (string) | volume driver name (default "local") |
| --label  (list) | Set metadata for a volume |
| -o, --opt map | Set driver specific options (default map[]) |
| 2 | inspect | -f, --format  (string) | Format the output using the Go template |
| 3 | ls | -f, --filter  (filter) | Provides filter values (ex: ‘dangling=true’)  $ docker volume ls filter ‘dangling=true) |
| --format  (string) | Pretty print using Go template |
| -q, --quiet | Only display volume names |
| 4 | prune | -f, --force | Do not prompt for confirmation |
| --filter | Provides filter (ex: label=<label-name>) |
| 5 | rm | -f, --force | Force removal of one more volumes |

**Examples:**

**Create a volume:**

$ docker volume create my-vol

**List created/available volumes:**

$ docker volume ls

DRIVER VOLUME NAME

local my-vol

**Starting a container with a volume using -v flag:**

$ docker run -d --name nginx-dev -v nginx-volume:/app nginx:latest

$ docker volume ls

DRIVER VOLUME NAME

local nginx-volume

**Starting a container with a volume using --mount flag:**

$ docker run -d --name nginx-dev2 --mount source=nginx-volume2,target=/app nginx:latest

$ docker volume ls

DRIVER VOLUME NAME

local nginx-volume2

**Start a service with volumes (-v not supported)**

$ docker service create --replicas=2 --name nginx-service --mount \

source=nginx-volume3,target=/app nginx:latest

$ docker volume ls

DRIVER VOLUME NAME

local nginx-volume3

**Populate a volume using a container with -v flag:**

$ docker run -d --name nginx-dev3 -v nginx-vol-pages:/usr/share/nginx/html nginx:latest

**Populate a volume using a container with --mount flag:**

$ docker run -d --name nginx-dev4 --mount \

source=nginx-vol-pages2,target=/usr/share/nginx/html nginx:latest

$ docker volume ls

DRIVER VOLUME NAME

local nginx-vol-pages

local nginx-vol-pages2

**Use a read-only volume (-v):**

$ docker run -d --name nginx-ro -v nginx-vol-ro:/usr/share/nginx/html:ro nginx:latest

**Use a read-only volume (--mount):**

$ docker run -d --name nginx-ro2 --mount \

source=nginx-vol-ro,target=/usr/share/nginx/html nginx:latest

$ docker volume ls

DRIVER VOLUME NAME

local nginx-vol-ro

**Using volume driver (sshfs):**

1. Setup two nodes (manager and worker) with ssh connectivity (manager-to-worker).
2. Install **vieux/sshfs** plugin in manager node ($ docker plugin install --grant-all-permissions vieux/sshfs)
3. Create volume using volume driver (use password=password if ssh authentication not enabled).

$ docker volume create --driver vieux/sshfs -o sshcmd=docker@node1:/home/docker ssh-vol

$ docker volume ls

DRIVER VOLUME NAME

vieux/sshfs:latest ssh-vol

1. Create a container with sshfs volume.

$ docker run -d --name sshfs-container --volume-driver vieux/sshfs –mount \

src=ssh-vol ,target=/app,volume-opt=sshcmd=docker@node1:/home/docker, \

volume-opt=password=testpassword nginx:1.17

**Create a service which creates an NFS volume**

$ docker service create -d --name nfs-service --mount \

'type=volume,source=nfsvolume,target=/app,volume-driver=local, \

volume-opt=type=nfs,volume-opt=device=:/,"volume-opt=o=10.0.0.10,rw,nfsvers=4,async"' \

nginx:1.17

**Backup:**

* Volumes are useful for backups, restores, and migrations.
* Use the --volumes-from flag to create a new container that mounts that volume.

**Backup container:**

$ docker run -v /dbdata --name dbstore ubuntu /bin/bash

* + Launch a new container and mount the volume from the dbstore container
  + Mount a local host directory as /backup
  + Pass a command that tars the contents of the dbdata volume to a backup.tarfile inside our /backup directory.

$ docker run --rm --volumes-from dbstore -v $(pwd):/backup ubuntu tar cvf /backup/backup.tar /dbdata

When the command completes and the container stops, we are left with a backup of our dbdata volume.

**Restore container from backup:**

* With the backup just created, you can restore it to the same container, or another that you made elsewhere.

For example, create a new container named dbstore2:

$ docker run -v /dbdata --name dbstore2 ubuntu /bin/bash

Then un-tar the backup file in the new container`s data volume:

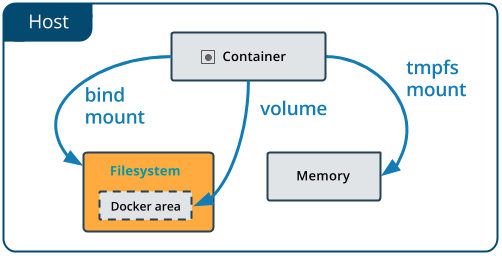
$ docker run --rm --volumes-from dbstore2 -v $(pwd):/backup \

ubuntu bash -c "cd /dbdata && tar xvf /backup/backup.tar --strip 1"

Use above techniques to automate backup, migration and restore testing using your preferred tools.

1. **Bind mounts:**

* Bind mounts are mounted in docker host file system and not managed by Docker (CLI).
* Bind mounts have limited functionality when compared to volumes.
* The file or directory referenced by its full/absolute path or relative path on host file system.
* Bind mounts are performant, but rely on the host file system structure.
* -v or --volume flag was used for standalone containers and –mount flag used for swarm services, however Docker 17.06 onwards --mount flag can be used for standalones containers.



**-v, --volume and –mount flags:**

**-v**or**--volume:** Consists of three fields, separated by colon characters (:). The fields must be in the correct order, and the meaning of each field is not immediately obvious.

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Field** | **Values / Options** | **Description** |
| 1 | First | Host path | Path to the file or directory in host machine |
| 2 | Second | Container path | Path where the file or directory is mounted in the container. |
| 3 | Third | ro, consistent, delegated,  cahed, z, and Z | Optional, and is a comma-separated list of options |

**--mount:** Consists of multiple key-value pairs, separated by commas and each consisting of a <key>=<value> tuple. The --mount syntax is more verbose than -v or –volume.

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Key** | **Values** | **Description** |
| 1 | type | volume, **bind** and tmpfs | Choose the mount type |
| 2 | source / src | Host path | Docker host file system path |
| 3 | destination / dst / target | Container path | path where the file or directory is mounted in the container |
| 4 | readonly |  | If present, causes the bind mount to be mounted into the container as read-only. |
| 5 | bind-propagation | rprivated, private,  rshared, shared  rslave, slave | If present, change the bind propagation based on the value provided |
| 6 | consistency | consistent, delegated,  cached | Only applies in Docker Desktop (ignores in all other systems), and affects based on the value. |
| 7 |  | Z and z | --mount, options does not support to modify selinux labels |

**Difference between -v and --mount:**

* -v or --volume: creates the endpoint (always directory) even if does not exist in host file system.
* --mount: if file system path does not exist, then it throws an error.

**Examples:**

**Start a container with bind mount (-v):**

$ docker run -d --name nginx-dev -v $(pwd)/nginx-app:/app nginx:1.16

Above creates nginx-data directory in the current directory as follows:

drwxr-xr-x. 2 root root 6 Jan 18 10:30 nginx-app

**Start a container with bind mount (--mount):**

$ docker run -d --name nginx-dev2 --mount \

type=bind,source="$(pwd)"/nginx-app,target=/app nginx:1.16

Used existing nginx-data directory.

**Configure bind propagation:**

* Bind propagation defaults to **rprivate** for both bind mounts and volumes.
* Bind propagation only configurable for bind mounts, and only on Linux host machines.
* Bind propagation refers to whether or not mounts created within a given bind-mount or named volume can be propagated to replicas of that mount.
* Consider a mount point /mnt, which is also mounted on /tmp. The propagation settings control whether a mount on /tmp/a would also be available on /mnt/a
* Each propagation setting has a recursive counterpoint (recursion).
* Before setting bind propagation on a mount point, the host filesystem needs to already support bind propagation.

|  |  |  |
| --- | --- | --- |
| **S. No.** | **Propagation Setting** | **Description** |
| 1 | private | The mount is private. Sub-mounts within it are not exposed to replica mounts, and sub-mounts of replica mounts are not exposed to the original mount. |
| 2 | rprivate | The default. The same as private, meaning that no mount points anywhere within the original or replica mount points propagate in either direction. |
| 3 | shared | Sub-mounts of the original mount are exposed to replica mounts, and sub-mounts of replica mounts are also propagated to the original mount. |
| 4 | rshared | The same as shared, but the propagation also extends to and from mount points nested within any of the original or replica mount points. |
| 5 | slave | similar to a shared mount, but only in one direction. If the original mount exposes a sub-mount, the replica mount can see it. However, if the replica mount exposes a sub-mount, the original mount cannot see it. |
| 6 | rslave | The same as slave, but the propagation also extends to and from mount points nested within any of the original or replica mount points. |

The following example mounts the nginx-app/ directory into the container twice, and the second mount sets both the ro option and the rslave bind propagation option.

The --mount and -v examples have the same result:

$ docker run -itd –name nginx-dev \

--mount type=bind,source=”$(pwd)”/nginx-app,target=/app \

--mount type=bind,source=”$(pwd)”/nginx-app,target=/app2, \

readyonly,bind-propagation=rslave nginx:1.16

$ docker run -itd --name nginx-dev -v “$(pwd)”/nginx-app:/app \

-v “$(pwd)”/nginx-app:/app2:ro,rslave nginx:1.16

**Configure the selinux label:**

* Z or z options to modify the selinux label of the host file or directory being mounted into the container. This affects the file or directory on the host machine itself and can have consequences outside of the scope of Docker.
* The z option indicates that the bind mount content is shared among multiple containers.
* The Z option indicates that the bind mount content is private and unshared.
* Bind-mounting a system directory such as /home or /usr with the Z option renders your host machine inoperable and need manual relabeling.

$ docker run -itd --name nginx-dev -v “$(pwd)”/nginx-app:/app:z nginx:1.16

1. **Use tmpfs mounts:**

* When container created using “tmpfs” mount, the container create files outside the container’s writable layer.
* A tmpfs mount is temporary, and only persisted in the host memory.
* When the container stops, the tmpfs mount is removed, and files written there won’t be persisted.
* This is useful to temporarily store sensitive files that you don’t want to persist in either the host or the container writable layer.

**Limitations of tmpfs mounts:**

* Unlike volumes and bind mounts, you can’t share tmpfs mounts between containers.
* This functionality is only available for Docker on Linux.

**Choose --tmpfs or --mount:**

* **--tmpfs:** Mounts a tmpfs mount with no options, and can only be used with standalone containers, cannot be used with swarm services.
* **--mount:** Consists of multiple key-value pairs, separated by commas and each consisting of a <key>=<value> tuple. The --mount syntax is more verbose than --tmpfs:

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Key** | **Values / Options** | **Description** |
| 1 | type | **tmpfs**, bind, volume | Type of mount (ex: type=tmpfs) |
| 2 | destination / dst / target | Container path | path where the file or directory is mounted in the container |
| 3 | tmpfs-size |  | Size of the tmpfs mount in bytes. Unlimited by default. |
| 4 | tmpfs-mode |  | File mode of the tmpfs in octal. For instance, 700 or 0770. Defaults to 1777 or world-writable. |

**Examples:**

**Start a container (--tmpfs):**

$ docker run -d --name nginx-dev --tmpfs /app nginx:1.16

**Start a container with bind mount (--mount):**

The following example sets the tmpfs-mode to 1770, so that it is not world-readable within the container.

$ docker run -itd --name nginx-dev --mount type=tmpfs,target=/app,tmpfs-mode=770 nginx:1.16

**Docker storage drivers:**

Docker supports several different storage drivers, using a pluggable architecture. The storage driver controls how images and containers are stored and managed on your Docker host.

Docker supports the following storage drivers:

1. overlay or overlay2
2. aufs
3. devicemapper
4. btfrs
5. zfrs
6. **overlay or overlay2 (file based):**

* Linux Kernel version: >= 4.0
* RHEL / CentOS: >= 3.10
* Docker version: >= 17.06.02-ee5
* Storage Driver: overlay2, Backing Filesystem: xfs (Supports d\_type: true)
* Default and preferred storage driver, for all currently supported Linux distributions, and requires no extra

configuration.

* OverlayFS is a modern union filesystem that is similar to AUFS, but faster and with a simpler implementation. Docker provides two storage drivers for OverlayFS: the original overlay, and the newer and more stable overlay2.
* Linux kernel driver as OverlayFS and to the Docker storage driver as overlay or overlay2.
* Supported on **xfs** **backing** **filesystems**, but only with **d\_type=true** enabled.
* Changing the storage driver makes existing containers and images inaccessible on the local system. Use docker save to save any images you have built or push them to Docker Hub or a private registry
* It is highly recommended that you use the overlay2 driver if possible, rather than the overlay driver. The overlay driver is **not** supported for Docker EE.

**Steps to configure the overlay2 storage driver (same applicable for overlay):**

* 1. Stop docker

$ sudo systemctl stop docker

* 1. Copy the contents of /var/lib/docker to a temporary location.

$ sudo cp -au /var/lib/docker /var/lib/docker.bk

* 1. Edit /etc/docker/daemon.json. If it does not yet exist, create it. Assuming that the file was empty, add

the following contents.

{

"storage-driver": "overlay2"

}

* 1. Start docker

$ sudo systemctl start docker

* 1. Verify that the daemon is using the overlay2 storage driver and backing file system.

$ docker info

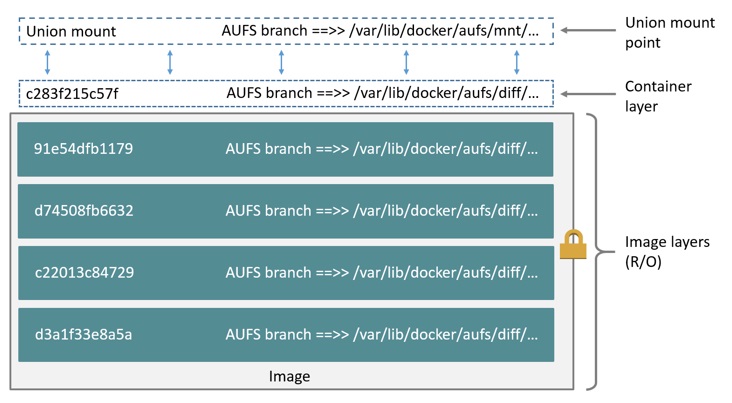
* OverlayFS layers two directories on a single Linux host and presents them as a single directory. These directories are called *layers* and the unification process is referred to as a ***union mount***. OverlayFS refers to the lower directory as **lowerdir** and the upper directory a **upperdir**. The unified view is exposed through its own directory called **merged**.



* The image layer is the lowerdir and the container layer is the upperdir. The unified view is exposed through a directory called merged which is effectively the containers mount point. The diagram shows how Docker constructs map to OverlayFS constructs.

1. **aufs (file based):**

* AUFS is a *union filesystem*, which means that it layers multiple directories on a single Linux host and presents them as a single directory. Tese directories are called *branches* in AUFS terminology, and *layers* in Docker terminology.
* Backing filesystem: **extfs**
* AUFS uses the Copy-on-Write (CoW) strategy to maximize storage efficiency and minimize overhead.
* AUFS is supported on Ubuntu, and on Debian versions prior to Stretch for Docker CE (EE only Ubuntu)
* Ubuntu 16.04 and higher supports overlay2.
* AUFS **cannot use** the following **backing** **filesystems**: **aufs, btrfs, or ecryptfs**. This means that the filesystem which contains /var/lib/docker/aufs cannot be one of these filesystem types.
* If you are using a different storage driver, either AUFS is not included in the kernel (in which case a different default driver is used) or that Docker has been explicitly configured to use a different driver. Check /etc/docker/daemon.json or the output of ps auxw | grep dockerd to see if Docker has been started with the --storage-driver flag.



* Each image layer, and the container layer, are represented on the Docker host as subdirectories within /var/lib/docker/. The union mount provides the unified view of all layers. The directory names do not directly correspond to the IDs of the layers themselves.
* If the AUFS driver is loaded into the kernel when you start Docker, and no other storage driver is configured, Docker uses it by default.

$ grep aufs /proc/filesystems

$ docker info

1. **devicemapper (block level storage):**

* Device Mapper is a kernel-based framework that underpins many advanced volume management technologies on Linux.
* Docker’s **devicemapper** storage driver leverages the thin provisioning and snapshotting capabilities of this framework for image and container management.
* The devicemapper driver uses block devices dedicated to Docker and operates at the block level, rather than the file level. These devices can be extended by adding physical storage to your Docker host, and they perform better than using a filesystem at the operating system (OS) level.
* Supports on Debian, Ubuntu, RHEL and CentOS.
* direct-lvm (production use) and loop-lvm (for testing purpose)
* Changing the storage driver makes existing containers and images inaccessible on the local system. Use docker save to save any images you have built or push them to Docker Hub or a private registry.
* Backing filesystem: **xfs**
* Refer more: <https://docs.docker.com/storage/storagedriver/device-mapper-driver/>

**Steps to configure Docker with the** devicemapper **storage driver:**

1. Stop docker

$ sudo systemctl stop docker

1. Copy the contents of /var/lib/docker to a temporary location.

$ sudo cp -au /var/lib/docker /var/lib/docker.bk

1. Edit /etc/docker/daemon.json. If it does not yet exist, create it. Assuming that the file was empty, add

the following contents.

{

"storage-driver": "devicemapper"

}

1. Start docker

$ sudo systemctl start docker

1. Verify that the daemon is using the devicemapper storage driver and backing file system.

$ docker info

1. **Btfrs (block level storage):**

* Btrfs is a next generation copy-on-write filesystem that supports many advanced storage technologies that make it a good fit for Docker.
* Can be easily combined multiple physical block devices into a single Btrfs filesystem.
* btrfs is only recommended on Ubuntu or Debian.
* btrfs requires a dedicated block storage device such as a physical disk. This block device must be formatted for Btrfs and mounted into /var/lib/docker/
* To manage BTRFS filesystems at the level of the operating system, you need the btrfs command. If you do not have this command, install the btrfsprogs package (SLES) or btrfs-tools package (Ubuntu).
* When space gets low, Btrfs automatically expands the volume in *chunks* of roughly 1 GB. To add a block device to a Btrfs volume, use the btrfs device add andbtrfs filesystem balance commands.
* Refer for more info: <https://docs.docker.com/storage/storagedriver/btrfs-driver/>

$ sudo btrfs device add /dev/svdh /var/lib/docker

$ sudo btrfs filesystem balance /var/lib/docker

**Steps to configure Docker with the** **btrfs** **storage driver:**

1. Stop docker

$ sudo systemctl stop docker

1. Copy the contents of /var/lib/docker to backup location and remove the contents.

$ sudo cp -au /var/lib/docker /var/docker.bk

$ sudo rm -rf /var/lib/docker/\*

1. Format your dedicated block device or devices as a Btrfs filesystem.

$ sudo mkfs.btrfs -f /dev/xvdf

1. Mount the new Btrfs filesystem on the /var/lib/docker/ mount point.

$ sudo mount -t btrfs /dev/xvdf /var/lib/docker

Make above change permanent by making an entry in /etc/fstab

1. Copy the contents from backup to /var/lib/docker/.

$ sudo cp -au /var/docker.bk/\* /var/lib/docker/

1. Configure Docker to use the btrfs storage driver. This is required even though /var/lib/docker/ is now using a Btrfs filesystem.

$ sudo vim /etc/docker/daemon.json

{

"storage-driver": "btrfs"

}

1. Start docker

$ sudo systemctl start docker

1. Verify that the daemon is using the btrfs storage driver.

$ docker info

1. **ZFS (block level storage):**

* ZFS is a next generation filesystem that supports many advanced storage technologies such as volume management, snapshots, checksumming, compression and deduplication, replication and more.
* ZFS requires one or more dedicated block devices, preferably solid-state drives (SSDs).
* ZFS is only supported on Docker Engine - Community with Ubuntu 14.04 or higher, with the zfs package (16.04 and higher) or zfs-native and ubuntu-zfspackages (14.04) installed.
* For Ubuntu 14.04, you need to enable a supplemental package repositoryppa:zfs-native/stable
* ZFS is not supported on Docker EE or CS-Engine, or any other Linux platforms.
* The /var/lib/docker/ directory must be mounted on a ZFS-formatted filesystem.
* More at: <https://docs.docker.com/storage/storagedriver/zfs-driver/>

**Steps to configure Docker with the** **ZFS** **storage driver:**

1. Stop docker

$ sudo systemctl stop docker

1. Copy the contents of /var/lib/docker to backup location and remove the contents.

$ sudo cp -au /var/lib/docker /var/docker.bk

$ sudo rm -rf /var/lib/docker/\*

1. Create a new zpool on your dedicated block device or devices, and mount it into /var/lib/docker/.

$ sudo zpool create -f zpool-docker -m /var/lib/docker /dev/xvdf

1. The command creates the zpool and names it zpool-docker. The name is for display purposes only, and you can use a different name. Check that the pool was created and mounted correctly using zfs lis

$ sudo zfs list

NAME USED AVAIL REFER MOUNTPOINT

zpool-docker 55K 96.4G 19K /var/lib/docker

1. Configure Docker to use the zfs storage driver.

$ sudo vim /etc/docker/daemon.json

{

"storage-driver": "zfs"

}

1. Start docker

$ sudo systemctl start docker

1. Verify that the daemon is using the zfs storage driver.

$ docker info

* To increase the size of the zpool, add a dedicated block device to the Docker host, and then add it to the zpool using the zpool add command:

$ sudo zpool add zpool-docker /dev/xvdh