**1 Installing Docker on --**

**CentOS7**

first, let’s update the package database:

* sudo yum check-update

Now run this command. It will add the official Docker repository, download the latest version of Docker, and install it:

* curl -fsSL https://get.docker.com/ | sh

After installation has completed, start the Docker daemon:

* sudo systemctl start docker

Verify that it’s running:

* sudo systemctl status docker

The output should be similar to the following, showing that the service is active and running:

Output

● docker.service - Docker Application Container Engine

Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)

Active: active (running) since Sun 2016-05-01 06:53:52 CDT; 1 weeks 3 days ago

Docs: https://docs.docker.com

Main PID: 749 (docker)

Lastly, make sure it starts at every server reboot:

* sudo systemctl enable docker

**Without Sudo –**

If you want to avoid typing sudo whenever you run the docker command, add your username to the docker group:

* sudo usermod -aG docker $(whoami)
* You will need to log out of the Droplet and back in as the same user to enable this change.

If you need to add a user to the docker group that you’re not logged in as, declare that username explicitly using:

* sudo usermod -aG docker username

**Ubuntu**

First, update your existing list of packages:

* sudo apt update

Next, install a few prerequisite packages which let apt use packages over HTTPS:

* sudo apt install apt-transport-https ca-certificates curl software-properties-common

Then add the GPG key for the official Docker repository to your system:

* curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

Add the Docker repository to APT sources:

* sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic stable"

Next, update the package database with the Docker packages from the newly added repo:

* sudo apt update

Make sure you are about to install from the Docker repo instead of the default Ubuntu repo:

* apt-cache policy docker-ce

You’ll see output like this, although the version number for Docker may be different:

Output of apt-cache policy docker-ce

docker-ce:

Installed: (none)

Candidate: 18.03.1~ce~3-0~ubuntu

Version table:

18.03.1~ce~3-0~ubuntu 500

500 https://download.docker.com/linux/ubuntu bionic/stable amd64 Packages

Notice that docker-ce is not installed, but the candidate for installation is from the Docker repository for Ubuntu 18.04 (bionic).

Finally, install Docker:

* sudo apt install docker-ce

Docker should now be installed, the daemon started, and the process enabled to start on boot. Check that it’s running:

* sudo systemctl status docker

The output should be similar to the following, showing that the service is active and running:

Output

● docker.service - Docker Application Container Engine

Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)

Active: active (running) since Thu 2018-07-05 15:08:39 UTC; 2min 55s ago

Docs: https://docs.docker.com

Main PID: 10096 (dockerd)

Tasks: 16

CGroup: /system.slice/docker.service

├─10096 /usr/bin/dockerd -H fd://

└─10113 docker-containerd --config /var/run/docker/containerd/containerd.toml

**Without Sudo**

If you want to avoid typing sudo whenever you run the docker command, add your username to the docker group:

* sudo usermod -aG docker ${USER}

To apply the new group membership, log out of the server and back in, or type the following:

* su - ${USER}

You will be prompted to enter your user’s password to continue.

Confirm that your user is now added to the docker group by typing:

* id -nG

Output

sammy sudo docker

If you need to add a user to the docker group that you’re not logged in as, declare that username explicitly using:

* sudo usermod -aG docker username

**2 Storage Drivers**

The **storage driver** controls how images and containers are stored and managed on your **Docker** host.

Docker supports the following storage drivers:

* overlay2 is the preferred storage driver, for all currently supported Linux distributions, and requires no extra configuration.
* aufs is the preferred storage driver for Docker 18.06 and older, when running on Ubuntu 14.04 on kernel 3.13 which has no support for overlay2.
* devicemapper is supported, but requires direct-lvm for production environments, because loopback-lvm, while zero-configuration, has very poor performance. devicemapper was the recommended storage driver for CentOS and RHEL, as their kernel version did not support overlay2. However, current versions of CentOS and RHEL now have support for overlay2, which is now the recommended driver.
* The btrfs and zfs storage drivers are used if they are the backing filesystem (the filesystem of the host on which Docker is installed). These filesystems allow for advanced options, such as creating “snapshots”, but require more maintenance and setup. Each of these relies on the backing filesystem being configured correctly.
* The vfs storage driver is intended for testing purposes, and for situations where no copy-on-write filesystem can be used. Performance of this storage driver is poor, and is not generally recommended for production use.

| **Linux distribution** | **Recommended storage drivers** | **Alternative drivers** |
| --- | --- | --- |
| Docker Engine - Community on Ubuntu | overlay2 or aufs (for Ubuntu 14.04 running on kernel 3.13) | overlay¹, devicemapper², zfs, vfs |
| Docker Engine - Community on Debian | overlay2 (Debian Stretch), aufs or devicemapper (older versions) | overlay¹, vfs |
| Docker Engine - Community on CentOS | overlay2 | overlay¹, devicemapper², zfs, vfs |
| Docker Engine - Community on Fedora | overlay2 | overlay¹, devicemapper², zfs, vfs |

¹) The overlay storage driver is deprecated in Docker Engine - Enterprise 18.09, and will be removed in a future release. It is recommended that users of the overlay storage driver migrate to overlay2.

²) The devicemapp storage driver is deprecated in Docker Engine 18.09, and will be removed in a future release. It is recommended that users of the devicemapper storage driver migrate to overlay2.

When possible, overlay2 is the recommended storage driver. When installing Docker for the first time, overlay2 is used by default. Previously, aufs was used by default when available, but this is no longer the case. If you want to use aufs on new installations going forward, you need to explicitly configure it, and you may need to install extra packages, such as linux-image-extra. See [aufs](https://docs.docker.com/storage/storagedriver/aufs-driver/).

On existing installations using aufs, it is still used.

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.

**3 Running a Container**

docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

sudo docker run -i -t --name myubuntuc2 ubuntumynewimage

sudo docker run -i -t --name mytomcat tomcat:8.0

sudo docker run -i -t -p 8888:8080 --name mytomcat tomcat:8.0 running the container in foreground

docker pull tomcat:8.0

sudo docker run -i -t -d -p 8888:8080 --name mytomcat tomcat:8.0 running the container in background

sudo docker run -i -t -p 8888:8080 --name mywebapp myapplicationimage:1.0

Working in interatctive mode (-it)

$ docker run --name test -it debian

This example runs a container named test using the debian:latest image. The -it instructs Docker to allocate a pseudo-TTY connected to the container’s stdin; creating an interactive bash shell in the container.

### 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.

### 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

### Mount volumes from container (--volumes-from)

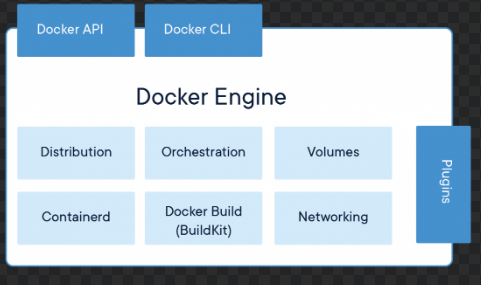
$ docker run --volumes-from 777f7dc92da7 --volumes-from ba8c0c54f0f2:ro -i -t ubuntu pwd

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

**4 What is Docker engine?**

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

**Docker Engine** supports the tasks and workflows involved to build, ship and run container-based applications. The **engine** creates a server-side daemon process that hosts images, containers, networks and storage volumes.



## **Create a backup**

Before upgrading Docker Engine - Enterprise, you should make sure you [create a backup](https://docs.docker.com/ee/backup/). This makes it possible to recover if anything goes wrong during the upgrade.

## **Check the compatibility matrix**

You should also check the [compatibility matrix](https://success.docker.com/Policies/Compatibility_Matrix), to make sure all Docker EE components are certified to work with one another. You may also want to check the [Docker EE maintenance lifecycle](https://success.docker.com/Policies/Maintenance_Lifecycle), to understand until when your version may be supported.

## **Apply firewall rules**

Before you upgrade, make sure:

* Your firewall rules are configured to allow traffic in the ports UCP uses for communication. Learn about [UCP port requirements](https://docs.docker.com/ee/ucp/admin/install/system-requirements/#ports-used).
* Make sure you don’t have containers or services that are listening on ports used by UCP.
* Configure your load balancer to forward TCP traffic to the Kubernetes API server port (6443/TCP by default) running on manager nodes.

**Certificates**

Externally signed certificates are used by the Kubernetes API server and the UCP controller.

## **IP address consumption in 18.09+**

In Swarm overlay networks, each task connected to a network consumes an IP address on that network. Swarm networks have a finite amount of IPs based on the --subnet configured when the network is created. If no subnet is specified then Swarm defaults to a /24 network with 254 available IP addresses. When the IP space of a network is fully consumed, Swarm tasks can no longer be scheduled on that network.

Starting with Docker Engine - Enterprise 18.09 and later, each Swarm node will consume an IP address from every Swarm network. This IP address is consumed by the Swarm internal load balancer on the network. Swarm networks running on Engine versions 18.09 or greater must be configured to account for this increase in IP usage. Networks at or near consumption prior to engine version 18.09 may have a risk of reaching full utilization that will prevent tasks from being scheduled on to the network.

Maximum IP consumption per network at any given moment follows the following formula:

Max IP Consumed per Network = Number of Tasks on a Swarm Network + 1 IP for each node where these tasks are scheduled

To prevent this from happening, overlay networks should have enough capacity prior to an upgrade to 18.09, such that the network will have enough capacity after the upgrade. The below instructions offer tooling and steps to ensure capacity is measured before performing an upgrade.

The above following only applies to containers running on Swarm overlay networks. This does not impact bridge, macvlan, host, or 3rd party docker networks.

## **Upgrade Docker Engine - Enterprise**

To avoid application downtime, you should be running Docker Engine - Enterprise in Swarm mode and deploying your workloads as Docker services. That way you can drain the nodes of any workloads before starting the upgrade.

If you have workloads running as containers as opposed to swarm services, make sure they are configured with a [restart policy](https://docs.docker.com/engine/admin/start-containers-automatically/). This ensures that your containers are started automatically after the upgrade.

To ensure that workloads running as Swarm services have no downtime, you need to:

1. Determine if the network is in danger of exhaustion; and remediate to a new, larger network prior to upgrading.
2. Drain the node you want to upgrade so that services get scheduled in another node.
3. Upgrade the Docker Engine on that node.
4. Make the node available again.

If you do this sequentially for every node, you can upgrade with no application downtime. When upgrading manager nodes, make sure the upgrade of a node finishes before you start upgrading the next node. Upgrading multiple manager nodes at the same time can lead to a loss of quorum, and possible data loss.

### Determine if the network is in danger of exhaustion

Starting with a cluster with one or more services configured, determine whether some networks may require updating the IP address space in order to function correctly after an Docker Engine - Enterprise 18.09 upgrade.

1. SSH into a manager node on a cluster where your applications are running.
2. Run the following:

$ docker run -it --rm -v /var/run/docker.sock:/var/run/docker.sock docker/ip-util-check

If the network is in danger of exhaustion, the output will show similar warnings or errors:

Overlay IP Utilization Report

----

Network ex\_net1/XXXXXXXXXXXX has an IP address capacity of 29 and uses 28 addresses

ERROR: network will be over capacity if upgrading Docker engine version 18.09

or later.

----

Network ex\_net2/YYYYYYYYYYYY has an IP address capacity of 29 and uses 24 addresses

WARNING: network could exhaust IP addresses if the cluster scales to 5 or more nodes

----

Network ex\_net3/ZZZZZZZZZZZZ has an IP address capacity of 61 and uses 52 addresses

WARNING: network could exhaust IP addresses if the cluster scales to 9 or more nodes

1. Once you determine all networks are sized appropriately, start the upgrade on the Swarm managers.

#### **TRIAGE AND FIX AN UPGRADE THAT EXHAUSTED IP ADDRESS SPACE**

With an exhausted network, you can triage it using the following steps.

1. SSH into a manager node on a cluster where your applications are running.
2. Check the docker service ls output. It will display the service that is unable to completely fill all its replicas such as:

ID NAME MODE REPLICAS IMAGE PORTS

wn3x4lu9cnln ex\_service replicated 19/24 nginx:latest

1. Use docker service ps ex\_service to find a failed replica such as:

ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR PORTS

...

i64lee19ia6s \\_ ex\_service.11 nginx:latest tk1706-ubuntu-1 Shutdown Rejected 7 minutes ago "node is missing network attac…"

...

1. Examine the error using docker inspect. In this example, the docker inspect i64lee19ia6s output shows the error in the Status.Err field:

...

"Status": {

"Timestamp": "2018-08-24T21:03:37.885405884Z",

"State": "rejected",

"Message": "preparing",

\*\*"Err": "node is missing network attachments, ip addresses may be exhausted",\*\*

"ContainerStatus": {

"ContainerID": "",

"PID": 0,

"ExitCode": 0

},

"PortStatus": {}

},

...

1. Adjust your network subnet in the deployment manifest, such that it has enough IPs required by the application.
2. Redeploy the application.
3. Confirm the adjusted service deployed successfully.

### Manager upgrades when moving to Docker Engine - Enterprise 18.09 and later

The following is a constraint introduced by architectural changes to the Swarm overlay networking when upgrading to Docker Engine - Enterprise 18.09 or later. It only applies to this one-time upgrade and to workloads that are using the Swarm overlay driver. Once upgraded to Docker Engine - Enterprise 18.09, this constraint does not impact future upgrades.

When upgrading to Docker Engine - Enterprise 18.09, manager nodes cannot reschedule new workloads on the managers until all managers have been upgraded to the Docker Engine - Enterprise 18.09 (or higher) version. During the upgrade of the managers, there is a possibility that any new workloads that are scheduled on the managers will fail to schedule until all of the managers have been upgraded.

In order to avoid any impactful application downtime, it is advised to reschedule any critical workloads on to Swarm worker nodes during the upgrade of managers. Worker nodes and their network functionality will continue to operate independently during any upgrades or outages on the managers. Note that this restriction only applies to managers and not worker nodes.

### Drain the node

If you are running live application on the cluster while upgrading, remove applications from nodes being upgrades as to not create unplanned outages.

Start by draining the node so that services get scheduled in another node and continue running without downtime.

For that, run this command on a manager node:

$ docker node update --availability drain <node>

### Perform the upgrade

To upgrade a node individually by operating system, please follow the instructions listed below:

* [Windows Server](https://docs.docker.com/install/windows/docker-ee/#update-docker-ee)
* [Ubuntu](https://docs.docker.com/install/linux/docker-ee/ubuntu/#upgrade-docker-ee)
* [RHEL](https://docs.docker.com/install/linux/docker-ee/rhel/#upgrade-docker-ee)
* [CentOS](https://docs.docker.com/install/linux/docker-ee/centos/#upgrade-docker-ee)
* [Oracle Linux](https://docs.docker.com/install/linux/docker-ee/oracle/#upgrade-docker-ee)
* [SLES](https://docs.docker.com/install/linux/docker-ee/suse/#upgrade-docker-ee)

### Post-Upgrade steps for Docker Engine - Enterprise

After all manager and worker nodes have been upgrades, the Swarm cluster can be used again to schedule new workloads. If workloads were previously scheduled off of the managers, they can be rescheduled again. If any worker nodes were drained, they can be undrained again by setting --availability active.

**5 Logging Driver and It’s Configuration**

Docker includes multiple logging mechanisms to help you [get information from running containers and services](https://docs.docker.com/engine/admin/logging/view_container_logs/). These mechanisms are called logging drivers.

Each Docker daemon has a default logging driver, which each container uses unless you configure it to use a different logging driver.

## **Configure the default logging driver**

To configure the Docker daemon to default to a specific logging driver, set the value of log-driver to the name of the logging driver in the daemon.json file, which is located in /etc/docker/ on Linux hosts or C:\ProgramData\docker\config\ on Windows server hosts. The default logging driver is json-file. The following example explicitly sets the default logging driver to syslog:

{

"log-driver": "syslog"

}

If the logging driver has configurable options, you can set them in the daemon.json file as a JSON object with the key log-opts. The following example sets two configurable options on the json-file logging driver:

{

"log-driver": "json-file",

"log-opts": {

"max-size": "10m",

"max-file": "3",

"labels": "production\_status",

"env": "os,customer"

}

}

If you do not specify a logging driver, the default is json-file. Thus, the default output for commands such as docker inspect <CONTAINER> is JSON.

To find the current default logging driver for the Docker daemon, run docker info and search for Logging Driver. You can use the following command on Linux, macOS, or PowerShell on Windows:

$ docker info --format '{{.LoggingDriver}}'

json-file

## **Configure the logging driver for a container**

When you start a container, you can configure it to use a different logging driver than the Docker daemon’s default, using the --log-driver flag. If the logging driver has configurable options, you can set them using one or more instances of the --log-opt <NAME>=<VALUE> flag. Even if the container uses the default logging driver, it can use different configurable options.

The following example starts an Alpine container with the none logging driver.

$ docker run -it --log-driver none alpine ash

To find the current logging driver for a running container, if the daemon is using the json-file logging driver, run the following docker inspect command, substituting the container name or ID for <CONTAINER>:

$ docker inspect -f '{{.HostConfig.LogConfig.Type}}' <CONTAINER>

json-file

### Use environment variables or labels with logging drivers

Some logging drivers add the value of a container’s --env|-e or --label flags to the container’s logs. This example starts a container using the Docker daemon’s default logging driver (let’s assume json-file) but sets the environment variable os=ubuntu.

$ docker run -dit --label production\_status=testing -e os=ubuntu alpine sh

If the logging driver supports it, this adds additional fields to the logging output. The following output is generated by the json-file logging driver:

"attrs":{"production\_status":"testing","os":"ubuntu"}

When using Docker Community Engine, the docker logs command is only available on the following drivers:

* local
* json-file
* journald

The capacity of the host storage where docker’s data directory resides determines the maximum size of the log file information.

**6 DOCKER SWARM**

What is a docker Swarm?

**Docker Swarm** is a clustering and scheduling tool for **Docker** containers. With **Swarm**, IT administrators and developers can establish and manage a cluster of **Docker** nodes as a single virtual system.

A **swarm** consists of multiple **Docker** hosts which run in **swarm** mode and act as managers (to manage membership and delegation) and workers (which run **swarm** services). ... **Docker works** to maintain that desired state. For instance, if a worker node becomes unavailable, **Docker** schedules that node's tasks on other nodes.

**7 Configuring docker swarm**

### Configure firewall

You will need to configure firewall rules for a swarm cluster to work properly on both nodes. Allow the ports 7946, 4789, 2376, 2376, 2377, and 80 using the UFW firewall with the following command:

sudo ufw allow 2376/tcp && sudo ufw allow 7946/udp && sudo ufw allow 7946/tcp && sudo ufw allow 80/tcp && sudo ufw allow 2377/tcp && sudo ufw allow 4789/udp Next, reload the UFW firewall and enable it to start on boot: sudo ufw reload && sudo ufw enable Restart the Docker service to affect the Docker rules: sudo systemctl restart docker

## Create Docker Swarm cluster

First, you will need to initialize the cluster with the IP address, so your node acts as a Manager node. On the Manager Node, run the following command for advertising IP address:

docker swarm init --advertise-addr 192.168.0.103

 Docker Engine joins the swarm depending on the join-token you provide to the docker swarm join command. The node only uses the token at join time.

check the status of the Manager Node with the following command: docker node ls

## Add Worker Node to swarm cluster

Manager Node is now configured properly, it’s time to add Worker Node to the Swarm Cluster. First, copy the output of the “swarm init” command from the previous step, then paste that output on the Worker Node to join the Swarm Cluster:

docker swarm join --token SWMTKN-1-5p5f6p6tv1cmjzq9ntx3zmck9kpgt355qq0uaqoj2ple629dl4-5880qso8jio78djpx5mzbqcfu 192.168.0.103:2377

You should see the following output:

This node joined a swarm as a worker.

Now, on the Manager Node, run the following command to list the Worker Node: docker node ls

You should see the Worker Node in the following output:

ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS

iwjtf6u951g7rpx6ugkty3ksa \* Manager-Node Ready Active Leader

snrfyhi8pcleagnbs08g6nnmp Worker-Node Ready Active

## Launch web service in Docker Swarm

Docker Swarm Cluster is now up and running, it’s time to launch the web service inside Docker Swarm Mode. On the Manager Node, run the following command to deploy a web server service: docker service create --name webserver -p 80:80 httpd

The above command will create an Apache web server container and map it to port 80, so you can access Apache web server from the remote system. Now, you can check the running service with the following command: docker service ls You should see the following output:

ID NAME

MODE REPLICAS IMAGE PORTS

nnt7i1lipo0h webserver replicated 0/1 apache:latest \*:80->80/tcp

Next, scale the web server service across two containers with the following command: docker service scale webserver=2

Then, check the status of web server service with the following command: docker service ps webserver You should see the following output:

ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR PORTS

7roily9zpjvq webserver.1 httpd:latest Worker-Node Running Preparing about a minute ago

r7nzo325cu73 webserver.2 httpd:latest Manager-Node Running Preparing 58 seconds ago

## **8 Swarm backup & restore**

## **Prerequisites**

Before performing these steps, you must meet the following requirements:

* Docker Engine >= v17.03
* Linux

**Backup**

To backup the Swarm on Linux using Docker Engine >= 17.03, you can use the following steps.

Since this will need you to stop the engine of a manager, your cluster need to be healthy with at least 3 managers.

Be sure to maintain the quorum of swarm managers. During the time that a manager is shut down, your swarm is more vulnerable to losing the quorum if further nodes are lost. The number of managers you run is a trade-off. If you regularly take down managers to do backups, consider running a 5-manager swarm, so that you can lose an additional manager while the backup is running, without disrupting your services.

1. Select a manager node to do the operation. Try not to choose the leader one in order to avoid a new election inside the cluster:
2. docker node ls -f "role=manager" | tail -n+2 | grep -vi leader
3. Optional: Store the Docker version to a variable for easy addition to your backup name.
4. ENGINE=$(docker version -f '{{.Server.Version}}')
5. Stop the Docker Engine on the manager before backing up the data, so that no data is being changed during the backup:
6. systemctl stop docker
7. Backup the entire Swarm folder:
8. tar cvzf "/tmp/swarm-${ENGINE}-$(hostname -s)-$(date +%s%z).tgz" /var/lib/docker/swarm/

Note: You can decode the Unix epoch in the filename by typing *date -d @timestamp*. For example:

date -d @1531166143

Mon Jul 9 19:55:43 UTC 2018

1. Restart the manager Docker Engine:
2. systemctl start docker

**Restore**

Before restoring the Swarm:

* You have to restore the backup on the same Docker Engine version.
* You must use the same IP as the node you made the backup with, this is mandatory. The command to force the new cluster does not reset the IP in the Swarm data.

1. Shutdown the Docker Engine on the node where you choose to restore:
2. systemctl stop docker
3. Remove the contents of the swarm folder /var/lib/docker/swarm.
4. Restore the swarm folder /var/lib/docker/swarm with the content of the backup.
5. Restart the new node:
6. systemctl start docker
7. Re-initialize the Swarm to isolate this node into his own new cluster. Avoid attempting to connect it to nodes that were part of the old swarm, and presumably no longer exist.
8. docker swarm init --force-new-cluster
9. Add the manager and worker nodes to bring your new swarm up to operating capacity.

**9 Namespace & c groups**

**cgroups** limits the resources which a process or set of processes can use these resources could be CPU,Memory,Network I/O or access to filesystem while **namespace** restrict the visibility of group of processes to the rest of the system.

* **cgroup**: Control Groups provide a mechanism for aggregating/partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behaviour.
* **namespace**: wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource.

In short:

* **Cgroups** = limits how much you can use;
* **namespaces** = limits what you can see (and therefore use)

See more at "[Anatomy of a Container: Namespaces, cgroups & Some Filesystem Magic](http://fr.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesystem-magic-linuxcon)" by [Jérôme Petazzoni](http://fr.slideshare.net/jpetazzo).

Cgroups involve resource metering and limiting:

* memory
* CPU
* block I/O
* network

Namespaces provide processes with their own view of the system

Multiple namespaces:

* pid
* net
* mnt
* uts
* ipc

There are currently **namespaces** for the hostname, process IDs, user and group IDs, mounts, networking and inter-process communications. Some of these are absolutely essential to containerization, whereas others are only needed in certain circumstances

## CGroups Examples

--cpu-shares

--cpuset-cpus

--memory-reservation

--kernel-memory

--blkio-weight (block IO)

--device-read-iops

--device-write-iops

## Namespace Examples

Cgroup CLONE\_NEWCGROUP Cgroup root directory

IPC CLONE\_NEWIPC System V IPC, POSIX message queues

Network CLONE\_NEWNET Network devices, stacks, ports, etc.

Mount CLONE\_NEWNS Mount points

PID CLONE\_NEWPID Process IDs

User CLONE\_NEWUSER User and group IDs

UTS CLONE\_NEWUTS Hostname and NIS domain name