### What is Docker Container ?

* Runtime of an image
* Instead of modifying the file inside the image, the container locates it in writable layer and makes the changethere. Container get full RW access without having write access to the image
* Container is a thin writable layer on the top of read only image, every container gets its own file-system, process tree, network stack. It’s a run time execution environment.

### Docker Container Create

***Syntax***

*$ docker container create [option] Image [Command] [Arg..]*

*When you create a Docker container , it is assigned a universally unique identifier.[UUID]*

***Example***

*$ docker container create --name container\_name image\_name*

### *List Docker Container*

***Syntax***

$ docker container ls [OPTIONS] or *$ docker ps [options]*

***Options***

-a Show all containers (default shows just running)

-f Filter output based on conditions provided

--format Pretty-print containers using a Go template

-n Show n last created containers (includes all states) default -1

-l Show the latest created container (includes all states)

--no-trunc Don't truncate output

-q Only display container IDs

-s Display total file sizes

**Examples**

* # Display Only Running Container
  + $ docker container ls or $ docker ps
* # Display All Containers
  + $ docker contailer ls -a
* # Only display container IDs [q]
  + $ docker container ls -aq
* # Don't truncate output [--no-trunc]
  + $ docker container ls -aq –no-trunc
* # Show n last created containers [n] (includes all states)
  + $ docker container ls -n5
* Show the latest created container [l] (includes all states)
  + $ docker container ls -l

**Display Disk size of the container [s]**

$ docker container ls -as

Command displays two different on-disk-sizes for each container:

The “size” information shows the amount of data (on disk) that is used for the writable layer of each container

The “virtual size” is the total amount of disk-space used for the read-only image data used by the container and the writable layer..

**Container size on disk**

* To view the approximate size of a running container, you can use the docker ps -s command. Two different columns relate to size.
* size: the amount of data (on disk) that is used for the writable layer of each container.
* virtual size: the amount of data used for the read-only image data used by the container plus the container’s writable layer size. Multiple containers may share some or all read-only image data. Two containers started from the same image share 100% of the read-only data, while two containers with different images which have layers in common share those common layers. Therefore, you can’t just total the virtual sizes. This over-estimates the total disk usage by a potentially non-trivial amount.
* The total disk space used by all of the running containers on disk is some combination of each container’s size and the virtual size values. If multiple containers started from the same exact image, the total size on disk for these containers would be SUM (size of containers) plus one image size (virtual size- size).
* This also does not count the following additional ways a container can take up disk space:
* Disk space used for log files if you use the json-file logging driver. This can be non-trivial if your container generates a large amount of logging data and log rotation is not configured.
* Volumes and bind mounts used by the container.
* Disk space used for the container’s configuration files, which are typically small.
* Memory written to disk (if swapping is enabled).
* Checkpoints, if you’re using the experimental checkpoint/restore feature.

**$ docker contianer ls -s**

+---------------+---------------+--------------------+

| CONTAINER ID | IMAGE | SIZE |

+===============+===============+====================+

| 6ca0cef8db8d | nginx | 2B (virtual 183MB) |

| 3ab1a4d8dc5a | nginx | 5B (virtual 183MB) |

+---------------+---------------+--------------------+

* When starting a container, the image that the container is started from is mounted read-only (virtual).
* On top of that, a writable layer is mounted, in which any changes made to the container are written.
* So the Virtual size (183MB in the example) is used only once, regardless of how many containers are started from the same image - I can start 1 container or a thousand; no extra disk space is used.
* The "Size" (2B in the example) is unique per container though, so the total space used on disk is:
* 183MB + 5B + 2B
* Be aware that the size shown does not include all disk space used for a container.
* Things that are not included currently are;
  + volumes
  + swapping
  + checkpoints
  + disk space used for log-files generated by container

Keep in mind that docker ps --size may be an expensive command, taking more than a few minutes to complete. The same applies to container list API requests with size=1. It's better not to run it too often.

* **Alternative to docker ps --size**
* As "docker ps --size" produces heavy IO load on host, it is not feasable running such command every minute in a production environment. Therefore we have to do a workaround in order to get desired container size or to be more precise, the size of the RW-Layer with a low impact to systems perfomance.
* This approach gathers the "device name" of every container and then checks size of it using "df" command. Those "device names" are thin provisioned volumes that a mounted to / on each container. One problem still persists as this observed size also implies all the readonly-layers of underlying image. In order to address this we can simple check size of used container image and substract it from size of a device/thin\_volume.
* One should note that every image layer is realized as a kind of a lvm snapshot when using device mapper. Unfortunately I wasn't able to get my rhel system to print out those snapshots/layers. Otherwise we could simply collect sizes of "latest" snapshots. Would be great if someone could make things clear. However...
* After some tests, it seems that creation of a container always adds an overhead of approx. 40MiB (tested with containers based on Image "httpd:2.4.46-alpine"):
* docker run -d --name apache httpd:2.4.46-alpine // now get device name from docker inspect and look it up using df
* df -T -> 90MB whereas "Virtual Size" from "docker ps --size" states 50MB and a very small payload of 2Bytes -> mysterious overhead 40MB
* curl/download of a 100MB file within container
* df -T -> 190MB whereas "Virtual Size" from "docker ps --size" states 150MB and payload of 100MB -> overhead 40MB

**Filterering [-f] [--filter]**

The filtering flag (-f or --filter) format is a key=value pair. If there is more than one filter, then pass multiple flags (e.g. --filter "foo=bar" --filter "bif=baz")

* **Filter by name**
* The following filter matches all containers with a name containing ct\_j string
  + $ docker container ls -a -f “name=ct\_j”
* **Filer by status**
* The status filter matches containers by status. You can filter using created, restarting, running, removing, paused, exited and dead. For example, to filter for running containers:
  + $ docker container ls -a -f “status=running”
* **Filter by images**
* The ancestor filter matches containers based on its image or a descendant of it. The filter supports the following image representation:
  + $ docker container ls -af “ancestor=imageName:tag”
* **Filter by Volume**
  + $ docker ps --filter volume=remote-volume --format "table {{.ID}}\t{{.Mounts}}"
* **Filter by publish & expose**
  + $ docker ps --filter publish=80
  + $ docker ps --filter expose=8000-8080/tcp range
  + $ docker ps --filter publish=80/udp
* **Filter by health**
* Filters containers based on their healthcheck status. One of starting, healthy, unhealthy or none.
  + $ docker continer ls -af “health=healthy”
* **Filter by Network**
* The network filter shows only containers that are connected to a network with a given name or id.
* The following filter matches all containers that are connected to a network with a name containing net1
  + $ docker run -d --net=net1 --name=test1 ubuntu top
  + $ docker run -d --net=net2 --name=test2 ubuntu top
  + $ docker ps --filter network=net1

**Formating**

The formatting option (--format) pretty-prints container output using a Go template.

Valid placeholders for the Go template are listed below:

**PlaceHolder** **Description**

* .ID Container ID
* .Image Image ID
* .Command Quoted command
* .CreatedAt Time when the container was created.
* .RunningFor Elapsed time since the container was started.
* .Ports Exposed ports.
* .State Container status (for example; “created”, “running”, “exited”).
* .Status Container status with details about duration and health-status.
* .Size Container disk size.
* .Names Container names.
* .Labels All labels assigned to the container.
* .Label Value of a specific label for this container. For example '{{.Label "com.docker.swarm.cpu"}}'
* .Mounts Names of the volumes mounted in this container.
* .Networks Names of the networks attached to this container.

Examples

$ docker ps --format "table {{.ID}}\t{{.Labels}}"

$ docker ps --format "{{.ID}}: {{.Command}}"

### *Docker Container Run*

***Syntax***

*$ docker container run [OPTIONS] IMAGE [COMMAND] [ARG...]*

**# Run a command in a new container**

Options

1. --add-host Add a custom host-to-IP mapping (host:ip)
2. --attach , -a Attach to STDIN, STDOUT or STDERR
3. --detach , -d Run container in background and print container ID
4. --detach-keys Override the key sequence for detaching a container
5. –device Add a host device to the container
6. --dns Set custom DNS servers
7. --entrypoint Overwrite the default ENTRYPOINT of the image
8. --env , -e S et environment variables
9. --expose Expose a port or a range of ports
10. --health-cmd Command to run to check health
11. --health-interval Time between running the check (ms|s|m|h) (default 0s)
12. --health-retries Consecutive failures needed to report unhealthy
13. --health-timeout Maximum time to allow one check to run (ms|s|m|h) (default 0s)
14. --hostname , -h Container host name
15. --interactive , -i Keep STDIN open even if not attached
16. --ip IPv4 address (e.g., 172.30.100.104)
17. --label , -l Set meta data on a container
18. --link Add link to another container
19. --log-driver Logging driver for the container
20. --log-opt Log driver options
21. --memory , -m Memory limit
22. --memory-reservation Memory soft limit
23. --mount Attach a filesystem mount to the container
24. --name Assign a name to the container
25. --net Connect a container to a network
26. --no-healthcheck Disable any container-specified HEALTHCHECK
27. --pid PID namespace to use
28. --privileged Give extended privileges to this container
29. --publish , -p Publish a container's port(s) to the host
30. --publish-all , -P Publish all exposed ports to random ports
31. --read-only Mount the container's root filesystem as read only
32. --restart no Restart policy to apply when a container exits
33. --rm Automatically remove the container when it exits
34. --rm Automatically remove the container when it exits
35. --security-opt Security Options
36. --stop-signal Signal to stop a container SIGTERM
37. --stop-timeout Timeout (in seconds) to stop a container
38. --tmpfs Mount a tmpfs directory
39. --tty , -t Allocate a pseudo-TTY
40. --ulimit Ulimit options
41. --user , -u Username or UID (format: <name|uid>[:<group|gid>])
42. --volume , -v Bind mount a volume
43. --volume-driver Optional volume driver for the container
44. --volumes-from Mount volumes from the specified container(s)
45. --workdir , -w Working directory inside the container

**Examples**

***Assign name [--name]***

*$ docker container run --name <containeName> <imageName>*

***Attach [-a] & Deatch [-d] mode***

*$ docker container run --name ContainerName -p 8081:80 nginx #Def : Attach mode*

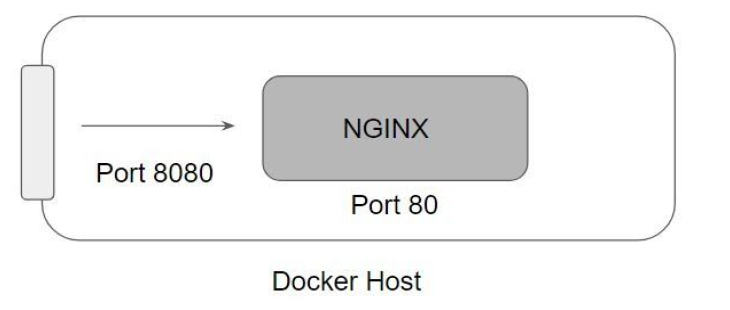
*$ docker container run -d --name ContainerName -p 8081:80 nginx*

***Port Binding [p]***

***Publish a container's port(s) to the host [p]***

*By default Docker containers can make connections to the outside world, but the outside world*

*cannot connect to containers. If we want containers to accept incoming connections from the world, you will have to bind it to a host port.*



*$ docker container run -dt --name ContainerName -p 8080:80 nginx*

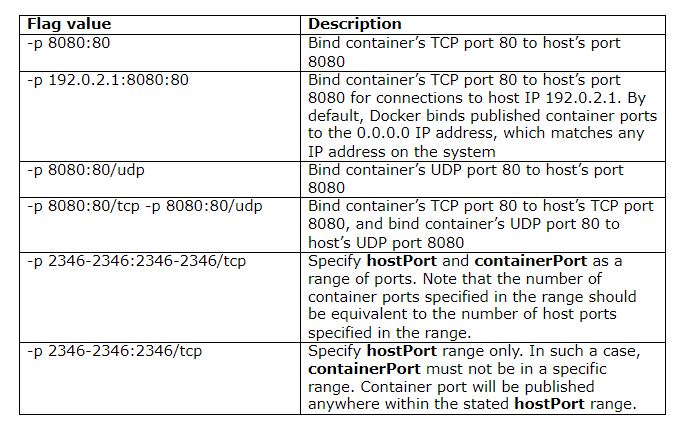
*By default, Docker exposes container ports to the IP address 0.0.0.0 (this matches any IP on the system).*

*$ docker container run -dt --name ContainerName -p 10.0.0.3:8080:80 nginx*

*To bind on IP address 10.0.0.3, host port 80, and container port 8080:*

*$ docker run -p 80:80/tcp -p 500:500/udp test*

*Host : Container/tcp*



***Port Binding***

***Publish all exposed ports to random ports [P]***

*$ docker container run -dt -P nginx*

*Here, Docker binds all exposed ports in the container to free random ports on the host.*

*while exposed ports can only be accessed internally, published ports can be accessible by external containers and services.*

***Each exposed port is mapped*** *automatically to a random port on the host interface. This automatic mapping also prevents potential port mapping conflicts*

***expose port [--expose)]***

*$ docker run --expose 80 ubuntu bash*

*This exposes port 80 of the container without publishing the port to the host system’s interfaces.*

***Container Restart policies [--restart]***

*By default, Docker containers will not start when they exit or when docker daemon is restarted.*

*Docker provides restart policies to control whether your containers start automatically when they*

*exit, or when Docker restarts.*

**Flag** **Description**

no Do not automatically restart the container [default]

on-failure Restart the container if it exits due to error , manifests as a not-zero exit code

unless-stopped Restart container unless it is explicitly stopped or docker itself stopped/restarted

always Always restart the container if it stops.

**Examples**

*$ docker container run -d --restart unless-stopped redis*

*$ docker container update --restart unless-stopped redis*

*$ docker container update --restart unless-stopped $(docker ps -q)*

***Detached | Background | Daeman mode [-d]***

*$ docker container run -d --name ContainerName -p 8081:80 nginx*

***Automatically delte container on exit [--rm]***

*By default, containers that are exited are not removed by Docker.*

*$ docker container run -dt --rm --name ContainerName busybox ping -c10 google.com*

***Interactive & Pseudo Terminal [i] [t]***

***IT flag***

*Every process that we create in the Linux environment, has three open file descriptors; stdin,*

*stdout, stderr. --interactive flag keeps stdin open even if not attached.&& --tty flag allocates a pseud*

*$ docker container run -it nginx bash*

*# Inside the container*

***Foreground | Attach mode [-a]***

*Docker run can start the process in the container and attach the console to the process’s standard input, output, and standard error. It can even pretend to be a TTY.*

*-a=[] : Attach to `STDIN`, `STDOUT` and/or `STDERR`*

*-t : Allocate a pseudo-tty*

*-i : Keep STDIN open even if not attached*

*If you do not specify -a then Docker will attach to both stdout and stderr . You can specify to which of the three standard streams (STDIN, STDOUT, STDERR) you’d like to connect instead, as in:*

*$ docker container run -a stdin -a stdout -i -t ubuntu /bin/bash*

*Note:  
Specifying -t is forbidden when the client is receiving its standard input from a pipe, as in*

*$ echo test | docker run -i busybox cat*

***Default Container Command***

*Whenever we run a container, a default command executes which typically runs as PID 1.*

*This command can be defined while we are defining the container image.*

*CMD ["nginx", "-g", "daemon off;"]*

*We can override the default container command by manually specifying the command.*

*$ docker container run -d nginx sleep 500*

*This container valid only for 500 sec as sleep is the PID 1*

***Mount Host Volumes***

*If you want to have persistent data that is stored even after the container stops, you need to enable sharing storage volumes.*

*For mounting volumes use the -v attribute with the specified location of the directory where you want to save the data, followed by where that data will be located inside the container.*

*$ docker container run -v [/host/volume/location]:[/container/storage] [docker\_image]*

***Add bind mounts or volumes using the --mount flag🔗***

*The --mount flag allows you to mount volumes, host-directories and tmpfs mounts in a container.*

*The --mount flag supports most options that are supported by the -v or the --volume flag, but uses a different syntax. For in-depth information on the --mount flag, and a comparison between --volume and --mount, refer to the service create command reference.*

*Even though there is no plan to deprecate --volume, usage of --mount is recommended.*

*$ docker run --read-only --mount type=volume,src=volumeName,target=/icanwrite busybox touch /icanwrite/here*

*$ docker run -t -i --mount type=bind,src=/data,dst=/data busybox sh*

***Capture container ID (--cidfile)***

***$*** *docker container run --cidfile /tmp/docker\_test.cid ubuntu echo "test"*

*This will create a container and print test to the console. The cidfile flag makes Docker attempt to create a new file and write the container ID to it. If the file exists already, Docker will return an error. Docker will close this file when docker run exits.*

***Full container capabilities (--privileged)***

*$ docker run -t -i --rm ubuntu bash*

*root:/# mount -t tmpfs none /mnt*

*mount: permission deined*

*This will not work, because by default, most potentially dangerous kernel capabilities are dropped; including cap\_sys\_admin (which is required to mount filesystems). However, the --privileged flag will allow it to run:*

*$ docker run -t -i --privileged ubuntu bash*

***Set working directory (-w)***

*$ docker run -w /path/to/dir/ -i -t ubuntu pwd*

*The -w lets the command being executed inside directory given, here /path/to/dir/. If the path does not exist it is created inside the container*

***Set storage driver options per container***

*$ docker run -it --storage-opt size=120G fedora /bin/bash*

*This (size) will allow to set the container rootfs size to 120G at creation time. This option is only available for the devicemapper, btrfs, overlay2, windowsfilter and zfs graph drivers. For the devicemapper, btrfs, windowsfilter and zfs graph drivers, user cannot pass a size less than the Default BaseFS Size. For the overlay2 storage driver, the size option is only available if the backing fs is xfs and mounted with the pquota mount option. Under these conditions, user can pass any size less than the backing fs size.*

***Mount tmpfs (--tmpfs)***

*$ docker run -d --tmpfs /run:rw,noexec,nosuid,size=65536k my\_image*

*The --tmpfs flag mounts an empty tmpfs into the container with the rw, noexec, nosuid, size=65536k options.*

***Set environment variables (-e, --env, --env-file)***

*$ docker run -e MYVAR1 --env MYVAR2=foo --env-file ./env.list ubuntu bash*

*Use the -e, --env, and --env-file flags to set simple (non-array) environment variables in the container you’re running, or overwrite variables that are defined in the Dockerfile of the image you’re running.*

*You can define the variable and its value when running the container:*

*$ docker run --env VAR1=value1 --env VAR2=value2 ubuntu env | grep VAR*

*VAR1=value1*

*VAR2=value2*

*You can also use variables that you’ve exported to your local environment:*

*export VAR1=value1*

*export VAR2=value2*

*$ docker run --env VAR1 --env VAR2 ubuntu env | grep VAR*

*VAR1=value1*

*VAR2=value2*

*You can also load the environment variables from a file. This file should use the syntax <variable>=value (which sets the variable to the given value) or <variable> (which takes the value from the local environment), and # for comments.*

*$ cat env.list*

*# This is a comment*

*VAR1=value1*

*VAR2=value2*

*USER*

*$ docker run --env-file env.list ubuntu env | grep VAR*

*VAR1=value1*

*VAR2=value2*

*USER=denis*

***Set metadata on container (-l, --label, --label-file)***

*label is a key=value pair that applies metadata to a container. To label a container with two labels:*

*$ docker run -l my-label --label com.example.foo=bar ubuntu bash*

*The my-label key doesn’t specify a value so the label defaults to an empty string (""). To add multiple labels, repeat the label flag (-l or --label).*

*The key=value must be unique to avoid overwriting the label value. If you specify labels with identical keys but different values, each subsequent value overwrites the previous. Docker uses the last key=value you supply.*

*Use the --label-file flag to load multiple labels from a file. Delimit each label in the file with an EOL mark. The example below loads labels from a labels file in the current directory:*

*$ docker run --label-file ./labels ubuntu bash*

*The label-file format is similar to the format for loading environment variables. (Unlike environment variables, labels are not visible to processes running inside a container.) The following example illustrates a label-file format:*

*com.example.lable1="a label1"*

*com.example.lable2="a label2"*

***Connect a container to a network (--network)***

*When you start a container use the --network flag to connect it to a network. This adds the busybox container to the my-net network.*

*$ docker run -itd --network=my-net busybox*

*You can also choose the IP addresses for the container with --ip and --ip6 flags when you start the container on a user-defined network.*

*$ docker run -itd --network=my-net --ip=10.10.9.75 busybox*

*If you want to add a running container to a network use the docker network connect subcommand.*

*You can connect multiple containers to the same network. Once connected, the containers can communicate easily using only another container’s IP address or name. For overlay networks or custom plugins that support multi-host connectivity, containers connected to the same multi-host network but launched from different Engines can also communicate in this way.*

*You can disconnect a container from a network using the docker network disconnect command.*

***Mount volumes from container (--volumes-from)***

*$ docker run --volumes-from 777f7dc92da7 --volumes-from ba8c0c54f0f2:ro -i -t ubuntu pwd*

*The --volumes-from flag mounts all the defined volumes from the referenced containers. Containers can be specified by repetitions of the --volumes-from argument. The container ID may be optionally suffixed with :ro or :rw to mount the volumes in read-only or read-write mode, respectively. By default, the volumes are mounted in the same mode (read write or read only) as the reference container.*

*Labeling systems like SELinux require that proper labels are placed on volume content mounted into a container. Without a label, the security system might prevent the processes running inside the container from using the content. By default, Docker does not change the labels set by the OS.*

*To change the label in the container context, you can add either of two suffixes :z or :Z to the volume mount. These suffixes tell Docker to relabel file objects on the shared volumes. The z option tells Docker that two containers share the volume content. As a result, Docker labels the content with a shared content label. Shared volume labels allow all containers to read/write content. The Z option tells Docker to label the content with a private unshared label. Only the current container can use a private volume.*

***Add host device to container (--device)***

*$ docker run --device=/dev/sdc:/dev/xvdc --device=/dev/sdd --device=/dev/zero:/dev/nulo -i -t ubuntu ls -l /dev/{xvdc,sdd,nulo}*

*It is often necessary to directly expose devices to a container. The --device option enables that. For example, a specific block storage device or loop device or audio device can be added to an otherwise unprivileged container (without the --privileged flag) and have the application directly access it.*

*By default, the container will be able to read, write and mknod these devices. This can be overridden using a third :rwm set of options to each --device flag. If the container is running in privileged mode, then the permissions specified will be ignored.*

***How to apply The Limits ??***

*$ free -f $ grep “model” /proc/cpuinfo Host mem & cpu info*

*$ docker container run -d –name nginx2 –memory “200mb” nginx:alpine*

*$ docker stat*

***Add entries to container hosts file (--add-host)***

*You can add other hosts into a container’s /etc/hosts file by using one or more --add-host flags. This example adds a static address for a host named docker:*

*$ docker run --add-host=docker:93.184.216.34 --rm -it alpine*

*/# ping docker*

*Sometimes you need to connect to the Docker host from within your container. To enable this, pass the Docker host’s IP address to the container using the --add-host flag. To find the host’s address, use the ip addr show command.*

*The flags you pass to ip addr show depend on whether you are using IPv4 or IPv6 networking in your containers. Use the following flags for IPv4 address retrieval for a network device named eth0:*

*$ HOSTIP=`ip -4 addr show scope global dev eth0 | grep inet | awk '{print $2}' | cut -d / -f 1 | sed -n 1p`*

*$ docker run --add-host=docker:${HOSTIP} --rm -it debian*

***How to apply The Limits ??***

*$ free -f $ grep “model” /proc/cpuinfo Host mem & cpu info*

*$ docker container run -d –name nginx2 –memory “200mb” nginx:alpine*

*$ docker stat*

***Stop container with timeout (--stop-timeout)***

*The --stop-timeout flag sets the number of seconds to wait for the container to stop after sending the pre-defined (see --stop-signal) system call signal. If the container does not exit after the timeout elapses, it is forcibly killed with a SIGKILL signal.*

*If --stop-timeout is set to -1, no timeout is applied, and the daemon will wait indefinitely for the container to exit.*

*The default is determined by the daemon, and is 10 seconds for Linux containers, and 30 seconds for Windows containers.*

*$ docker run --stop-timeout=30 --name=test test*

***How to apply The Limits ??***

*$ free -f $ grep “model” /proc/cpuinfo Host mem & cpu info*

*$ docker container run -d –name nginx2 –memory “200mb” nginx:alpine*

*$ docker stat*

***Memory Access***

*There are several RAM limitations you can set for a Docker container. Some of them include:*

*Configuring the maximum amount of memory a container can use.*

*Defining the amount of memory a Docker container can swap to disk.*

*Setting the soft limit for the amount of memory assigned to a container.*

*Below, find out how to configure Docker memory limitations.*

***Set Maximum Memory Access***

*To limit the maximum amount of memory usage for a container, add the --memory option to the docker run command.*

*$ sudo docker run -it --memory="[memory\_limit]" [docker\_image]*

*The value of memory\_limit should be a positive integer followed by the suffix b, k, m, or g (short for bytes, kilobytes, megabytes, or gigabytes). For example, to limit the container with 1 GB of RAM, add --memory="1g".*

***Set Swap to Disk Memory Limit***

*Using the swap option allows you to store data even after all RAM assigned to the container has been used up. It does this by ignoring the memory limitation and writing directly to the disk. Although this is a useful feature, it is not a recommended practice as it slows down performance.*

*To configure this additional RAM space, define the total amount of swap memory. Before doing this, you should already have the maximum memory (--memory) of the non-swap memory set. The swap includes the total amount of non-swap memory plus the amount of swap memory reserved as backup.*

*For example, if you set --memory to 1 GB, as in the example above, the amount of swap memory needs to be more than that. To run a container with an additional 1 GB of swap memory, set the swap memory to 2 GB.*

*The syntax for running a container with limited memory and additional swap memory is:*

*$ sudo docker run -it --memory="[memory\_limit]" --memory-swap="[memory\_limit]" [docker\_image]*

*$ sudo docker run -it --memory="1g" --memory-swap="2g" ubuntu*

*Note: If you don’t want to use swap memory, give --memory and --memory-swap the same values.*

***Limit Docker Container CPU Usage***

*Just like RAM usage, Docker containers don’t have any default limitations for the host’s CPU. Giving containers unlimited CPU usage can lead to issues.*

*There are several ways to define how much CPU resources from the host machine you want to assign to containers.*

*For example, if you have a host with 2 CPUs and want to give a container access to one of them, use the option --cpus="1.0". The command for running an Ubuntu container with access to 1 CPU would be:*

*sudo docker run -it --cpus="1.0" ubuntu*

*You can also use the --cpu-shares option to give the container a greater or lesser proportion of CPU cycles. By default, this is set to 1024.*

*To run a container with lesser CPU shares, run:*

*$ sudo docker run -it --cpus-shares="700" ubuntu*

***Set Soft Limit to Container Memory***

*Limiting the memory usage of a container with --memory is essentially setting a hard limit that cannot be surpassed. Alternatively, you can set a soft limit (--memory-reservation) which warns when the container reaches the end of its assigned memory but doesn’t stop any of its services.*

*If --memory limitations see are not set, setting the soft limit with --memory-reservation doesn’t completely limit container space. If you have both features enabled, the soft limit is always lower than the maximum space capacity.*

*As an example, for an Ubuntu container to have the memory reservation of 750 MB and the maximum RAM capacity of 1 BG, use the command:*

*$ sudo docker run -it --memory="1g" --memory-reservation="750m" ubuntu*

***How to apply The Limits ??***

*$ free -f $ grep “model” /proc/cpuinfo Host mem & cpu info*

*$ docker container run -d –name nginx2 –memory “200mb” nginx:alpine*

*$ docker stat*

### *Docker Container Exec*

***Run a command in a running container***

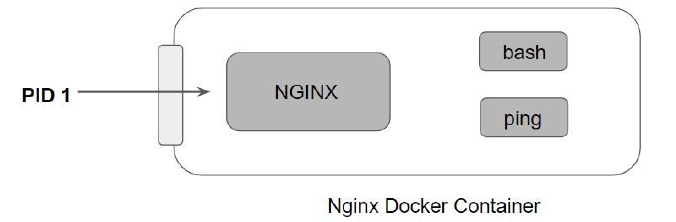
***Syntax***

*$ docker container exec [OPTIONS] CONTAINER COMMAND [ARG...]*

*The docker container exec command runs a new command in a running container. It allows you to create a new process in the container. The command started using docker exec only runs while the container’s primary process (PID 1) is running, and it is not restarted if the container is restarted.*

**Options**

* --detach , -d Detached mode: run command in the background
* --detach-keys Override the key sequence for detaching a container
* --env , -e Set environment variables
* --env-file Read in a file of environment variables
* --interactive , -i Keep STDIN open even if not attached
* --tty , -t Allocate a pseudo-TTY
* --user , -u Username or UID (format: <name|uid>[:<group|gid>])
* --workdir , -w Working directory inside the container



**Examples**

***Login & execute the commands***

*$ docker container exec -it containerName bash #/bin/bash or sh*

*# apt-get update*

*# apt-get install net-tools*

*# netstat -ntnlp*

*Note: bash should exit in the image it can be sh as well.*

***Execute the commands without login to the container***

*$ docker container exec -it containerName <command>*

*$ docker container exec -it containerName ls -lrt #touch #netstat -nlp #pwd*

*Here it flgs displays the output to the console*

***Execute the command in the background***

*$ docker container exec -d containerName ls -lrt #touch #netstat -nltnlp*

*Here -d run your commands backround doesn’t display the result in console*

***Working Directory to execute the commands [w]***

*By default docker exec command runs in the same working directory set when container was created.*

*$ docker exec -it ubuntu\_bash pwd*

*/*

*You can select working directory for the command to execute into*

*$ docker exec -it -w /root ubuntu /bin/bash -c “pwd”*

*/root*

***Execute the commands as other user [u]***

*$ docker container exec -it -u 0 -w /root containerName touch f1 # executed as ROOT user*

***Set an environment variable in the current bash session.***

*$ docker container exec -it -e var=”value” containerName command*

***Multiple commands***

*$ docker container exec -it containerName bash -c “command1 ; command 2; command3”*

*$ docker container exec -it f1 bash -c "cd /venkat; ls -lrt; ls"*

### Container Life Cycle Mgmt

There are different stages when we create a Docker container which is known as Docker Container Lifecycle. Some of the states are:

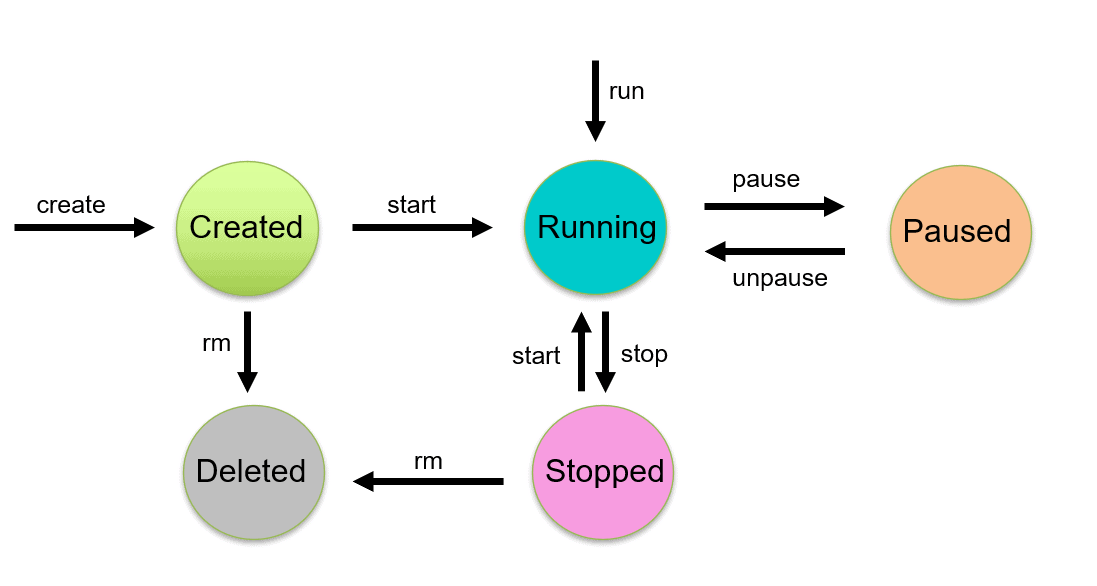
***Created****: A container that has been created but not started*

***Running****: A container running with all its processes*

***Paused****: A container whose processes have been paused*

***Stopped****: A container whose processes have been stopped*

***Deleted****: A container in a dead state*



***Docker container stop [Stop one or more running containers]***

*$ docker container stop <containerName> # -t=20 Seconds to wait for stop before killing it*

*Stop a running container (send SIGTERM, and then SIGKILL after grace period) [...] The main process inside the container will receive SIGTERM, and after a grace period, SIGKILL*

*When we use the docker stop command, the main process inside the container receives SIGTERM signal, and after some time, SIGKILL. Also, it will release the memory used after the container is stopped.*

***Docker Start [Start one or more stopped containers]***

*$ docker container start <containerName> # --interactive , -i*

***Docker kill [Kill one or more running containers]***

*$ docker container kill my\_container # -s=signal*

**Stop vs Kill**

* Kill a running container (send SIGKILL, or specified signal) [...] The main process inside the container will be sent SIGKILL, or any signal specified with option --signal.
* Even though these two commands seem to have the same result, there are some differences. The docker kill command stops the main process of a container abruptly while docker stop attempts to stop the container gracefully allowing the main process to cleanup before shutting down.
* And inspection of the container using docker inspect would shows that the Exit Code under container Status is 0 when the container was stopped using docker stop , while the same command run after docker kill shows an Exit Code of 137 (non 0 exit) in the container status..

***Docker pause [Pause all processes within one or more containers]***

***Docker unpause [Unpause all processes within one or more containers]***

*$ docker container pause <containerName> # -t=20 Seconds to wait for stop before killing it*

*$ docker container unpause <containerName>*

* The docker pause command suspends all processes in the specified containers. Traditionally, when suspending a process the SIGSTOP signal is used, which is observable by the process being suspended. Also, the memory portion would be there while the container is paused and again the memory is used when the container is resumed.
* the docker pause does not stop the container processes. Instead it suspends the container processes. In this state the container is not consuming CPU but would still keep its portion of the memory.
* After running the pause command you would notice that the container shows while running docker ps command. The docker top command would show the processes on the container.
* Now the docker stats output should show that CPU is not in use, but Memory is still consumed.

***Docker container remove [Remove one or more containers] [rm]***

*$ docker container rm <containerName>*

*$ docker container rm --force redis Force-remove a running container*

*$ docker rm -v redis Remove a container and its volumes*

*Note: It will not remove the named volume.*

*$ docker rm $(docker ps --filter status=exited -q) Remove all stopped containers*

*-f force the removal of running container*

*-l Remove the specified link*

*-v Remove anonymous volumes associated with the container*

**Example**

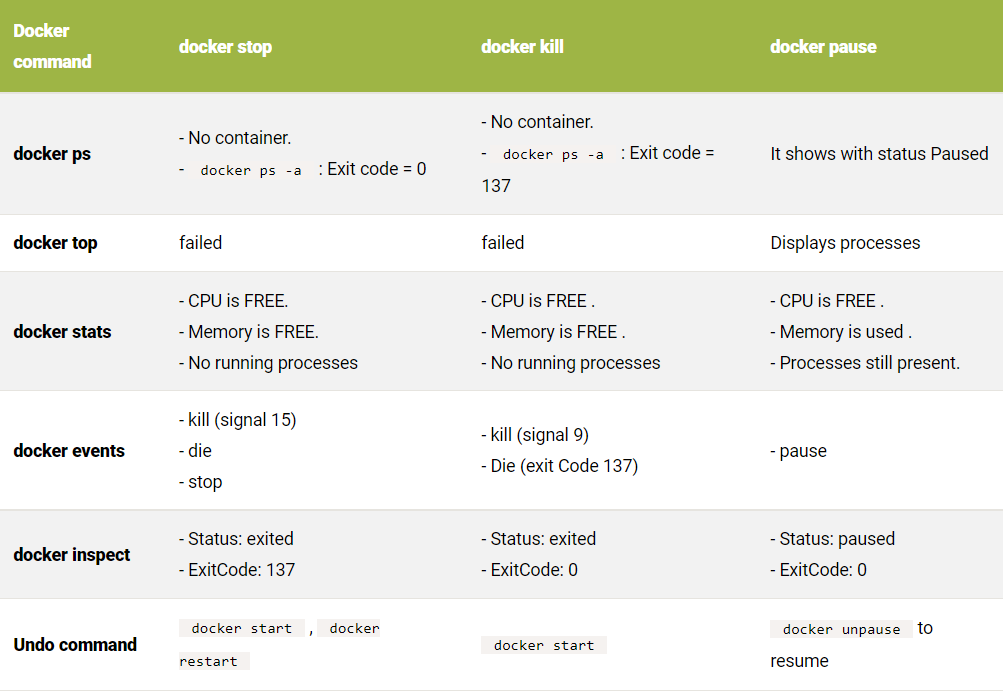
* $ docker rm --link /webapp/redis
* This removes the underlying link between /webapp and the /redis containers only on the **default bridge network**, removing all network communication between the two containers
* $ docker contianer rm $(docker ps --filter status=exited -q)

***Docker container rename [Rename a container]***

*$ docker container rename CONTAINER NEW\_NAME*

***#Restart one or more containers***

*$ docker container restart [OPTIONS] CONTAINER [CONTAINER...]*



### *Docker Container Prune*

Prune Container

When you stop a container, it is not automatically removed unless you started it with the --rm flag. To see all containers on the Docker host, including stopped containers, use docker ps -a. You may be surprised how many containers exist, especially on a development system! A stopped container’s writable layers still take up disk space. To clean this up, you can use the docker container prune command.

***Syntax***

*$ docker container prune [options] -f force*

*--filter*

*#Removes all stopped containers.*

**Filter options**

**until** (<timestamp>) only remove containers created before given timestamp

**label** (label=<key>, label=<key>=<value>, label!=<key>, or label!=<key>=<value>) - only remove containers with (or without, in case label!=... is used) the specified labels.

Examples

***The following removes containers created more than 5 minutes ago****:*

*$ docker ps -a --format 'table {{.ID}}\t{{.Image}}\t{{.Command}}\t{{.CreatedAt}}\t{{.Status}}'*

*$ docker container prune --force --filter "until=5m"*

***The following removes containers created before 2017-01-04T13:10:00:***

*$ docker ps -a --format 'table {{.ID}}\t{{.Image}}\t{{.Command}}\t{{.CreatedAt}}\t{{.Status}}'*

*$ docker container prune --force --filter "until=2017-01-04T13:10:00"*

### Docker Container cp

*Copy files/folders between a container and the local filesystem*

*Use ‘-‘ as the source to read a tar archive from stdin and extract it to a directory destination in a container. Use ‘-‘ as the destination to stream a tar archive of a container source to stdout.*

***Syntax***

*$ docker container cp [OPTIONS] CONTAINER:SRC\_PATH DEST\_PATH|-*

*--archive , -a Archive mode (copy all uid/gid information)*

*--follow-link , -L Always follow symbol link in SRC\_PATH*

*The docker cp command does have some limitations. First, we cannot use it to copy between two containers. It can only be used to copy files between the host system and a single container.*

*If /tmp does not exist, Docker will create it and copy the contents of /tmp/foo from the container into this new directory. If /tmp already exists as a directory, then Docker will copy the contents of /tmp/foo from the container into a directory at /tmp/foo.*

*When copying a single file to an existing LOCALPATH, the docker cp command will either overwrite the contents of LOCALPATH if it is a file or place it into LOCALPATH if it is a directory, overwriting an existing file of the same name if one exists.*

***Example***

* ***Host to container***
* *$ docker container cp /tmp/config.ini grafana:/usr/share/grafana/conf/*
* ***Container to Host***
* *$ docker cp grafana:/usr/share/grafana/conf/defaults.ini /tmp*

### Docker Container commit

*Create a new image from a container’s changes*

***Syntax***

*$ docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]*

*--author , -a Author (e.g., "John Hannibal Smith <hannibal@a-team.com>")*

*--change , -c Apply Dockerfile instruction to the created image*

*--message , -m Commit message*

*--pause , -p true Pause container during commit*

***Example***

*$ docker container commit c3f279d17e0a svendowideit/testimage:version3*

*$ docker commit --change "ENV DEBUG=true" c3f279d17e0a svendowideit/testimage:version3*

*$ docker commit --change='CMD ["apachectl", "-DFOREGROUND"]' -c "EXPOSE 80" c3f279d17e0a svendowideit/testimage:version4*

### Docker Container diff

***Syntax***

*$ docker container diff CONTAINER*

*Inspect changes to files or directories on a container’s filesystem*

*List the changed files and directories in a container᾿s filesystem since the container was created. Three different types of change are tracked:*

*Symbol Description*

*A A file or directory was added*

*D A file or directory was deleted*

*C A file or directory was changed*

***Example***

*Inspect the changes to an nginx container:*

*$ docker diff 1fdfd1f54c1b*

### Docker Container export

*Export a container’s filesystem as a tar archive*

***Syntax***

*$ docker container export [OPTIONS] CONTAINER*

*--output , -o Write to a file, instead of STDOUT*

*$ docker export {Container Name} > {filename} or*

*$ docker export -o={filename} {Container Name}*

$ docker export red\_panda > latest.tar

$ docker export --output="latest.tar" red\_panda

*It takes at least one argument that is a container name or container ID. When we run this command it actually saves the container’s file system as an archive. It creates a flat Docker image that makes the Docker image slightly smaller in size, however, it lost its history and metadata which means we cannot perform any rollback to a previous layer if we import a Docker image using any exported tar file.*

### Docker Container inspect

*Display detailed information on one or more containers*

***Syntax****$ docker container inspect [OPTIONS] CONTAINER [CONTAINER...]*

*--format , -f Format the output using the given Go template*

*--size , -s Display total file sizes*

*Docker inspect provides detailed information on constructs controlled by Docker.*

*By default, docker inspect will render results in a JSON array.*

***Example***

* ***Get an instance’s IP address***

*For the most part, you can pick out any field from the JSON in a fairly straightforward manner.*

* *$ docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' $INSTANCE\_ID*
* ***Get an instance’s MAC address***
* *$ docker inspect --format='{{range .NetworkSettings.Networks}}{{.MacAddress}}{{end}}' $INSTANCE\_ID*
* ***Get an instance’s log path***
* *$ docker inspect --format='{{.LogPath}}' $INSTANCE\_ID*
* ***Get an instance’s image name***
* *$ docker inspect --format='{{.Config.Image}}' $INSTANCE\_ID*
* ***List all port bindings***
* *You can loop over arrays and maps in the results to produce simple text output:*
* *$ docker inspect --format='{{range $p, $conf := .NetworkSettings.Ports}} {{$p}} -> {{(index $conf 0).HostPort}} {{end}}' $INSTANCE\_ID*

***Find a specific port mapping***

*The .Field syntax doesn’t work when the field name begins with a number, but the template language’s index function does. The .NetworkSettings.Ports section contains a map of the internal port mappings to a list of external address/port objects. To grab just the numeric public port, you use index to find the specific port map, and then index 0 contains the first object inside of that. Then we ask for the HostPort field to get the public address.*

*$ docker inspect --format='{{(index (index .NetworkSettings.Ports "8787/tcp") 0).HostPort}}' $INSTANCE\_ID*

### Docker Container logs

*Fetch the logs of a container*

*This command is only functional for containers that are started with the json-file or journald logging driver.*

***Syntax***

*$ docker logs [OPTIONS] CONTAINER*

* *--details Show extra details provided to logs*
* *--follow , -f Follow log output*
* *--tail , -n all Number of lines to show from the end of the logs*
* *--timestamps , -t Show timestamps*
* *--until Show logs before a timestamp (e.g. 2013-01-02T13:23:37Z) or relative (e.g. 42m for 42 minutes)*
* *--since Show logs since timestamp (e.g. 2013-01-02T13:23:37Z) or relative (e.g. 42m for 42 minutes)*

***/var/lib/docker/containers/<container\_id>/<container\_id>-json.log***

***Examples***

***Retrieve logs until a specific point in time***

*In order to retrieve logs before a specific point in time, run:*

*The docker logs --follow command will continue streaming the new output from the container’s STDOUT and STDERR.*

*$ docker run --name test -d busybox sh -c "while true; do $(echo date); sleep 1; done"*

*$ date*

* *Tue 14 Nov 2017 16:40:00 CET*

*$ docker logs -f --until=2s test*

* *Tue 14 Nov 2017 16:40:00 CET*
* *Tue 14 Nov 2017 16:40:01 CET*
* *Tue 14 Nov 2017 16:40:02 CET*

*$ docker logs <container ID>*

* *Although this will show us the logs, it won’t allow us to view continuous log output. In Docker jargon, we refer to creating a continuous stream of log output as tailing logs. To tail the logs for our container, we can use the follow option.*
* *docker logs --follow <container ID>*

*#1: Display Only the Latest Lines*

* *docker logs --tail 100 <container ID>*

*#2: Stream Logs Until a Specific Point in Time*

* *docker logs --follow --until=30m*

*#3: Stream Logs From a Specific Point in Time*

* *docker logs --since 2019-03-02 <container ID>*

***4 Best Practices When Logging in Docker***

* ***Export Logs to Persistent Storage***
* *You can easily create and destroy containers. However, every time a container restarts, you lose all the data it holds. Therefore, never store application-specific data in your container.*
* *For the same reason, you should take good care of your logs. Logs can be stored persistently to a volume, but it’s even better to store them long-term. For example, you can pipe logs to a local hard drive or you can send them to a log management platform. Both options allow you to save your logs long-term and use them for future analysis.*
* ***Consider Using a Logging Container***
* *A logging container helps you scale your logging. The idea is that you pipe your logging output from multiple containers to a logging container. Next, your logging container takes care of saving logs to persistent storage or a log management platform.*
* *Also, you can spin up multiple logging containers to scale your logging service when you decide to host more containers. It’s a flexible and easy solution for handling log output.*
* ***Log Data to Standard Text: Output Channels***
* *To be able to aggregate logs from your containers, you need to make sure the applications running in those containers log data to STDOUT or STDERR, both standard channels for logging output messages or error messages. Docker is configured to automatically pick up data from both outputs. If you log data to a file inside your container, you risk losing all this data when the container crashes or restarts. Therefore, if you don’t want to lose important logging data, it’s important to log to STDOUT or STDERR.*
* *Log Data as JSON Format*
* *Docker supports the JSON logging format, and logging data in this format is recommended. Docker itself stores logs as JSON files; therefore, it’s optimized to handle JSON data.*
* *For this reason, many Node.js logging libraries such as Bunyan or Winston prefer to log data using the JSON format.*

***Where Are Docker Container Logs Stored by Default?***

*You see, by default, Docker containers emit logs to the stdout and stderr output streams. Containers are stateless, and the logs are stored on the Docker host in JSON files by default.*

***The default logging driver is json-file.***

***What’s a logging driver?***

*A logging driver is a mechanism for getting info from your running containers. Here’s a more elaborate explanation from the Docker docs. There are several different log drivers you can use except for the default json-file, like syslog, journald, fluentd, or logagent.*

***How to find the logs?***

*These logs are emitted from output streams, annotated with the log origin, either stdout or stderr, and a timestamp. Each log file contains information about only one container and is in JSON format. Remember, one log file per container.*

*You find these JSON log files in the /var/lib/docker/containers/ directory on a Linux Docker host. The <container\_id> here is the id of the running container.*

*/var/lib/docker/containers/<container\_id>/<container\_id>-json.log*

*You collect the logs with a log aggregator and store them in a place where they’ll be available forever. It’s dangerous to keep logs on the Docker host because they can build up over time and eat into your disk space. That’s why you should use a central location for your Docker logs and enable log rotation for your containers.*

***Debugging Docker Issues with Container Logs***

* *Docker has a dedicated API for working with logs. But, keep in mind, it will only work if you use the json-file log driver. I strongly recommend not changing the log driver! Let’s start debugging.*
* *First of all, to list all running containers, use the docker ps command.*

*$ docker ps*

* *Then, with the docker logs command you can list the logs for a particular container.*

*$ docker logs <container\_id>*

* *Most of the time you’ll end up tailing these logs in real time, or checking the last few logs lines.*
* *Using the --follow or -f flag will tail -f (follow) the Docker container logs:*

*$ docker logs <container\_id> -f*

* *The --tail flag will show the last number of log lines you specify:*

*$ docker logs <container\_id> --tail N*

* *The -t or --timestamp flag will show the timestamps of the log lines:*

*$ docker logs <container\_id> -t*

* *The --details flag will show extra details about the log lines:*

*$ docker logs <container\_id> --details*

* *But what if you only want to see specific logs? Luckily, grep works with Docker logs as well.*

*$ docker logs <container\_id> | grep pattern*

***Storing Docker Container Logs in a Central Location Using a Log Shipper***

* *With your infrastructure growing, you can rely on just using the Docker API to troubleshoot logs. You need to store all logs in a secure place, so you can analyze and troubleshoot any issues after-the-fact.*
* *By storing logs in one place you can also set up alerts that notify you if anything breaks, or whenever you’re experiencing unexpected behavior.*
* *Container logs can be a mix of plain text messages from start scripts and structured logs from applications, which makes it difficult for you to tell which log event belongs to what container and application.*
* *Although Docker log drivers can ship logs to log management tools, most of them don’t allow you to parse container logs. You need a separate tool called a log shipper, such as Logagent, Logstash or rsyslog to structure and enrich the logs before shipping them.*
* *The solution is to have a container dedicated solely to logging and collecting logs. You deploy the dedicated logging container within your Docker environment. It will automatically aggregate logs from all containers, as well as monitor, analyze, and store or forward them to a central location.*
* *This makes it easier to move containers between hosts and easily scale your infrastructure. It also lets you collect logs through various streams, including log events, Docker API data, stats, etc.*
* *By far the most reliable and convenient way of log collection is to use the json-file driver and set up a log shipper to ship the logs. You always have a local copy of logs on your server and you get the advantage of centralized log management.*

*This is one of the biggest challenges to Docker logging. However basic your Docker installation is, you will have to work with two levels of aggregation. One refers to the logs from the Dockerized application inside the container. The other involves the logs from the host servers, which consist of the system logs, as well as the Docker Daemon logs which are usually located in /var/log or a subdirectory within this directory.*

### Docker Container port

List port mappings or a specific mapping for the container

***Syntax***

*$ docker container port CONTAINER [PRIVATE\_PORT[/PROTO]]*

***Example***

* *$ docker port test*

*7890/tcp -> 0.0.0.0:4321*

*9876/tcp -> 0.0.0.0:1234*

* *$ docker port test 7890/tcp*

*0.0.0.0:4321*

* *$ docker port test 7890/udp*

*2014/06/24 11:53:36 Error: No public port '7890/udp' published for test*

* *$ docker port test 7890*

*0.0.0.0:4321*

### Docker Container stat

*Display a live stream of container(s) resource usage statistics*

***Syntax***

*$ docker container stats [OPTIONS] [CONTAINER...]*

*Options*

* *--all , -a Show all containers (default shows just running)*
* *--format Pretty-print images using a Go template*
* *--no-stream Disable streaming stats and only pull the first result*
* *--no-trunc Do not truncate output*

*Example*

***Running docker stats on all running containers against a Linux daemon.***

*$ docker container stats*

***CONTAINER ID NAME CPU % MEM USAGE / LIMIT MEM % NET I/O BLOCK I/O PIDS***

*b95a83497c91 awesome\_brattain 0.28% 5.629MiB / 1.952GiB 0.28% 916B / 0B 147kB / 0B 9*

*67b2525d8ad1 foobar 0.00% 1.727MiB / 1.952GiB 0.09% 2.48kB / 0B 4.11MB / 0B 2*

*e5c383697914 test 0.00% 196KiB / 1.952GiB 0.01% 71.2kB / 0B 770kB / 0B 1*

*4bda148efbc0 random 0.00% 1.672MiB / 1.952GiB 0.08% 110kB / 0B 578kB / 0B 2*

***Formating***

* *.Container Container name or ID (user input)*
* *.Name Container name*
* *.ID Container ID*
* *.CPUPerc CPU percentage*
* *.MemUsage Memory usage*
* *.NetIO Network IO*
* *.BlockIO Block IO*
* *.MemPerc Memory percentage (Not available on Windows)*
* *.PIDs Number of PIDs (Not available on Windows)*

***Running docker stats with customized format on all (Running and Stopped) containers.***

*$ docker stats --all --format "table {{.Container}}\t{{.CPUPerc}}\t{{.MemUsage}}" fervent\_panini 5acfcb1b4fd1 drunk\_visvesvaraya big\_heisenberg*

***To list all containers statistics with their name, CPU percentage and memory usage in a table format you can use:***

*$ docker stats --format "table {{.Container}}\t{{.CPUPerc}}\t{{.MemUsage}}"*

### Docker Container top

*Display the running processes of a container*

*ps-OPTION can be any of the options you would pass to a Linux ps command.*

***Syntax***

*$ docker container top CONTAINER [ps OPTIONS]*

### Docker Container update

*Update configuration of one or more containers*

***Syntax***

*$ docker container update [OPTIONS] CONTAINER [CONTAINER...]*

*--blkio-weight Block IO (relative weight), between 10 and 1000, or 0 to disable (default 0)*

*--cpu-period Limit CPU CFS (Completely Fair Scheduler) period*

*--cpu-quota Limit CPU CFS (Completely Fair Scheduler) quota*

*--cpu-rt-period Limit the CPU real-time period in microseconds*

*--cpu-rt-runtime Limit the CPU real-time runtime in microseconds*

*--cpu-shares , -c CPU shares (relative weight)*

*--cpus Number of CPUs*

*--cpuset-cpus CPUs in which to allow execution (0-3, 0,1)*

*--cpuset-mems MEMs in which to allow execution (0-3, 0,1)*

*--kernel-memory Kernel memory limit*

*--memory , -m Memory limit*

*--memory-reservation Memory soft limit*

*--memory-swap Swap limit equal to memory plus swap: '-1' to enable unlimited swap*

*--pids-limit Tune container pids limit (set -1 for unlimited)*

*--restart Restart policy to apply when a container exits*

***Example***

***Update a container’s cpu-shares***

*To limit a container’s cpu-shares to 512, first identify the container name or ID. You can use docker ps to find these values. You can also use the ID returned from the docker run command. Then, do the following:*

*$ docker update --cpu-shares 512 abebf7571666*

***Update a container with cpu-shares and memory***

*To update multiple resource configurations for multiple containers:*

*$ docker update --cpu-shares 512 -m 300M abebf7571666 hopeful\_morse*

***Update a container’s restart policy***

*You can change a container’s restart policy on a running container. The new restart policy takes effect instantly after you run docker update on a container.*

*To update restart policy for one or more containers:*

*$ docker update --restart=on-failure:3 abebf7571666 hopeful\_morse*

*Note that if the container is started with “--rm” flag, you cannot update the restart policy for it. The AutoRemove and RestartPolicy are mutually exclusive for the container.*

### Docker Container wait

*Block until one or more containers stop, then print their exit codes*

***Syntax***

*$ docker container wait CONTAINER [CONTAINER...]*

***Example***

* *Start a container in the background.*
* *$ docker run -dit --name=my\_container ubuntu bash*
* *Run docker wait, which should block until the container exits.*
* *$ docker wait my\_container*
* *In another terminal, stop the first container. The docker wait command above returns the exit code.*
* *$ docker stop my\_container*
* *This is the same docker wait command from above, but it now exits, returning 0.*
* *$ docker wait my\_container*
* *0*

### Change Docker Root directory

*$ docker info | grep -i root # Shows Docker Root Directory*

*Docker root Dir : /var/lib/docker*

* *Migrate Deafult docker location to other filesystem*
* *$ Sudo systemctl stop docker*
* *Next, you need to update the Docker unit file.*
* *sudo nano /lib/systemd/system/docker.service*
* *Find the following line:*
* *ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock*
* *And new directory :*
* *ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock --data-root /mnt/docker*
* *#Reload Docker Daemon*
* *$ systemctl daemon-reload*
* *$ sysemctl restart docker*
* *# New Directory Mounted*
* *$ docker info | grep -i root*
* *# Copy all data from /var/lib/docker to /mnt/docker*