Docker Certified Associate Training

Source: https://docs.docker.com

simplilearn

Orchestration

Learning Objectives

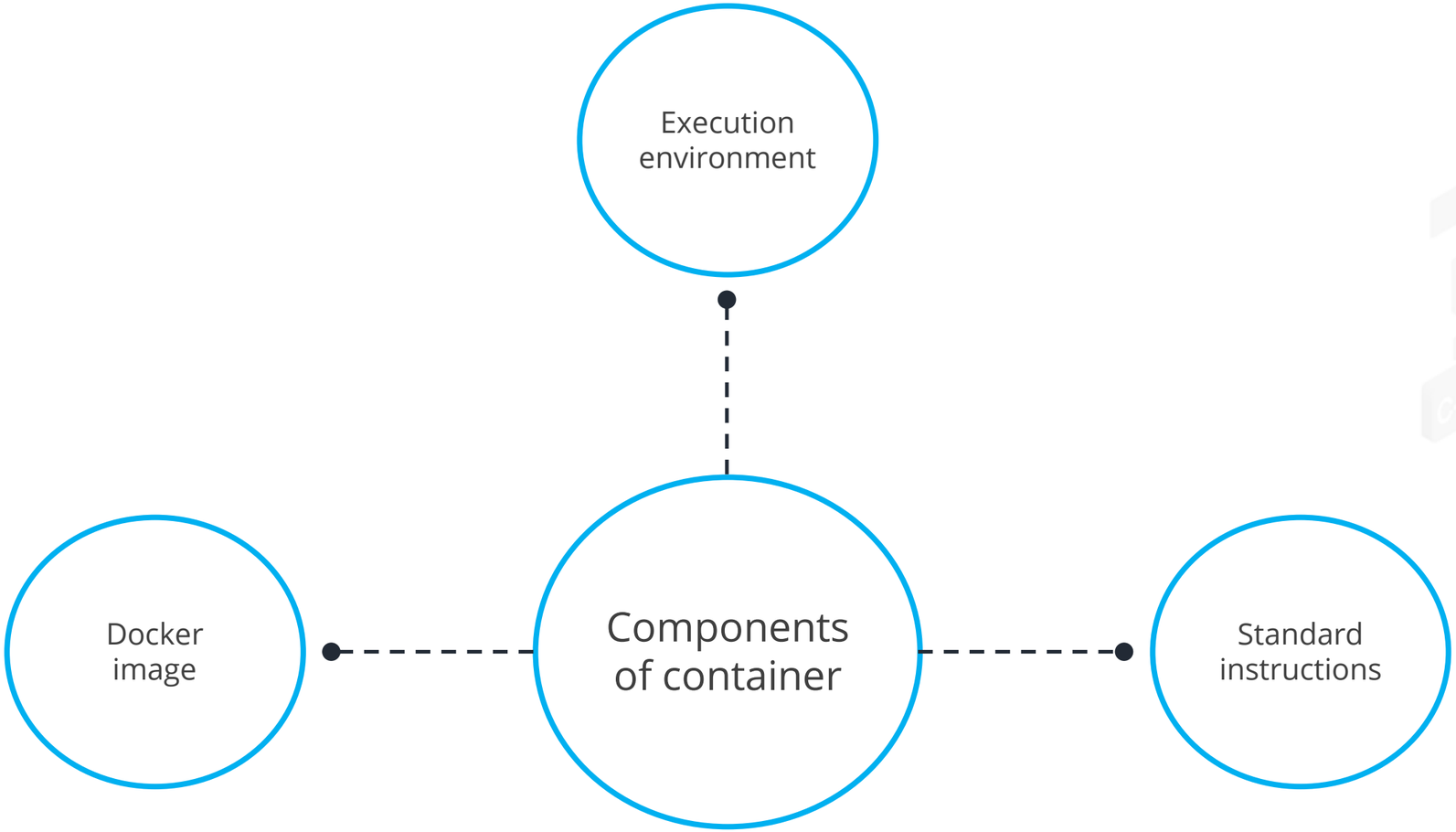
By the end of this lesson, you will be able to:

- Describe services, tasks, containers, and their relationship
- Define swarm mode, its features, and its advantages
- Set up a swarm mode cluster with nodes
- Use compose, perform swarm draining, deploy docker stack, and inspect docker
- Perform scaling of services, mount volumes, and comprehend bind/tmpfs mounts
- Understand the significance of labels, quorum, and templates

Containers

Container: Overview

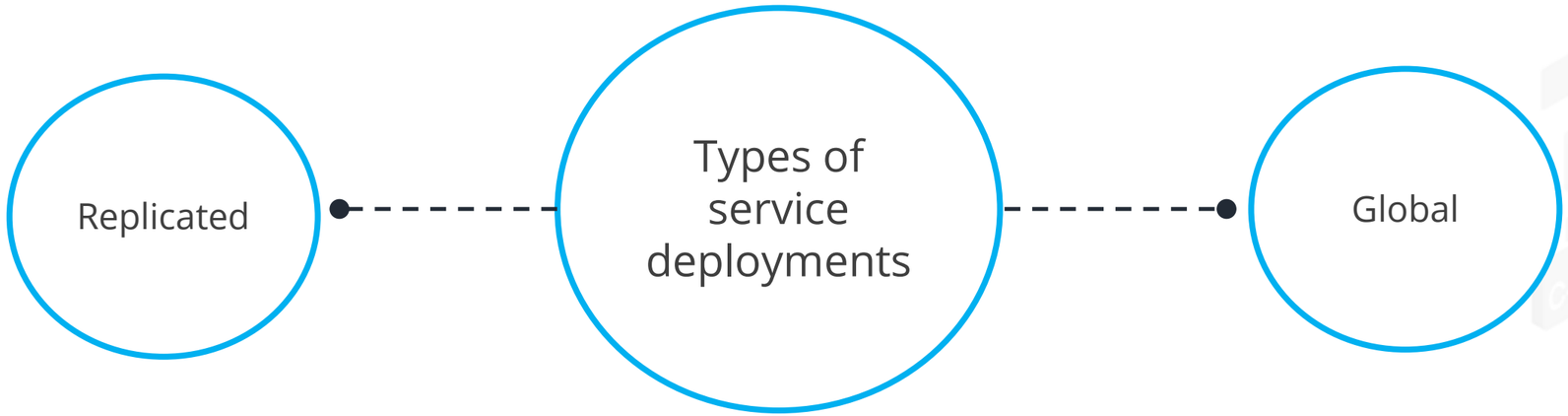
It is a runtime instance of a Docker image.



Service and Its Types

Service: Overview

Service defines the desired way in which the application containers can be run in a swarm.



Service: Overview

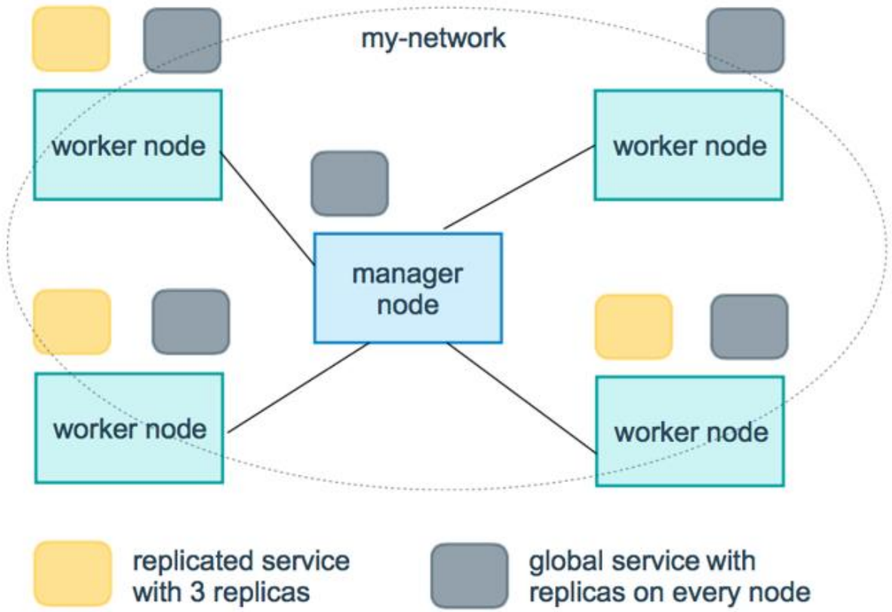
Service defines:

- Which container image should run in the swarm
- Which commands should run in the container

"Desired State" is defined by service in orchestration, that is:

- The number of containers that can be run as tasks
- The number of constraints to deploy the containers

Replicated and Global Services



Replicated Service:

- The number of identical tasks that need to run is specified in Replicated Service

Global Service:

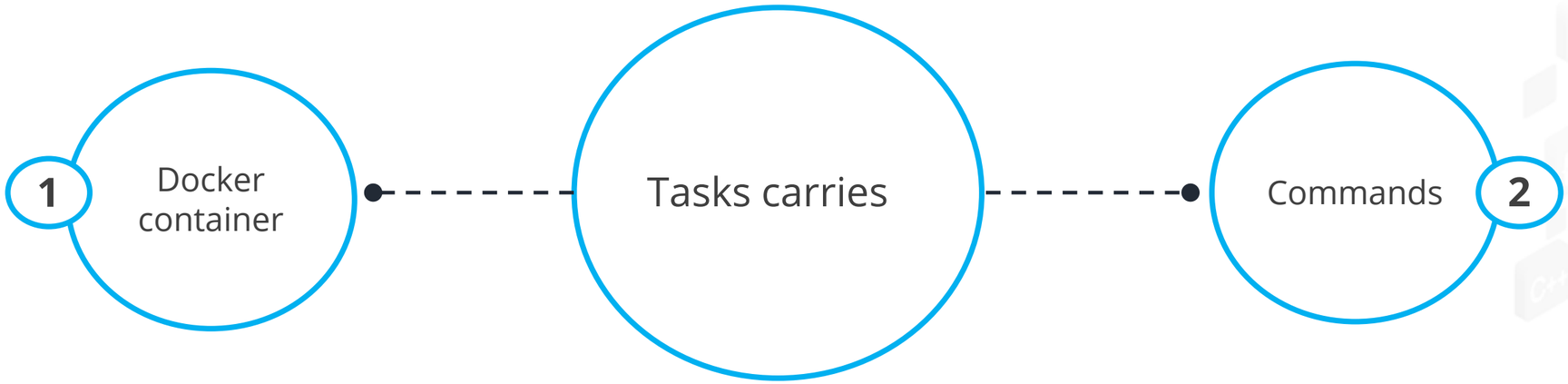
- The tasks that need to run are not specified in Global Service

Source: https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/#replicated-and-global-services

Tasks

Task: Overview

Task: An atomic unit of scheduling within a swarm



Note: According to replicas set in service scale, worker nodes are tasks that are assigned by manager nodes.

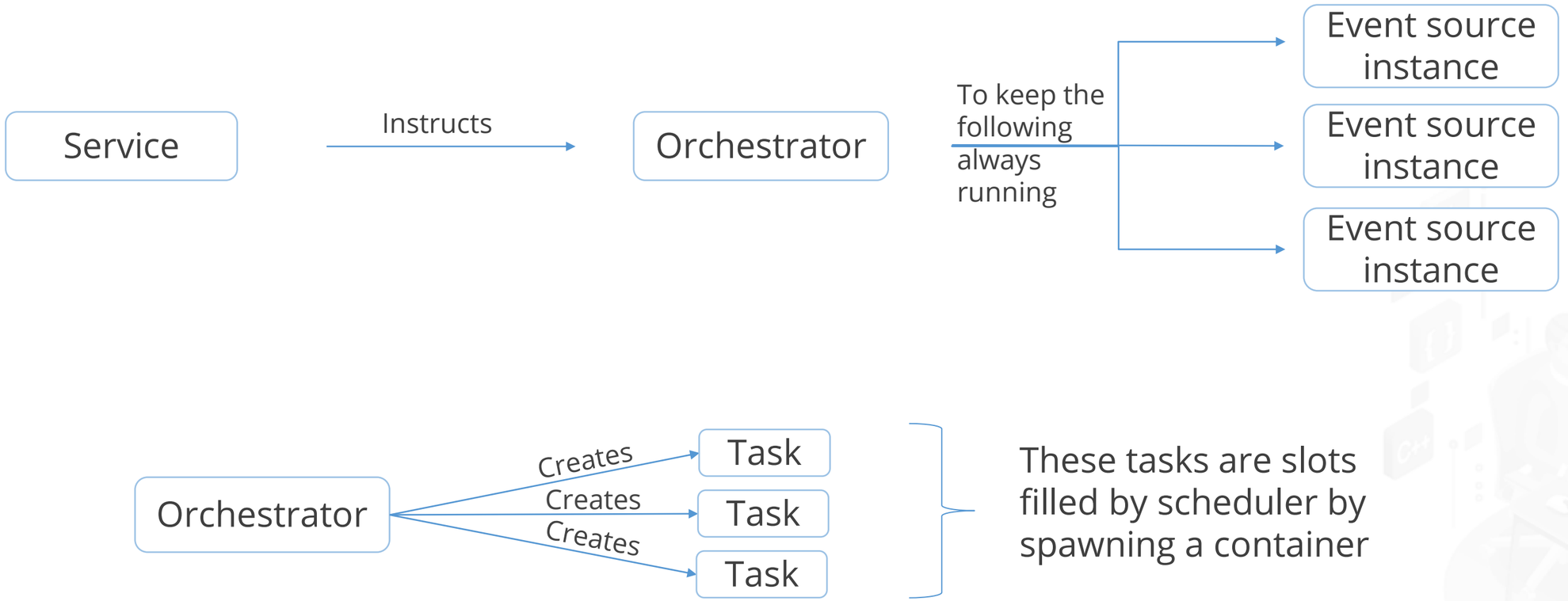
Task: Overview

It is a one-directional mechanism which progresses monotonically through different states:

The task and its container are removed by the orchestrator on task failure. The failed task is then replaced by the newly created task, which is according to the desired state specified by the service.

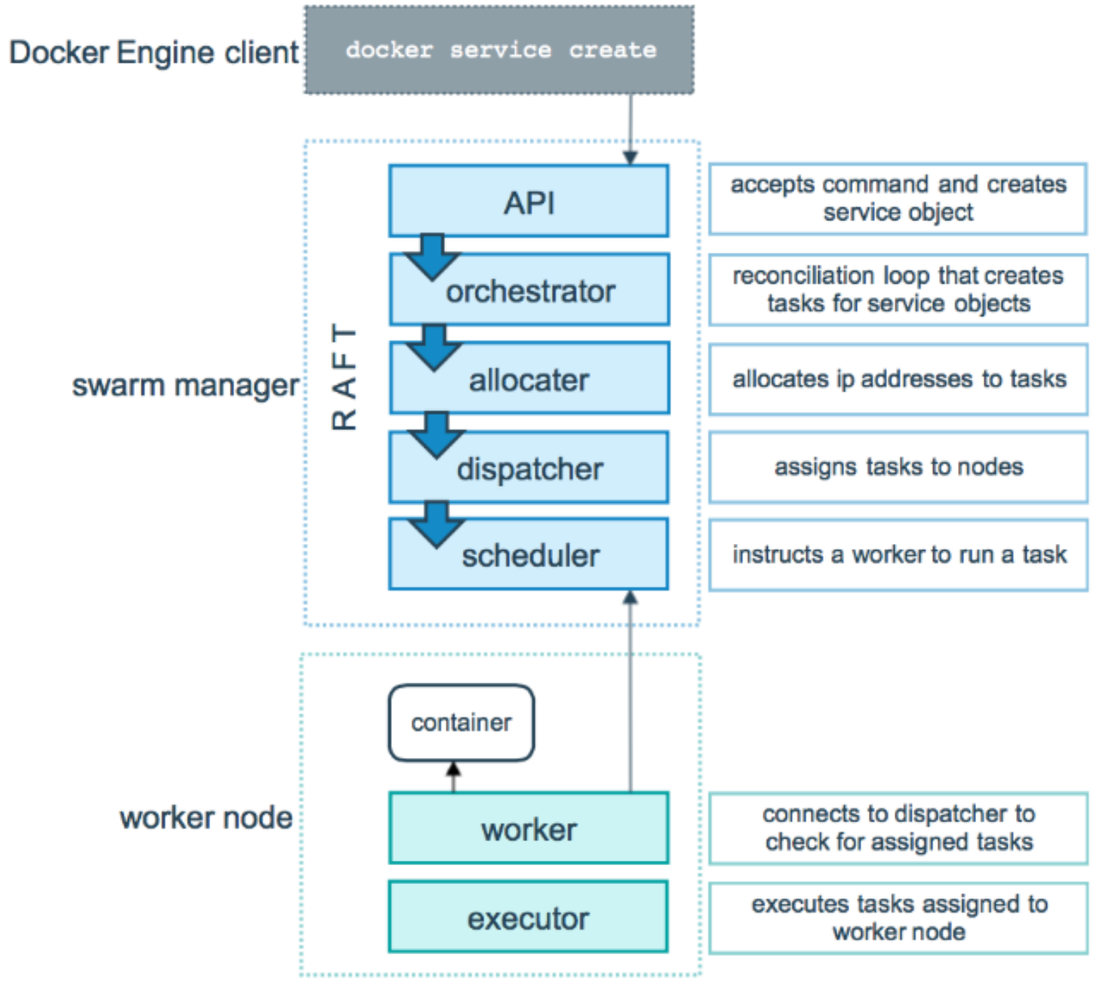
Note: The replicas set in service scale tell that the worker nodes are tasks that are assigned by manager nodes.

Task Scheduling



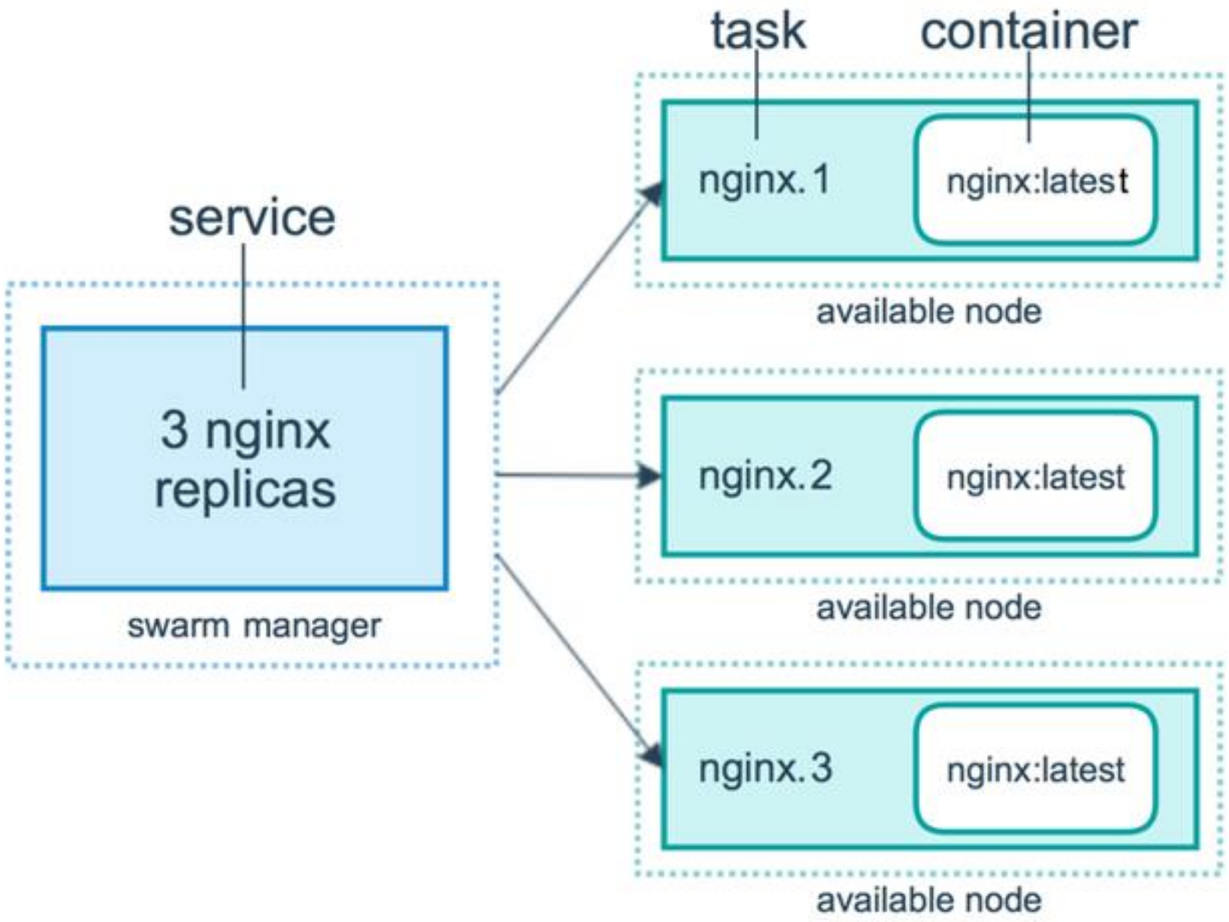
Note: The orchestrator creates a new replica task if an event source task keeps on subsequently crashing or fails its health check. This creation of new replica tasks leads to the spawning of new containers.

Task Scheduling



The diagram above shows how a swarm mode accepts service, creates requests, and schedules tasks to the worker nodes.

Services, Tasks, and Containers: Relationship

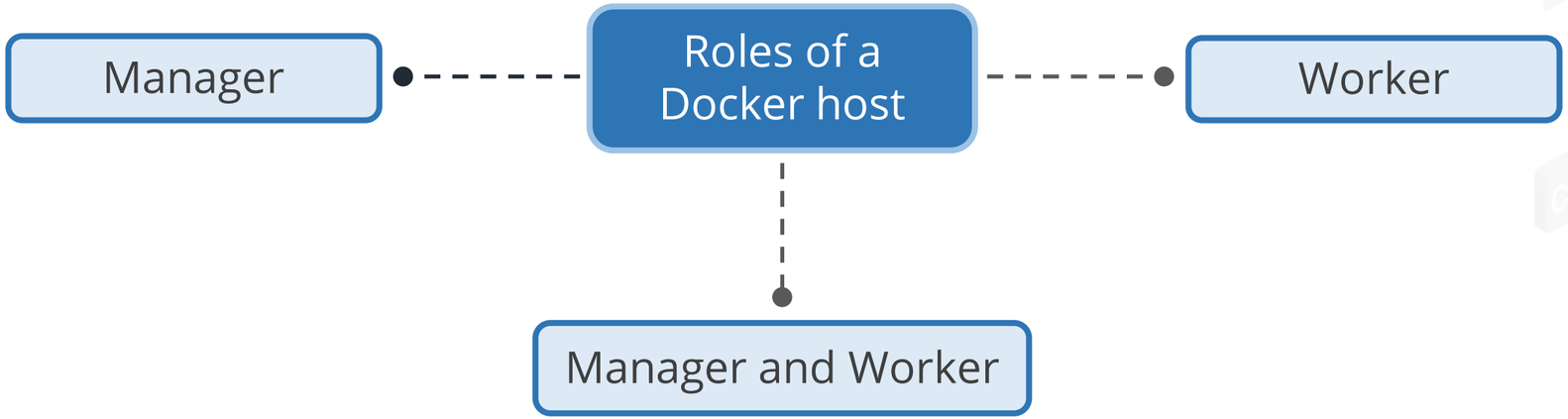


Source: https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/#services-tasks-and-containers

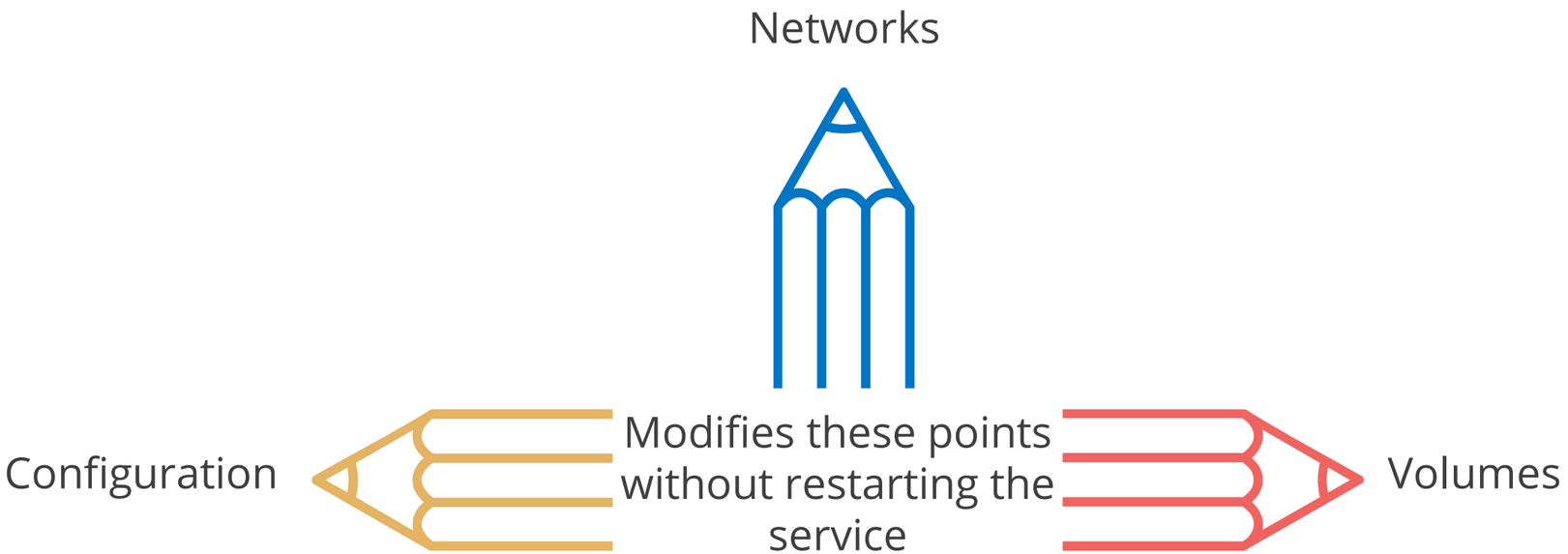
Cluster Management and Orchestration

Swarm: Overview

A cluster of one or more Docker Engines running in swarm mode is called a Swarm.



Swarm: Advantage



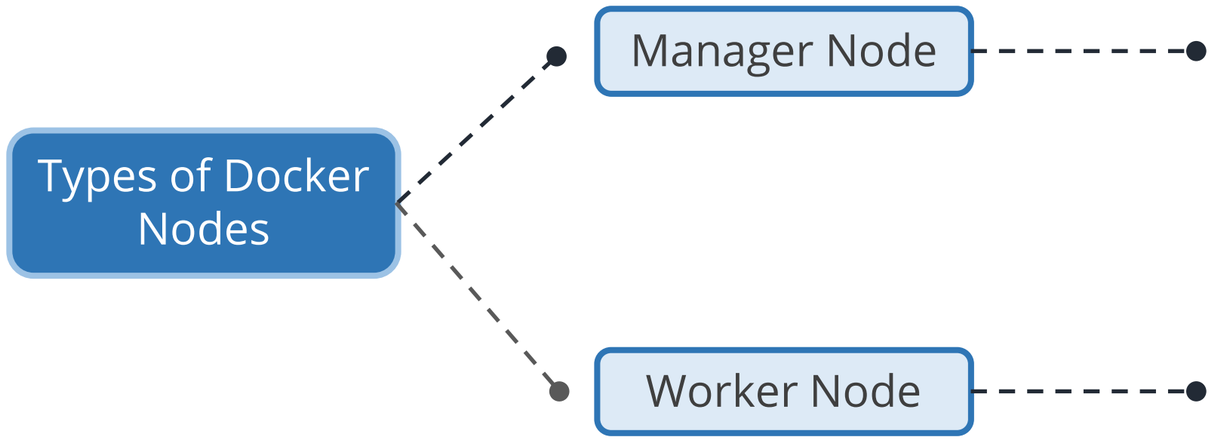
Swarm: Features

- Integrated cluster management and Docker Engine
- Decentralized design
- Declarative service model
- Scaling
- Desired state reconciliation
- Multi-host networking
- Service discovery
- Load balancing
- Secure by default
- Rolling updates

Cluster Management and Orchestration: Nodes

Nodes: Overview

In swarm, the instances of Docker Engine are distributed across multiple physical or virtual machines. These instances of Docker engine are known as Nodes.



Manager Node

Manager Node: Tasks Handled

Cluster management tasks handled by Manager Node:

- Maintaining cluster state
- Scheduling services
- Serving HTTP API endpoints (swarm mode)

Manager Node: Fault-Tolerance Feature

How to take advantage of the features of swarm mode fault-tolerance:

You can take advantage by implementing an odd number of nodes according to high-availability requirements.

Multiple managers help in recovering from the failure of a manager node without downtime.

For example:

- One manager loss is tolerated by three manager swarms.
- Two manager nodes' loss is simultaneously tolerated by five manager swarms.
- (N-1)/2 managers loss is tolerated by an N manager cluster.

Note:

- A maximum of seven manager nodes is recommended by Docker for a swarm.
- High performance or increased scalability does not depend on the addition of more managers. Generally, the opposite is true.

Worker Node

Worker Node

It is a Docker Engine instance that executes the containers.

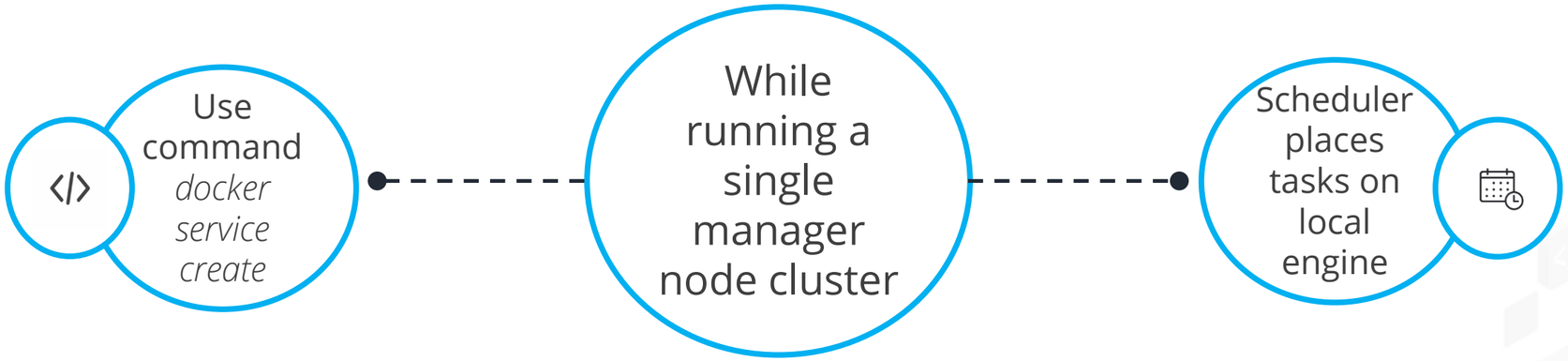
Worker nodes do not play a role in:

- Raft distributed state
- Scheduling decisions
- Serving the HTTP API (swarm mode)

Note:

- All manager nodes are worker nodes.
- Swarm can be created for one manager node.
- At least one manager node is necessary to have a worker node.

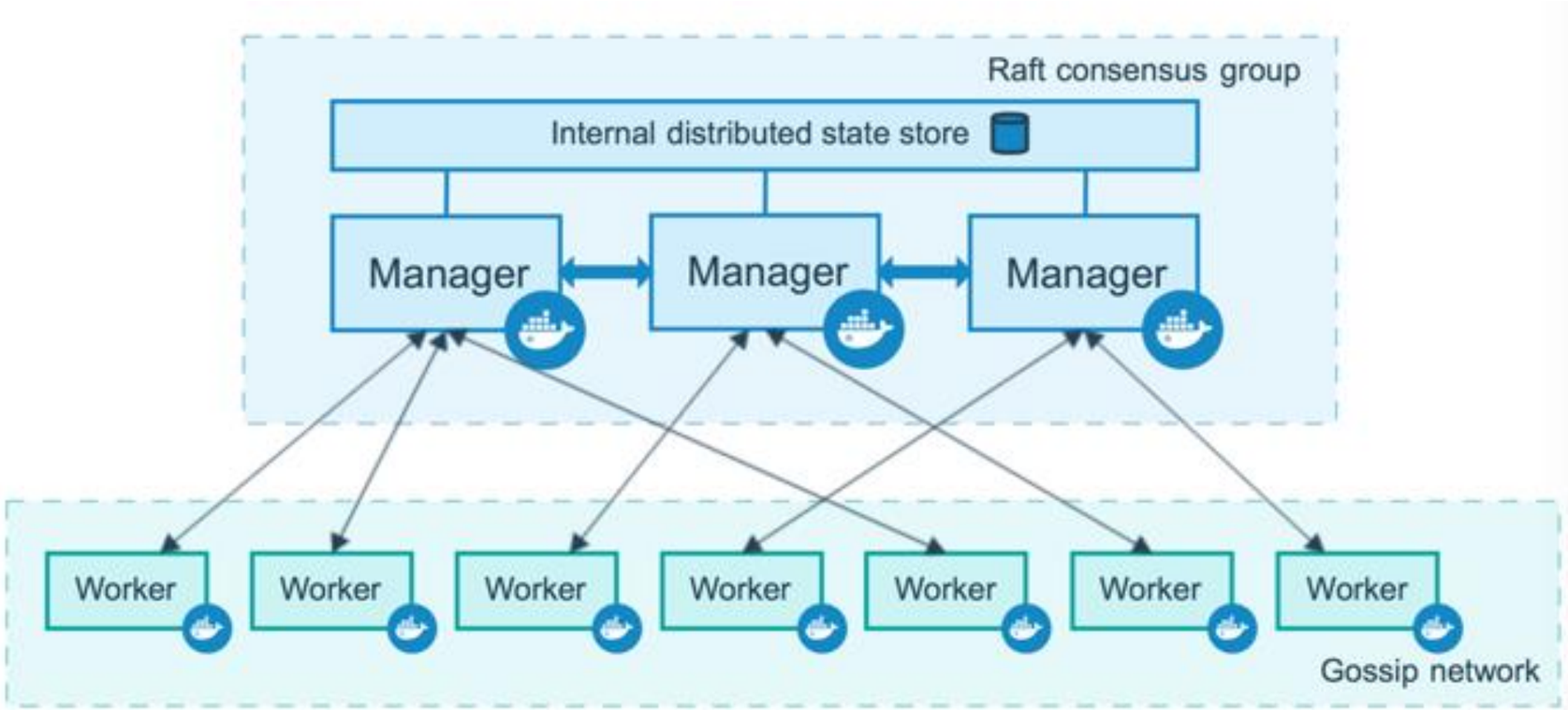
Worker Node: Scheduler



How to stop scheduler from placing tasks on manager node in a multi-node swarm:

- Manager node availability must be set to Drain
  - Result:
    - Tasks are stopped on nodes by the scheduler.
    - Tasks are scheduled on an Active node by the scheduler.
    - New tasks are not assigned to nodes by the scheduler.

How Does a Swarm Node Work?



- Swarm mode helps in creating one or more Docker Engine's cluster.
- These clusters of Docker Engine are known as the swarm.
- Swarm contains one or more than one nodes.

Source: https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/

Set Up Swarm Cluster with Manager and Worker Nodes

Problem Statement: You have been asked by your manager to set up a swarm node cluster and add the manager and worker nodes to it.

Steps to Perform:

1. Install *docker-machine* using the *curl* command and list the files present in it.
2. Install *yum* and import *Oracle public key* to your system.
3. Add *Oracle Virtualbox PPA* and install *virtualbox* image.
4. Launch *virtualbox* and initialize docker swarm using *init* command.
5. Make the current node as a manager and add worker nodes to the swarm.

ASSISTED PRACTICE

Join Nodes to Swarm

Problem Statement: You are required to join worker nodes to a swarm cluster which can be used for load balancing of traffic.

Steps to Perform:

1. Initiate Docker swarm cluster.
2. Use *join-token manager* command to retrieve join command for manager nodes.
3. Add a manager node to the swarm.
4. Use *join-token worker* command to retrieve join command for worker nodes.
5. Add a worker node to the swarm.

ASSISTED PRACTICE

Create Replicated and Global Services

Problem Statement: Your team lead wants you to create instances of replicated and global services so that you can run multiple tasks on them.

Steps to Perform:

1. Create a replicated service with 3 replicas.
2. Create a global service with *--mode* flag set to *global*.
3. List all the Docker services.
4. Check the status of replicated and global services.

ASSISTED PRACTICE

Running Container vs. Running Service

Problem Statement: You are required to run a container as well as a service using same docker image so that you can understand the difference between the two.

Steps to Perform:

1. Run a container using the nginx image.
2. List all the running containers.
3. Run the nginx image as a service.
4. List all the running services in the swarm.
5. List all the tasks of nginx service.

ASSISTED PRACTICE

Create an Overlay Network

Problem Statement: Your colleague wants you to create an overlay network so that it can be used by individual container or swarm services in order to communicate with each other.

Steps to Perform:

1. Initialize Docker daemon as a swarm manager.
2. Create an overlay network with both *--opt encrypted* and *--attachable* flag to attach unmanaged containers to that network.
3. Remove the existing *ingress* network and create a new overlay network using the *--ingress* flag. Set the *MTU* to 1200, the *subnet* to 10.11.0.0/16, and the *gateway* to 10.11.0.2.

ASSISTED PRACTICE

Deploy a Service on an Overlay Network

Problem Statement: Your manager asks you to deploy a service on an overlay network that can be used by different containers in the network.

Steps to Perform:

1. Attach a service to an existing overlay network.
2. List the networks connected to the service.
3. Inspect the network to check which containers and services are attached.

ASSISTED PRACTICE

Run a Container into a Running Service under Swarm

Problem Statement: You are required to run a container into the running service under swarm cluster that can be used to run multiple tasks.

Steps to Perform:

1. On each node in the swarm, run the nginx image as a service in *global* mode.
2. Deploy an nginx service running three replica containers on the swarm.
3. Check the status of each container instance running within the service.

ASSISTED PRACTICE

Demonstrate Locking in Swarm Cluster

Problem Statement: For safety reasons, you have been asked to lock the swarm cluster preventing it from unlocking itself on the restart of Docker Daemon.

Steps to Perform:

1. Use the *--autolock* flag to enable auto-locking while initializing a new swarm.
2. Restart the Docker Daemon to check if auto-locking is enabled.
3. Use --*autolock=false* or *--autolock=true* flag to disable or enable *autolock the* on an existing swarm.
4. Unlock a swarm with *unlock* command.
5. View the current unlock-key with the *unlock-key* command.

UNASSISTED PRACTICE

Drain Swarm Node

Swarm Draining

Purpose of setting the node to drain availability is to prevent the node from receiving new tasks from swarm manager.

What happens to the rest of the nodes with active availability?

The manager launches the replica tasks on the nodes with active availability

Setting a node to drain availability will not remove the following containers from that node:

- Containers created with *docker run*
- Containers created with *docker-compose up*
- Containers created with Docker Engine API

Draining the Swarm Node

Problem Statement: Your manager wants you to drain a node in the swarm cluster preventing it from receiving new tasks from the swarm manager.

Steps to Perform:

1. List all active nodes.
2. Start the *redis* service and check the status of the individual tasks.
3. Drain a node that has an assigned with a task and inspect it for *Availability*.
4. Check how the tasks have been updated for the *redis* service.
5. Return the drained node to an active state and inspect it for updated state.

ASSISTED PRACTICE

Docker Inspect

Command: Docker Inspect

docker inspect → This command provides detailed information on various constructs controlled by the Docker. The result of this command reflects in a JSON array.

Command:

docker inspect [OPTIONS] NAME|ID [NAME|ID...]

It's the base command for Docker CLI

Options	Description
--format, -f	An output is formatted using GO template
--size, -s	File sizes are displayed if the type is a container
--type, -t	Return JSON for a specified type

Docker Inspect: Examples

Fetch IP address of an instance:

```
$ docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' $INSTANCE_ID
```

Fetch log path of an instance:

```
$ docker inspect --format='{{.LogPath}}' $INSTANCE_ID
```

Fetch image name of an instance:

```
$ docker inspect --format='{{.Config.Image}}' $INSTANCE_ID
```

Inspect a Service on Swarm

Problem Statement: You are required to inspect a service on a particular node to check the *availability* of it allowing swarm manager to accordingly assign tasks.

Steps to Perform:

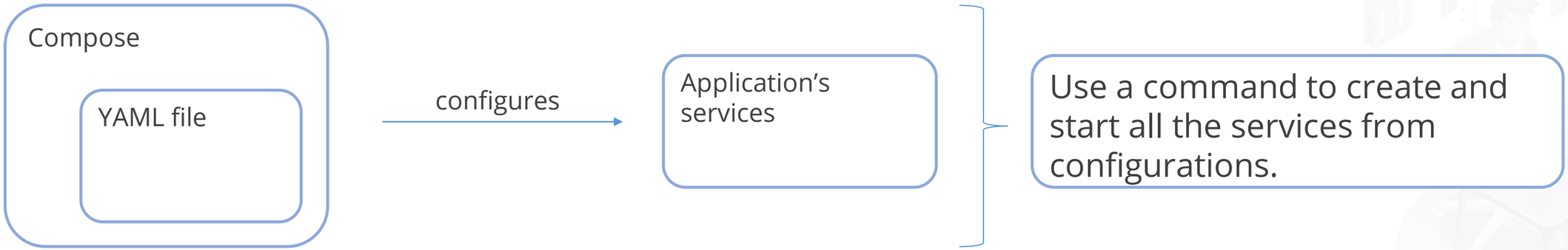
1. Start the *redis* service.
2. Check the tasks assigned to nodes by swarm manager.
3. Inspect the node to check its availability.
4. Return the service details in JSON format.

ASSISTED PRACTICE

Compose

Compose: Overview

It is a tool that defines and runs multi-container Docker applications.



Compose: Overview

Compose is great for:

- Development environment
- Testing environment
- Staging environment
- CI workflow

The three-step process that is required to use the Compose:

Define the environment of the app with a Dockerfile in order to reproduce it anywhere.

Define services that are part of an app in the *docker-compose.yml* in order to run them together in an isolated environment.

Run the command *docker-compose up.* This command will start the Compose and run the entire app.

Compose: Overview

A *docker-compose.yml* looks like this:

```
version: '3’
services:
        web:
                build: .
                ports:
                - "5000:5000"
                volumes:
                - .:/code
                - logvolume01:/var/log
                links: - redis
        redis:
                image: redis
        volumes:
                logvolume01: {}
```

Compose: Features

Features:

- Multiple isolated environments are provided on a single host
- Volume data is preserved when containers are created
- Containers that have changed are recreated
- Variables and moving a composition between environments are allowed

Compose has commands that:

- Start, stop, and rebuild the services
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service

Compose: Configurations

List of Configurations

build | cap_add, cap_drop | cgroup_parent | command | configs | container_name

depends_on | dns

- CONTEXT
- DOCKERFILE
- ARGS
- CACHE_FROM
- LABELS
- SHM_SIZE
- TARGET

Here *build* is specified as a string that contains a path

```
version: "3.7"
services:
        webapp:
        build: ./dir
```

List of Configurations

build | cap_add, cap_drop | cgroup_parent | command | configs | container_name

depends_on | dns

- CONTEXT
- DOCKERFILE
- ARGS
- CACHE_FROM
- LABELS
- SHM_SIZE
- TARGET

Here *build* is specified as an object with the path

```
version: "3.7"
services:
        webapp:
                build:
                        context: ./dir
                        dockerfile: Dockerfile-alternate
                        args:
                                buildno: 1
```

List of Configurations

build | cap_add, cap_drop | cgroup_parent | command | configs | container_name

depends_on | dns

- CONTEXT
- DOCKERFILE
- ARGS
- CACHE_FROM
- LABELS
- SHM_SIZE
- TARGET

```
build:
        context: ./dir
```

List of Configurations

build | cap_add, cap_drop | cgroup_parent | command | configs | container_name

depends_on | dns

- CONTEXT
- DOCKERFILE
- ARGS
- CACHE_FROM
- LABELS
- SHM_SIZE
- TARGET

```
build:
        context: .
        dockerfile : Dockerfile-alternate
```

List of Configurations

build | cap_add, cap_drop | cgroup_parent | command | configs | container_name

depends_on | dns

- CONTEXT
- DOCKERFILE
- ARGS
- CACHE_FROM
- LABELS
- SHM_SIZE
- TARGET

Specify the following arguments in the Dockerfile:

ARG buildno
ARG gitcommithash

RUN echo "Build number: $buildno"
RUN echo "Based on commit: $gitcommithash"

List of Configurations

build | cap_add, cap_drop | cgroup_parent | command | configs | container_name

depends_on | dns

- CONTEXT
- DOCKERFILE
- ARGS
- CACHE_FROM
- LABELS
- SHM_SIZE
- TARGET

After specifying the arguments under *build*, pass a mapping or a list:

```
build:
    context: .
    args:
        buildno: 1
        gitcommithash: cdc3b19
```

```
build:
    context: .
    args:
        - buildno=1
        - gitcommithash=cdc3b19
```

List of Configurations

build | cap_add, cap_drop | cgroup_parent | command | configs | container_name

depends_on | dns

- CONTEXT
- DOCKERFILE
- ARGS
- CACHE_FROM
- LABELS
- SHM_SIZE
- TARGET

On omitting, the values at the build time are same as that of the values in the environment where Compose is running.

```
args:
    - buildno
    - gitcommithash
```

List of Configurations

build | cap_add, cap_drop | cgroup_parent | command | configs | container_name

depends_on | dns

- CONTEXT
- DOCKERFILE
- ARGS
- CACHE_FROM
- LABELS
- SHM_SIZE
- TARGET

```
build:
        context: .
        cache_from:
                - alpine:latest
                - corp/web_app:3.14
```

List of Configurations

build | cap_add, cap_drop | cgroup_parent | command | configs | container_name

depends_on | dns

CONTEXT
DOCKERFILE
ARGS
CACHE_FROM
LABELS
SHM_SIZE
TARGET

Use Docker labels to add metadata to the resulting image. This can be done using either an array or a dictionary.

```
build:
        context: .
        labels:
                com.example.description: "Accounting webapp“
                com.example.department: "Finance"
                com.example.label-with-empty-value: “”
```

```
build:
        context: .
        labels:
                - "com.example.description=Accounting webapp"
                - "com.example.department=Finance"
                - "com.example.label-with-empty-value"
```

List of Configurations

build | cap_add, cap_drop | cgroup_parent | command | configs | container_name

depends_on | dns

- CONTEXT
- DOCKERFILE
- ARGS
- CACHE_FROM
- LABELS
- SHM_SIZE
- TARGET

The values are either mentioned in bytes or as a string which expresses a byte value:

```
build:
        context: .
        shm_size: '2gb'
```

```
build:
        context: .
        shm_size: 10000000
```

List of Configurations

build | cap_add, cap_drop | cgroup_parent | command | configs | container_name

depends_on | dns

CONTEXT
DOCKERFILE
ARGS
CACHE_FROM
LABELS
SHM_SIZE
TARGET

```
build:
        context: .
        target: prod
```

List of Configurations

build | cap_add, cap_drop | cgroup_parent | command | configs | container_name

depends_on | dns

```
cap_add:
        - ALL
```

```
cap_drop:
        - NET_ADMIN
        - SYS_ADMIN
```

List of Configurations

build | cap_add, cap_drop | cgroup_parent | command | configs | container_name

depends_on | dns

cgroup_parent: m-executor-abcd

List of Configurations

build | cap_add, cap_drop | cgroup_parent | **command** | configs | container_name

depends_on | dns

command: bundle exec thin -p 3000

The command can also be written as a list:

command: ["bundle", "exec", "thin", "-p", "3000"]

List of Configurations

build | cap_add, cap_drop | cgroup_parent | command | **configs** | container_name

depends_on | dns

SHORT SYNTAX

LONG SYNTAX

The container gets the access to the config and mounts the config name at */<config name>* within the container

```
version: "3.7"
services:
        redis:
        image: redis:latest
                deploy:
                        replicas: 1
                configs:
                        - my_config
                        - my_other_config
configs:
        my_config:
                file: ./my_config.txt
        my_other_config:
                external: true
```

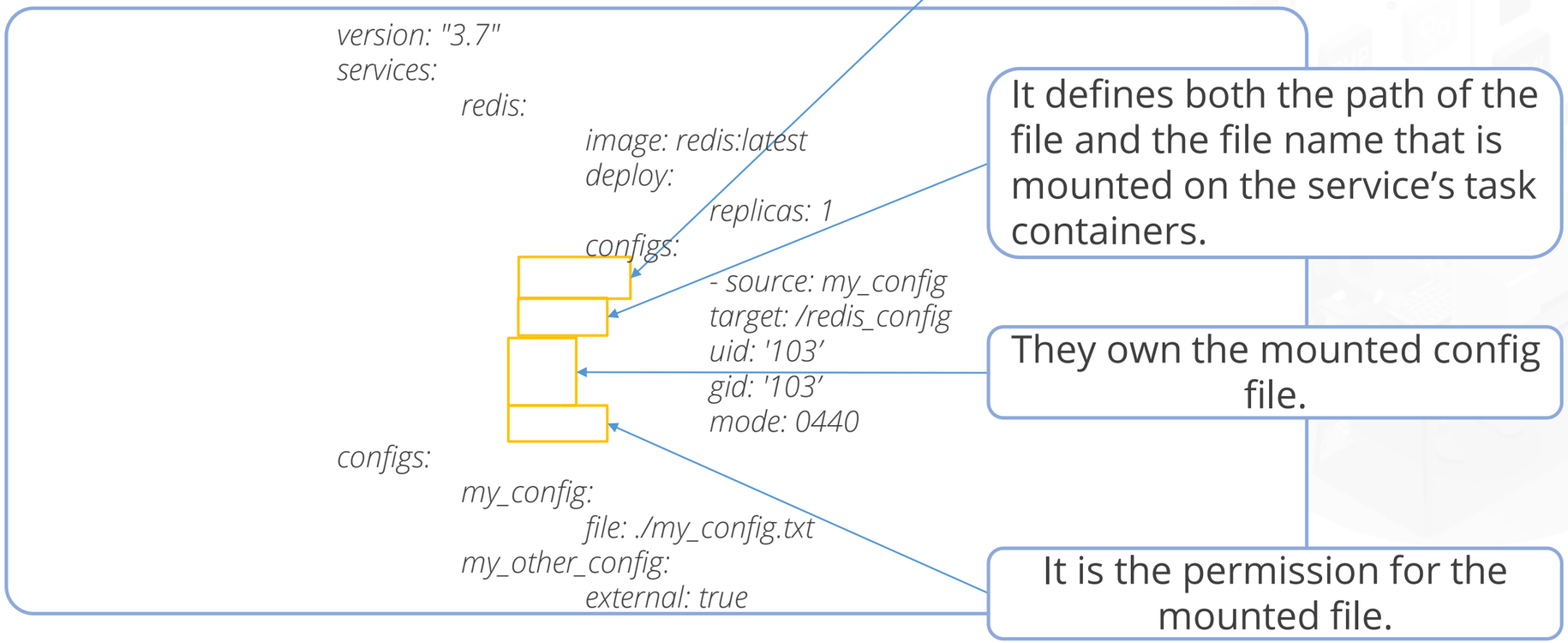
List of Configurations

build | cap_add, cap_drop | cgroup_parent | command | **configs** | container_name

depends_on | dns

SHORT SYNTAX

LONG SYNTAX



List of Configurations

build | cap_add, cap_drop | cgroup_parent | command | configs | container_name

depends_on | dns

```
container_name: my-web-container
```

List of Configurations

build | cap_add, cap_drop | cgroup_parent | command | configs | container_name

depends_on | dns

This expresses the dependency between services.

```
version: "3.7"
services:
        web:
                build: .
                depends_on:
                        - db
                        - redis
        redis:
                image: redis
        db:
                image: postgres
```

List of Configurations

build | cap_add, cap_drop | cgroup_parent | command | configs | container_name

depends_on | dns

Custom DNS servers can be portrayed as a value or a list.

dns: 8.8.8.8

dns:
- 8.8.8.8
- 9.9.9.9

List of Configurations

The remaining configurations are listed below:

Configuration	Options
deploy	*endpoint_mode* *lables* *mode* *placement* *replicas* *resources* *restart_policy* *rollback_config* *update_config*
devices	-
dns_search	-
entrypoint	-
env_file	-
environment	-
expose	-

List of Configurations

Configuration	Options
external_links	-
extra_hosts	-
healthcheck	-
image	-
init	-
isolation	-
labels	-
links	-
logging	-
network_mode	-
networks	*aliases* *ipv4_address, ipv6_address*
pid	-
ports	*Short syntax* *Long Syntax*
restart	-

List of Configurations

Configuration	Options
secret	*Short syntax* *Long syntax*
restart	-
secrets	*Short syntax* *Long syntax*
security_opt	-
stop_grace_period	-
stop_signal	-
sysctls	-
tmpfs	-
ulimits	-
userns_mode	-
volume	*Short syntax* *Long syntax*

Deploying Stack

Docker Stack Deploy

Command *docker stack deploy [OPTIONS] STACK* deploys a new stack or updates an existing stack.

Option	Description
--bundle-file	Provides a path to a Distributed Application Bundle file
--compose-file , -c	Provides a path to a Compose file, or "-" to read from stdin
--namespace	Allows to use Kubernetes namespace
--prune	Prunes services that are no longer referenced
--resolve-image	Queries the registry in order to resolve the image digest and supported platforms ("always"|"changed"|"never")
--with-registry-auth	Sends registry authentication details to Swarm agents
--kubeconfig	Provides a Kubernetes config file
--orchestrator	Allows the Orchestrator to use (swarm|kubernetes|all)

Note: In order to run this command, the client and daemon API must be of version 1.25.

Examples

Create services that form the stack using command *docker stack deploy:*

Command:

$ docker stack deploy --compose-file docker-compose.yml vossibility

It's the name of the Compose file.

Output:

```
Creating network vossibility_vossibility
Creating network vossibility_default
Creating service vossibility_nsqd
Creating service vossibility_logstash
Creating service vossibility_elasticsearch
Creating service vossibility_kibana
Creating service vossibility_ghollector
Creating service vossibility_lookupd
```

An output like this tells that the service has been deployed

Examples

Provide multiple *--compose-file* flags when the configuration is split between multiple Compose files:

Command:

$ docker stack deploy --compose-file docker-compose.yml -c docker-compose.prod.yml vossibility

These are the names of Compose files

Output:

```
Creating network vossibility_vossibility
Creating network vossibility_default
Creating service vossibility_nsqd
Creating service vossibility_logstash
Creating service vossibility_elasticsearch
Creating service vossibility_kibana
Creating service vossibility_ghollector
Creating service vossibility_lookupd
```

An output like this tells that the service has been deployed

Use command *$ docker service* to verify that the services were correctly created

Convert an Application Deployment into a Stack

Problem Statement: You have been asked to convert an application deployment into a stack using a file named docker-compose.yml with Docker stack deploy.

Steps to Perform:

1. Start the registry as a service and check its status with *curl*.
2. Create a project directory and create a project file in it.
3. Create a *.txt* file for requirements and Dockerfile to build a Docker image.
4. Create a docker-cofile mpose.yml and install docker-compose.
5. Push the application to a registry and deploy it.
6. List all the stacks that are a part of the created stack.

ASSISTED PRACTICE

Manipulate Services in Stack

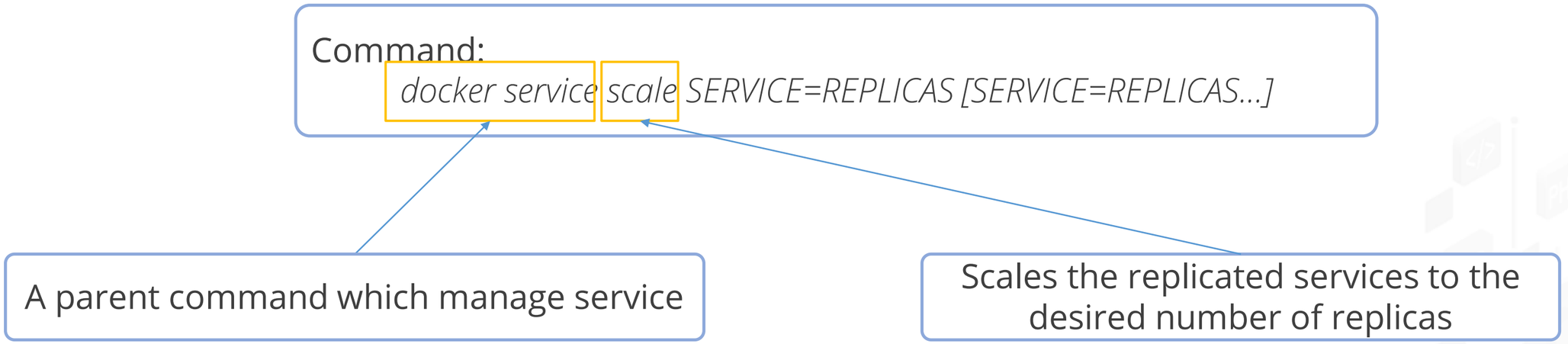
Manipulating Services

Following commands are used to manipulate the running services in a stack:

Command	Description
docker stack deploy	Deploys a new stack or updates an existing stack
docker stack ls	Lists stacks
docker stack ps	Lists the tasks in the stack
docker stack rm	Removes one or more stacks
docker stack services	Lists the services in the stack

Scaling

Scaling the Service



Option	Description
--detach , -d	This does not wait for the service to converge and exit immediately

Increase the Number of Replicas

Problem Statement: You have been asked to increase the replicas of a service so that multiple tasks can be assigned to different replicas.

Steps to Perform:

1. List the Docker services.
2. Scale the *redis* service to two tasks and the *registry* service to four tasks.
3. Use the *scale* command to scale both *redis* and *registry* services at the same time.
4. Check the actual number of replicas created.
5. Create a *global* service and try to scale it to 10 tasks.

ASSISTED PRACTICE

Persistent Storage

Persistent Storage

Persistent storage is a data storage device that stores data after the system has been shut down. It is also sometimes referred to as non-volatile storage.

- Common types of persistent storage are:
- Magnetic media, such as hard disk drives and tape.

- Persistent storage systems can be in the form of file, block, or object storage.

Persistent Storage

Best Practices of Persistence Storage:

Deduplicate full clones

Perform local caching

Store linked clones on solid-state drives

Persistent Storage

These are the four options that support persistent storage in Docker:

- Docker data volumes
- Data volume container
- Directory mounts
- Storage plugins

- Docker data volumes provide the ability to create a resource for persistent storage and retrieval of data in a container.
- Data volumes are a step forward from storing data within the container itself and offers better performance for the application.

Persistent Storage

- Docker data volumes
- Data volume container
- Directory mounts
- Storage plugins

- The storage of data volume is to use a dedicated container to host a volume and mount it to other containers
- A volume is a persistent data stored in /var/lib/docker/volumes/

Persistent Storage

- Docker data volumes
- Data volume container
- Directory mounts
- Storage plugins

- A persistent data mounts a local host and directory into a container.
- The volume and mount point is specified at the start time of the container on the **Docker run** command and it provides a directory within the container that the application can use.

Persistent Storage

- Docker data volumes
- Data volume container
- Directory mounts
- **Storage plugins**

- The most interesting development for persistent storage has been the ability to connect to external storage platforms through storage plugins.
- The plugin architecture provides an interface and API that allows storage vendors to build drivers to automate the creation.

Volume

Volume: Overview

It is a directory that is designated within one or more container which bypasses the Union File System.

Docker → secures → Volumes

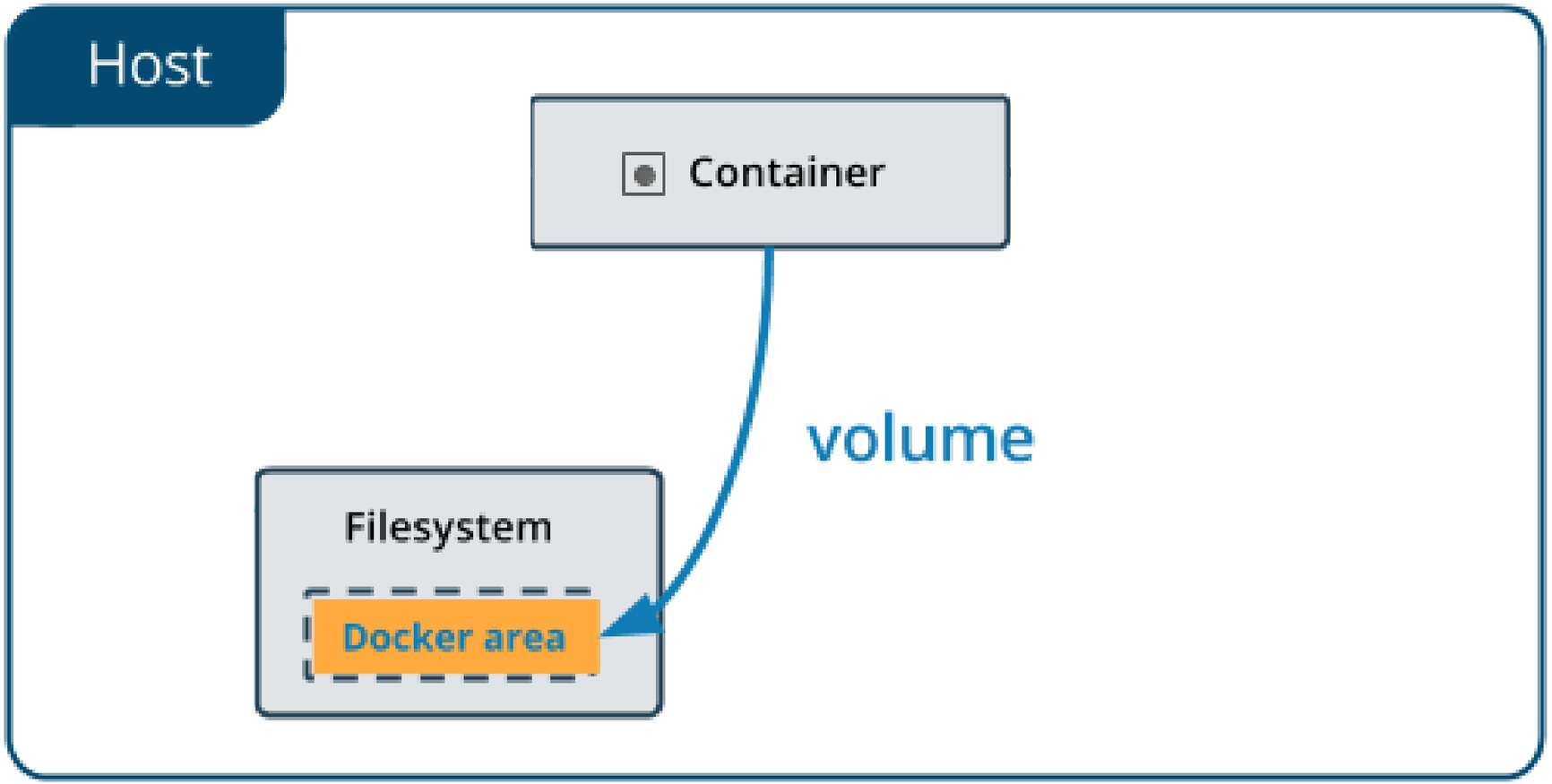
Docker → refrains from → Garbage collection

Types of volumes:

- Host
- Named
- Anonymous

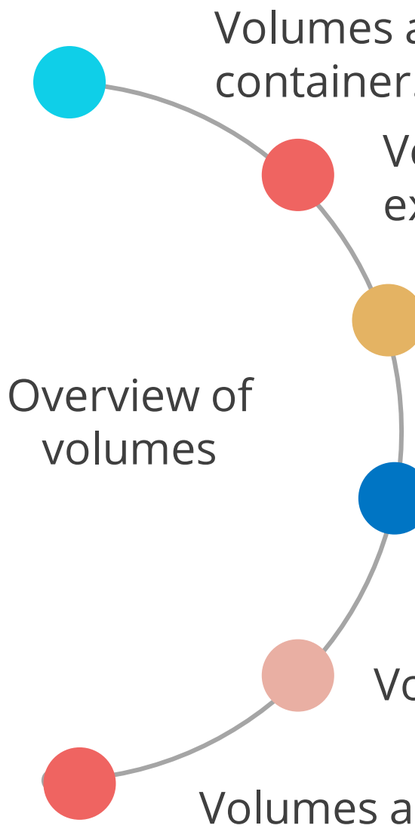
Volume: Overview

The contents of a volume exist outside the lifecycle of a given container.



Source: https://docs.docker.com/v17.09/engine/admin/volumes/volumes/

Volumes



Volumes are used to share files between a host system and the Docker container.

Volumes are directories that are outside the default Union File System and exist as normal directories and files on the host filesystem.

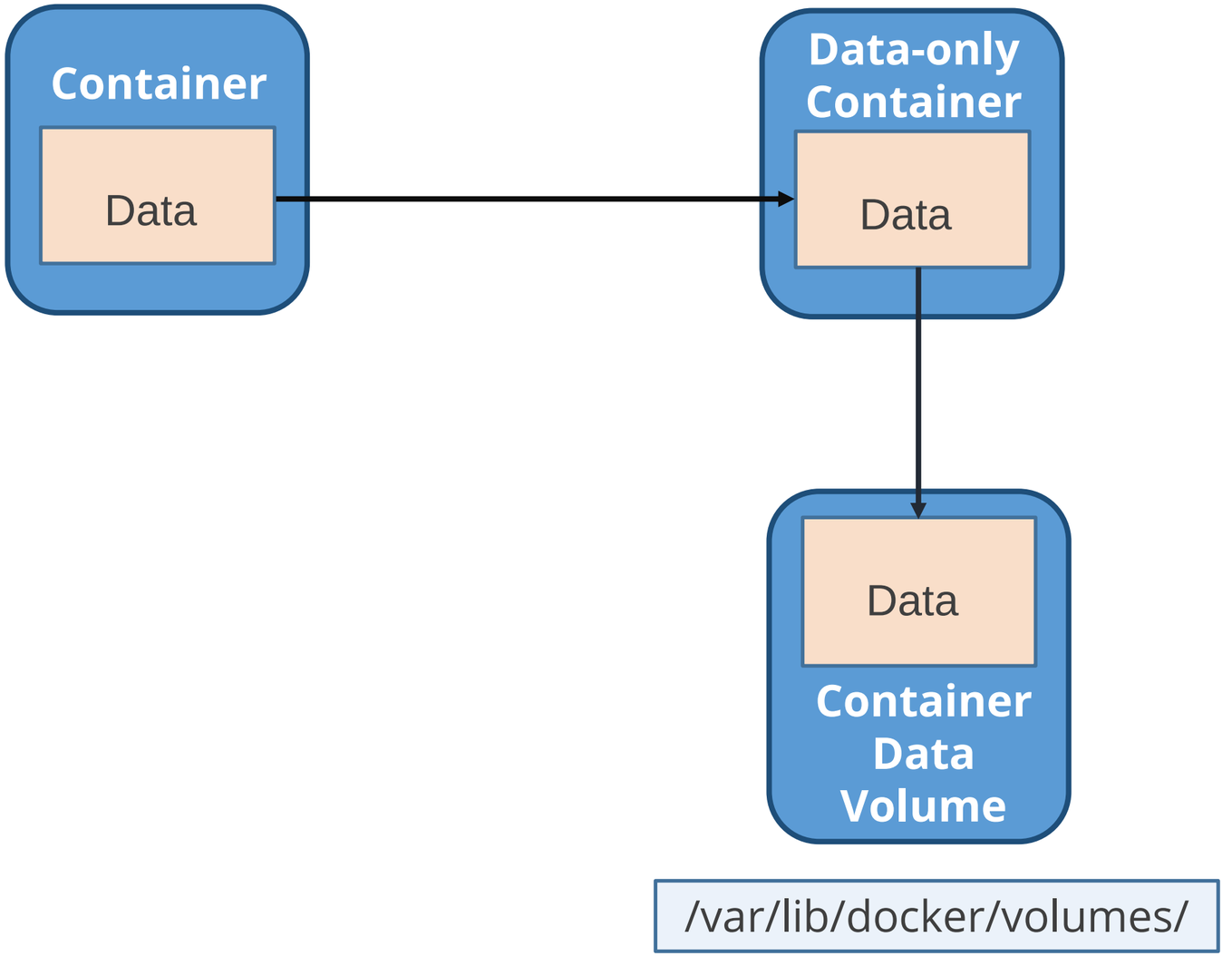
Volumes are the preferred mechanism for persisting data generated by and used by Docker containers.

Volumes are external storage areas used to store data produced by a Docker container.

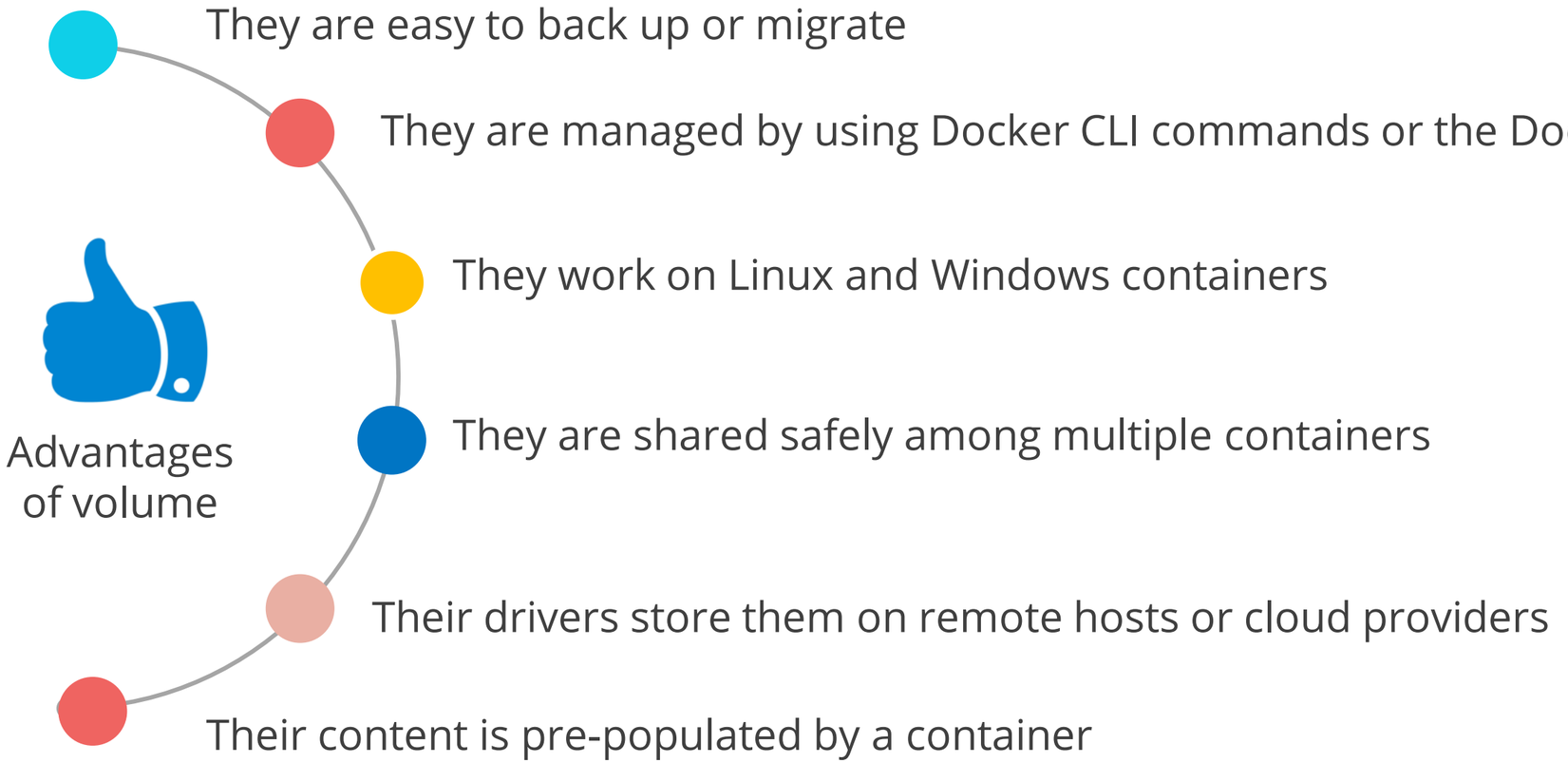
Volumes are located on the docker host or even on remote machines.

Volumes are used to store the data generated by a container. So even if the volume gets deleted, the data it produces remains.

Volumes



Volume: Advantages



Volume: Flags

Command:

Create a volume: *$ docker volume create my-vol*
List the volume: *$ docker volume ls*
Inspect the volume: *$ docker volume inspect my-vol*
Remove the volume: *$ docker volume rm my-vol*

Description of syntax "*-v*":
-v has fields:

- Volume field: It lists the name of the volume.
- Path field: It lists the path where the file mounts in the container.
- Read-only field: It represents the read-only volume.

All the fields are separated by a colon.

Volume: Flags

Description of syntax "*--mount*":

--mount has key-value pairs:

- Key= *type;* value= *volume*
- Key= *source;* value= The name of the volume, but the value is omitted in case of anonymous volume
- Key= *destination;* value= The path where the file mounts in the container
- Key= *readonly* and *volume-opt*

All the key-value pairs are separated by a comma.

Bind Mount

Bind Mount: Overview

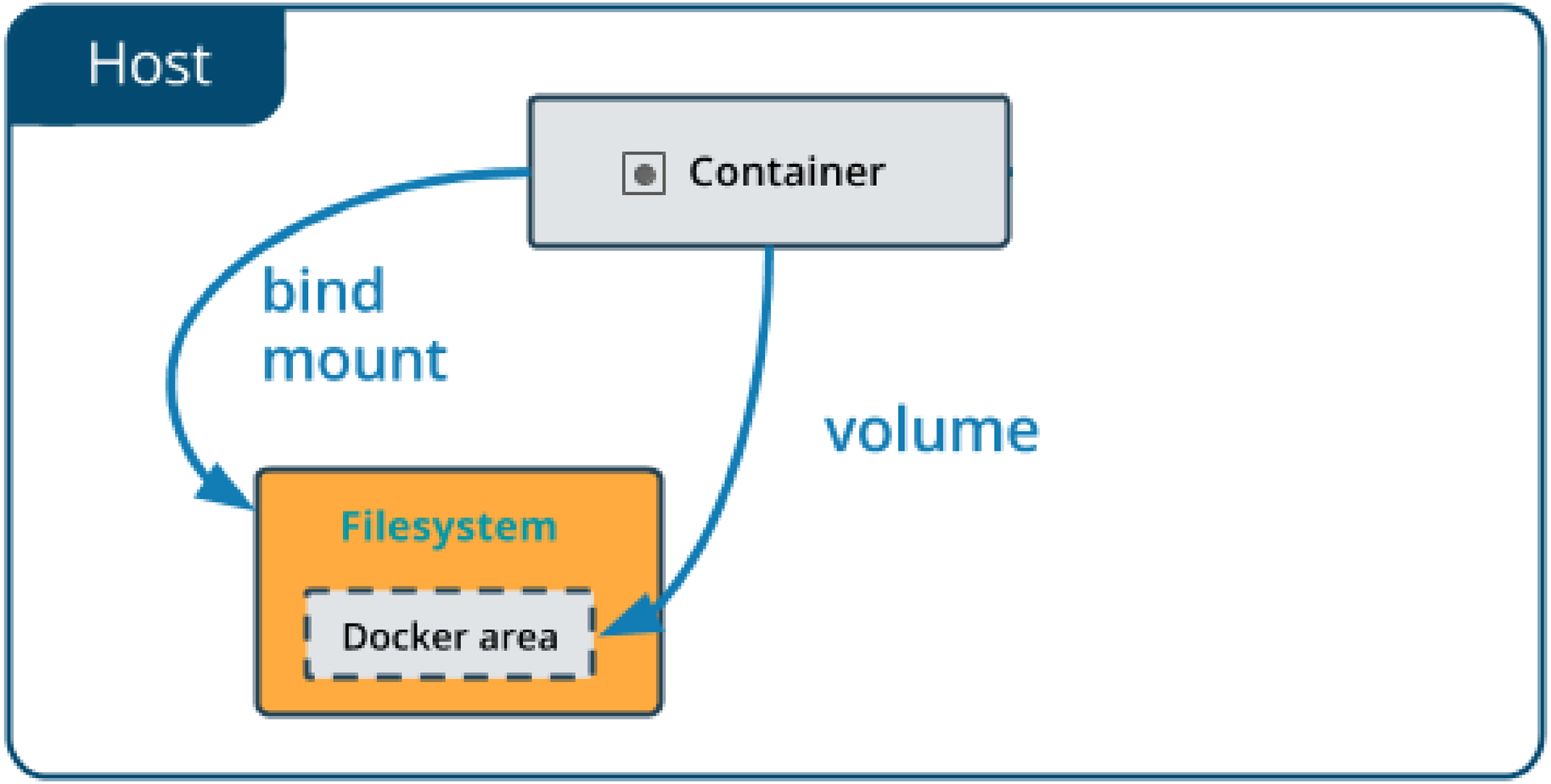
A file or directory is mounted from the host machine to the docker container.

Difference	
Bind mount	Volume
On the host machine, the file or a directory is referenced by its full or relative path.	On the host machine, a new directory is created within Docker's storage directory, and Docker manages that directory's content.

Note: Docker CLI commands cannot be used directly to manage bind mount.

Bind Mount: Overview

Bind mount relies on the host machine's filesystem with a specific directory structure.



Source: https://docs.docker.com/v17.09/engine/admin/volumes/volumes/

Bind Mount: Flags

Description of syntax "*-v*":
-v has the following fields:

- Path field: It lists the path to file on the host machine.
- Path field: It lists the path where the file mounts in the container.
- Optional field: It has lists of options, such as *ro, consistent, delegated,* and *cached.*

Note: All the fields are separated by a colon.

Description of syntax "*--mount*":
--mount have key-value pairs:
Key= *type;* value= *bind*
Key= *source;* value= Path to the file present on Docker daemon host
Key= *destination;* value= Path where the file mounts in the container
Key= *readonly, consistency,* and *bind-propagation*

Note: All the fields are separated by a comma.

Advantages of Volumes Over Bind Mounts

Advantages	Volume	Bind mount
Easy backup and migration	Yes	No
Easily managed by using Docker CLI command and the Docker API	Yes	No
Works on Linux as well as Windows	Yes	No
Shares safely among multiple containers	Yes	No
Encrypts the content of volumes	Yes	No
The contents of volume are prepopulated by a container	Yes	No

tmpfs Mount

tmpfs Mount: Overview

The tmpfs mount allows the container to create files outside the container's writable layer.

Features:

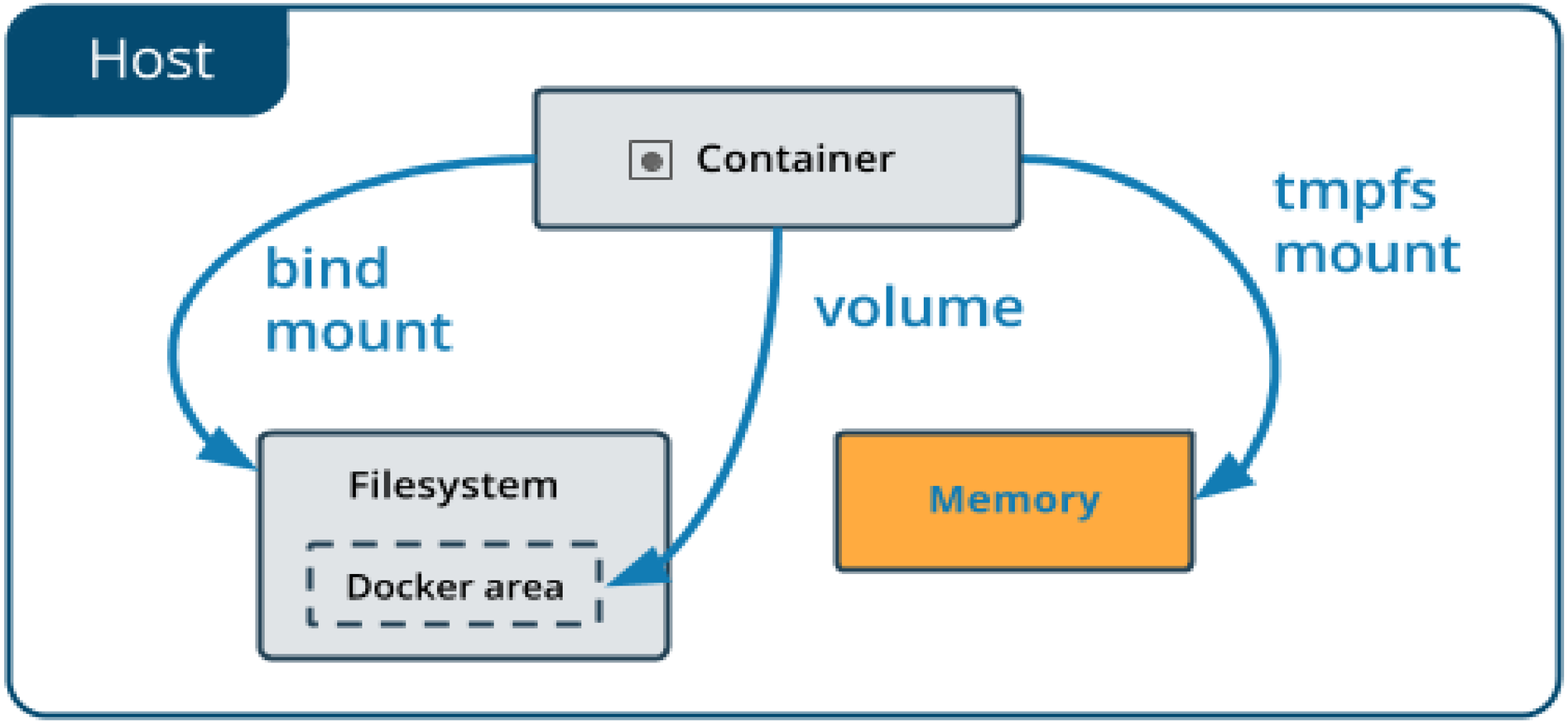
- tmpfs mounts are temporary
- tmpfs mounts persist in host memory

Limitations:

- The tmpfs mounts cannot be shared between containers.
- The tmpfs mounts are only available on Docker that runs on Linux.

tmpfs Mount: Overview

tmpfs mount is used to store the files temporarily.



Source: https://docs.docker.com/v17.09/engine/admin/volumes/volumes/

tmpfs Mount: Flags

Description of syntax "*--tmpfs*":

--tmpfs does not have configurable options
--tmpfs mounts the tmpfs mount

Description of syntax "*--mount*":

--mount has the following key-value pairs:
Key= *type;* value= *tmpfs*
Key= *destination;* value= Path where the tmpfs mount is mounted in the container
Key= *tmpfs-size* and *tmpfs-mode*

All the fields are separated by a comma.

Mount Volumes via Swarm Services

Problem Statement: You are asked to mount volumes in Docker via swarm services that can later be used to store files and back up the data.

Steps to Perform:

1. Create a Docker volume and start a container with it.
2. Create and use bind mounts.
3. Create a volume from *alpine* image.
4. Move inside */var/www/html*, create three files, and exit.
5. Back up the volume in Docker to a *.tar* file.

ASSISTED PRACTICE

Demonstrate How to Use Storage Across Cluster Nodes

Problem Statement: You and your colleague are working on an application and both of you want to use the same storage for your individual machines. Therefore, create a storage such that it can be used across cluster nodes.

Steps to Perform:

1. Create a volume for storage on the manager node.
2. Use the volume to create a replicated service.
3. Inspect worker nodes accessing the same volume for storage across cluster nodes.

ASSISTED PRACTICE

Prune Volumes

Prune Volumes

Volumes can be used by one or more containers and takes up space on the Docker host. Volumes are never removed automatically, because doing so could destroy data.

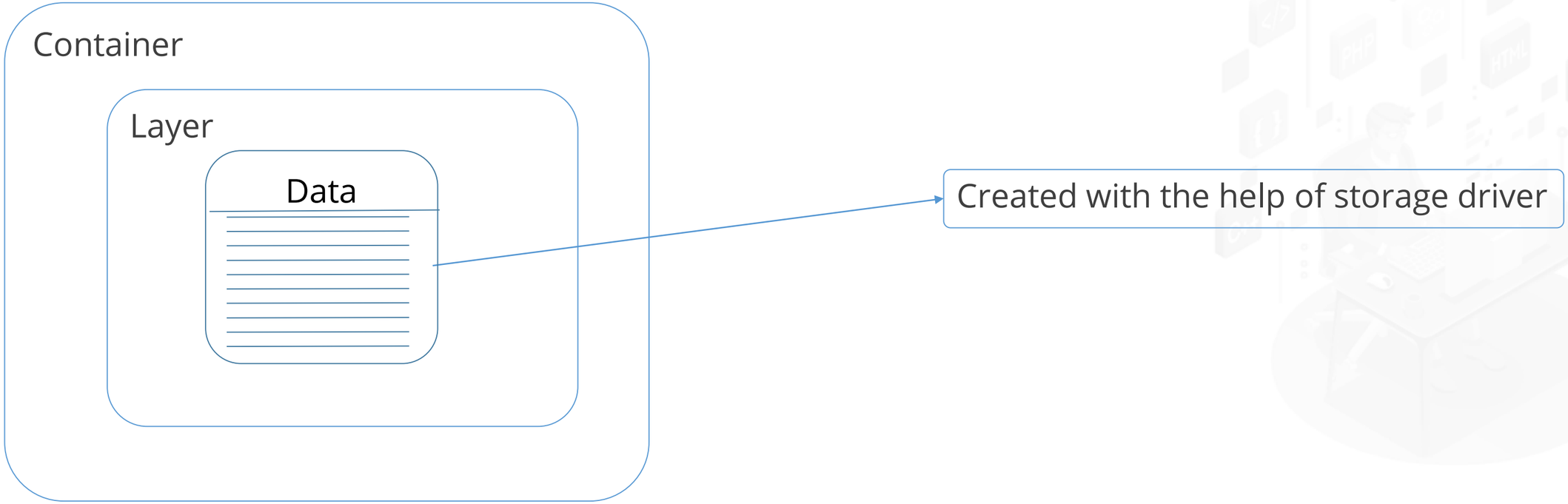
The user can use the following command to remove unused volumes:

```
$ docker volume prune
```

Storage Drivers

Docker Storage Drivers

The storage driver controls how the Docker host stores and manages files and containers. Docker uses a pluggable architecture to support many different storage drivers.



Docker Storage Drivers

Docker supports the following storage drivers:

Storage Drivers	Supporting backing filesystem
overlay2, overlay	xfs with ftype=1, ext4
aufs	xfs, ext4
devicemapper	direct-lvm
btrfs	btrfs
zfs	zfs
vfs	any filesystem

DeviceMapper

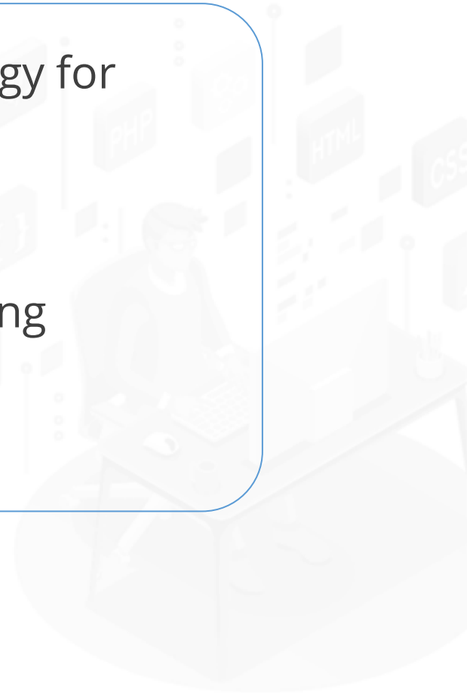
DeviceMapper

The devicemapper is a framework provided by the Linux kernel for mapping physical block devices onto higher-level virtual block devices.

DeviceMapper

- DeviceMapper is a kernel-based framework that supports several advanced technology for Linux volume management.

- Docker’s devicemapper storage driver leverages the thin provisioning and snapshotting capabilities of this framework for image and container management



DeviceMapper

Prerequisites:

- Devicemapper storage driver is a supported storage driver for Docker EE on many OS distributions.
- Devicemapper is also supported on Docker Engine, community running on CentOS, Fedora, Ubuntu, or Debian.
- If the user change the storage driver, the local system makes any container inaccessible that the user has already created.
- Device storage driver will let the user:
  - Use **docker save** to save containers
  - Push existing images to Docker Hub or a private repository, so that the user does not need to recreate them later

DeviceMapper

How does the devicemapper storage work?

1. The user needs to use the **lsblk** command to see the devices and their pools, from the operating system's point of view:

```
$ sudo lsblk

NAME                  MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
xvda                  202:0    0    8G  0 disk
└─xvda1               202:1    0    8G  0 part /
xvdf                  202:80   0  100G  0 disk
├─docker-thinpool_tmeta 253:0    0 1020M  0 lvm
│ └─docker-thinpool     253:2    0   95G  0 lvm
└─docker-thinpool_tdata 253:1    0   95G  0 lvm
  └─docker-thinpool     253:2    0   95G  0 lvm
```

DeviceMapper

How does the devicemapper storage work?

2. The user needs to use the **mount** command in order to see the mount-point which Docker is using:

```
$ mount |grep devicemapper
/dev/xvda1 on /var/lib/docker/devicemapper type xfs (rw,relatime,seclabel,attr2,inode64,noquota)
```

Note: When the user uses devicemapper, Docker stores image and layer contents in the thin pool, and exposes them to containers by mounting them under subdirectories of **/var/lib/docker/devicemapper/**.

DeviceMapper

Configure loop-lvm mode for testing

1. Stop Docker.

```
$ sudo systemctl stop docker
```

1. Edit/etc/docker/daemon.json. If it does not exist yet, create it. If the file is empty, add the following contents:

```
{
  "storage-driver": "devicemapper"
}
```

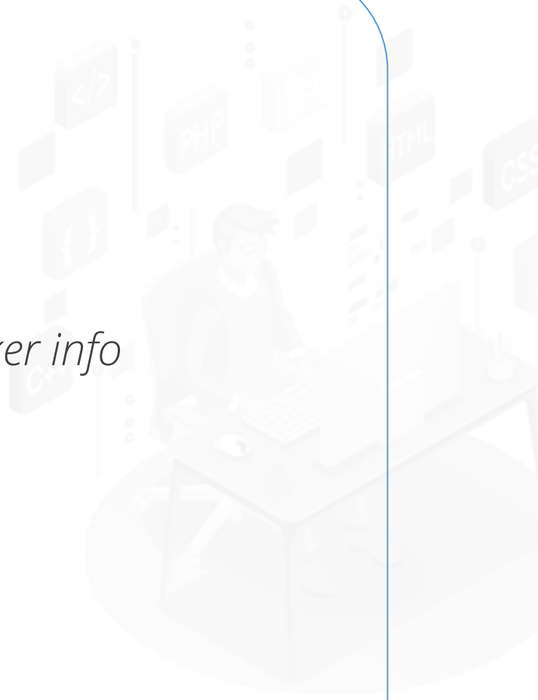
DeviceMapper

Configure loop-lvm mode for testing

3. Start Docker.

```
$ sudo systemctl start docker
```

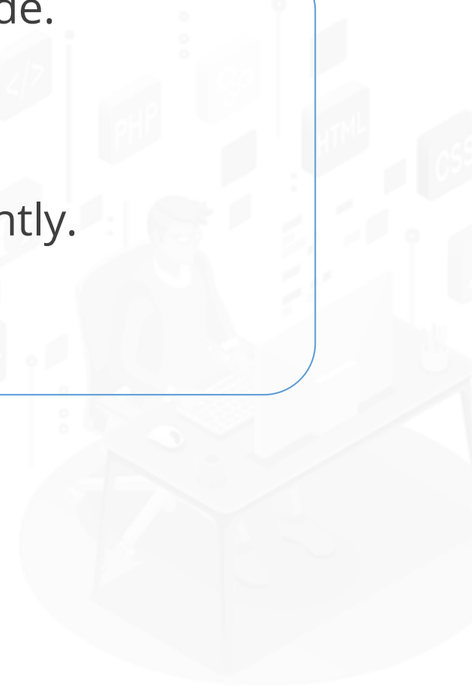
3. Verify that the daemon is using the Devicemapper storage driver. Use the *docker info* command and look for Storage Driver.



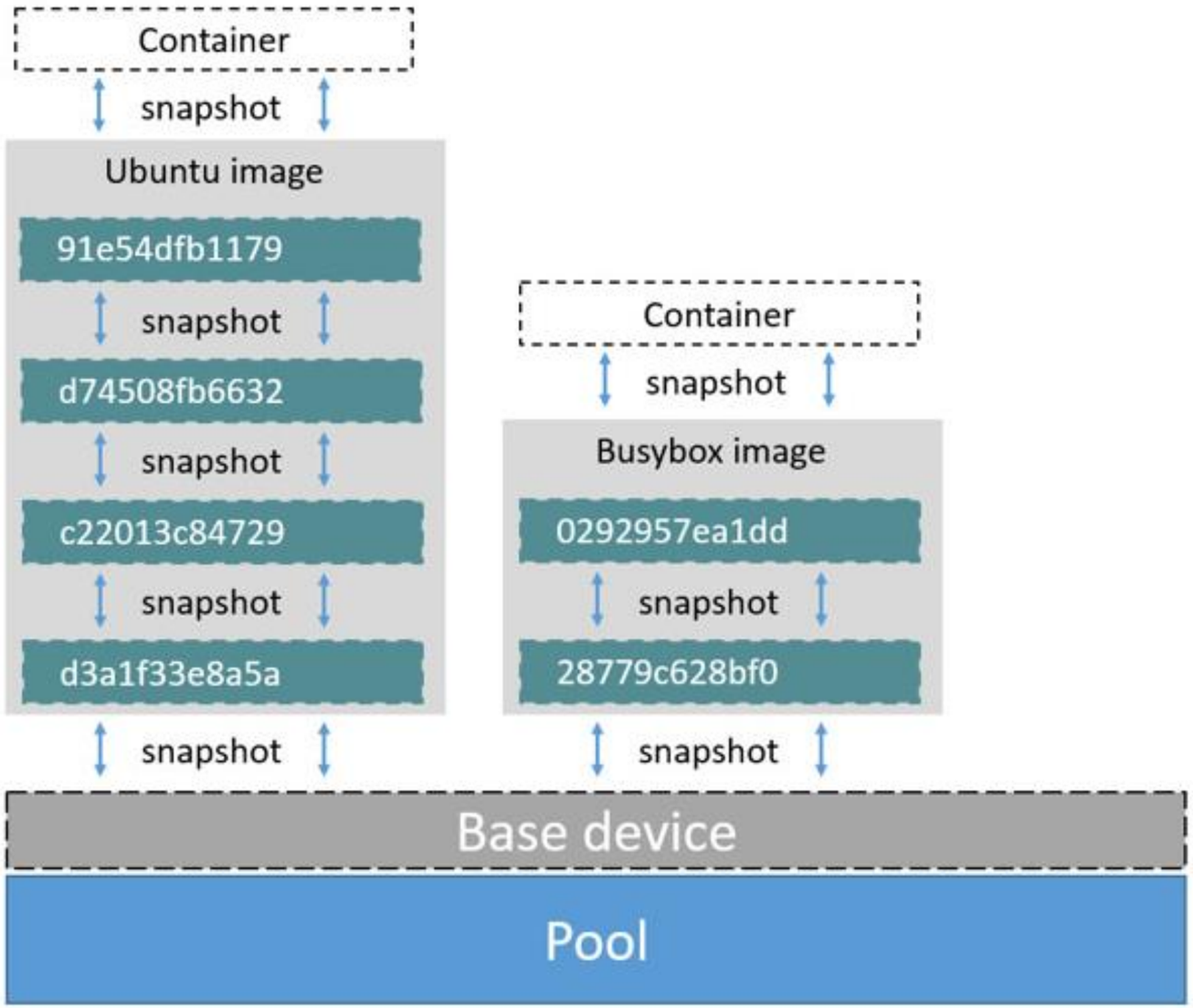
DeviceMapper

Configure direct-lvm mode for production

- Production hosts using the devicemapper storage driver must use **direct-lvm** mode.
- **direct-lvm** mode uses block devices to create the thin pool.
- This is faster than using loopback devices and uses system resources more efficiently.



Devicemapper Workflow



Source:: https://docs.docker.com/storage/storagedriver/device-mapper-driver/

Devicemapper and Docker Performance

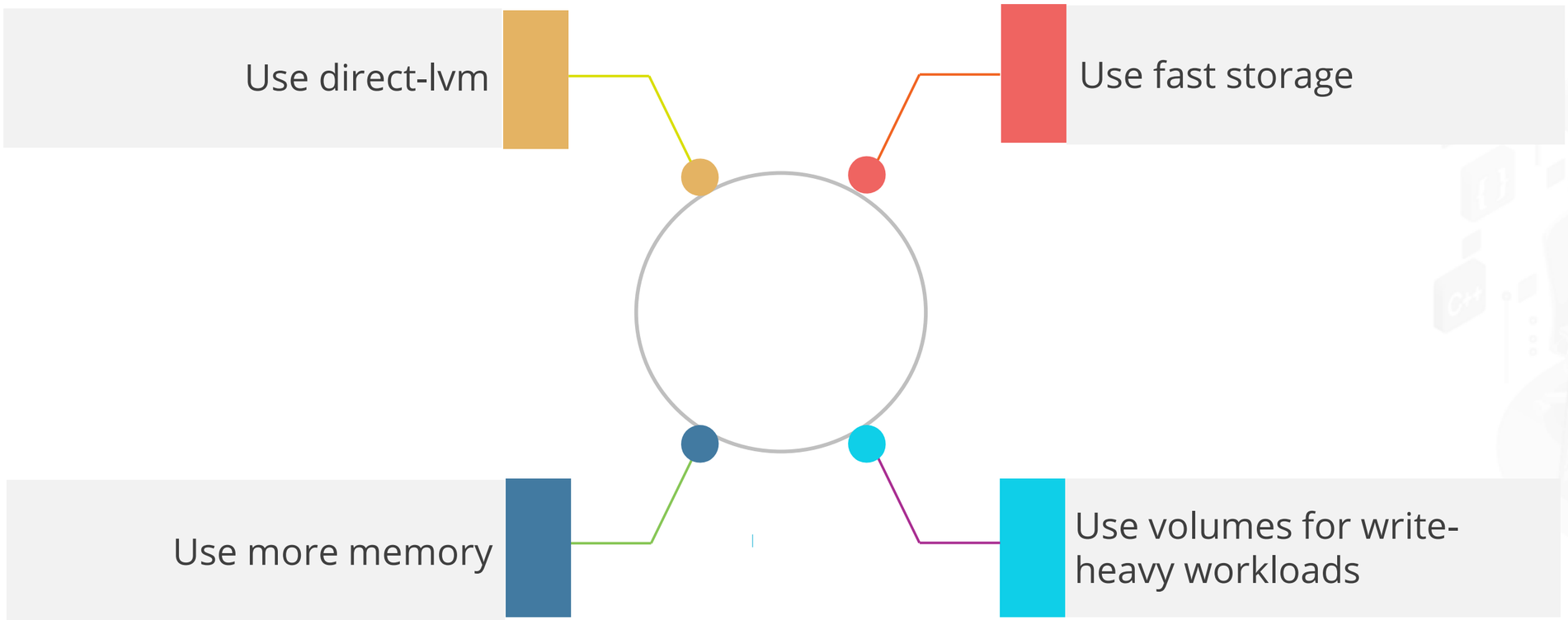
The Docker Performance creates impact on devicemapper. The Performance impacts are:

- **Allocate-on-demand** performance impact: An allocate-on-demand operation is used by the devicemapper storage driver to allocate new blocks into the writable layer of a container from the thin pool.

- **Copy-on-write** performance impact: The first time a container modifies a specific block, that block is written to the container's writable layer.

Device Mapper

Performance Best Practices



Select Storage Driver and Configure Device Mapper

Problem Statement: You are required to select a suitable storage driver according to your requirement and configure device mapper.

Steps to Perform:

1. Navigate to *daemon.json* file to change the default storage driver.
2. Restart *Docker daemon* and check the default storage driver.
3. Configure the device mapper by updating the *storage-driver* key in *daemon.json* file.
4. Restart *Docker daemon* again and check the storage driver value.

ASSISTED PRACTICE

Graph Driver

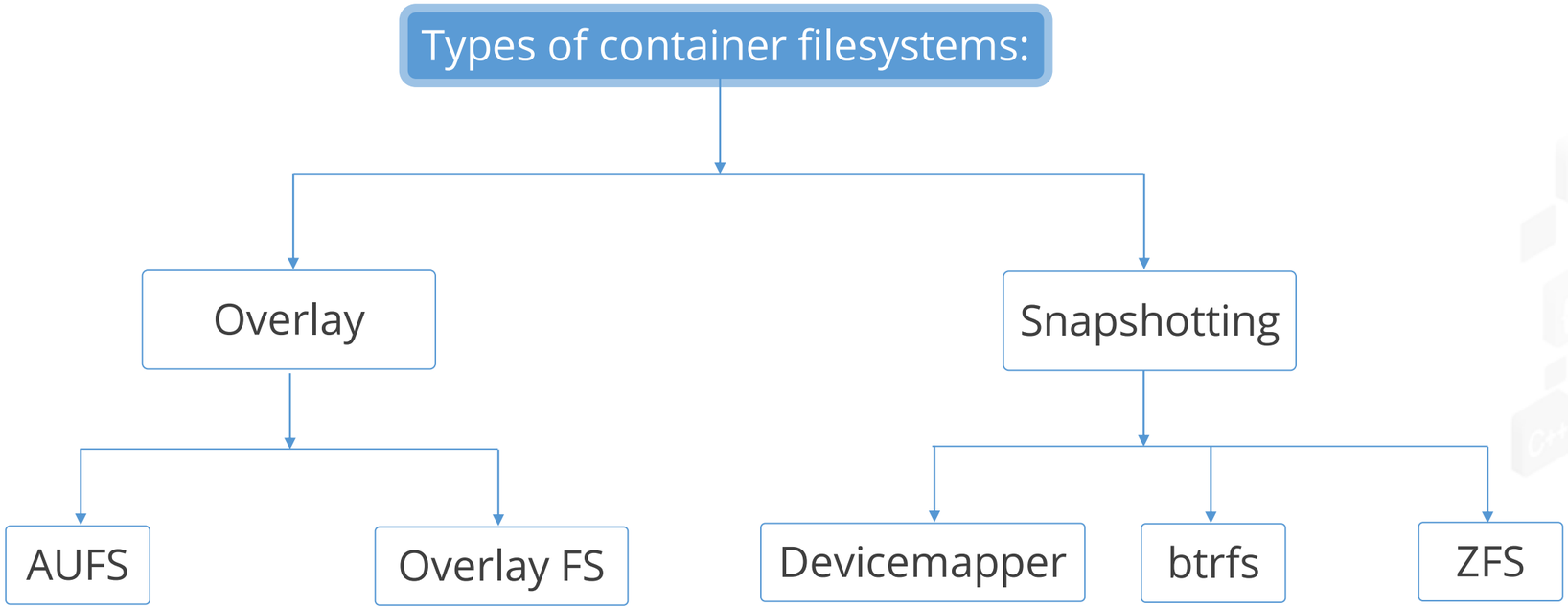
Graph Driver

Graph driver was introduced in order to provide Docker to a broader user base on a variety of distros. It was decided that filesystem support in Docker needs to be pluggable.

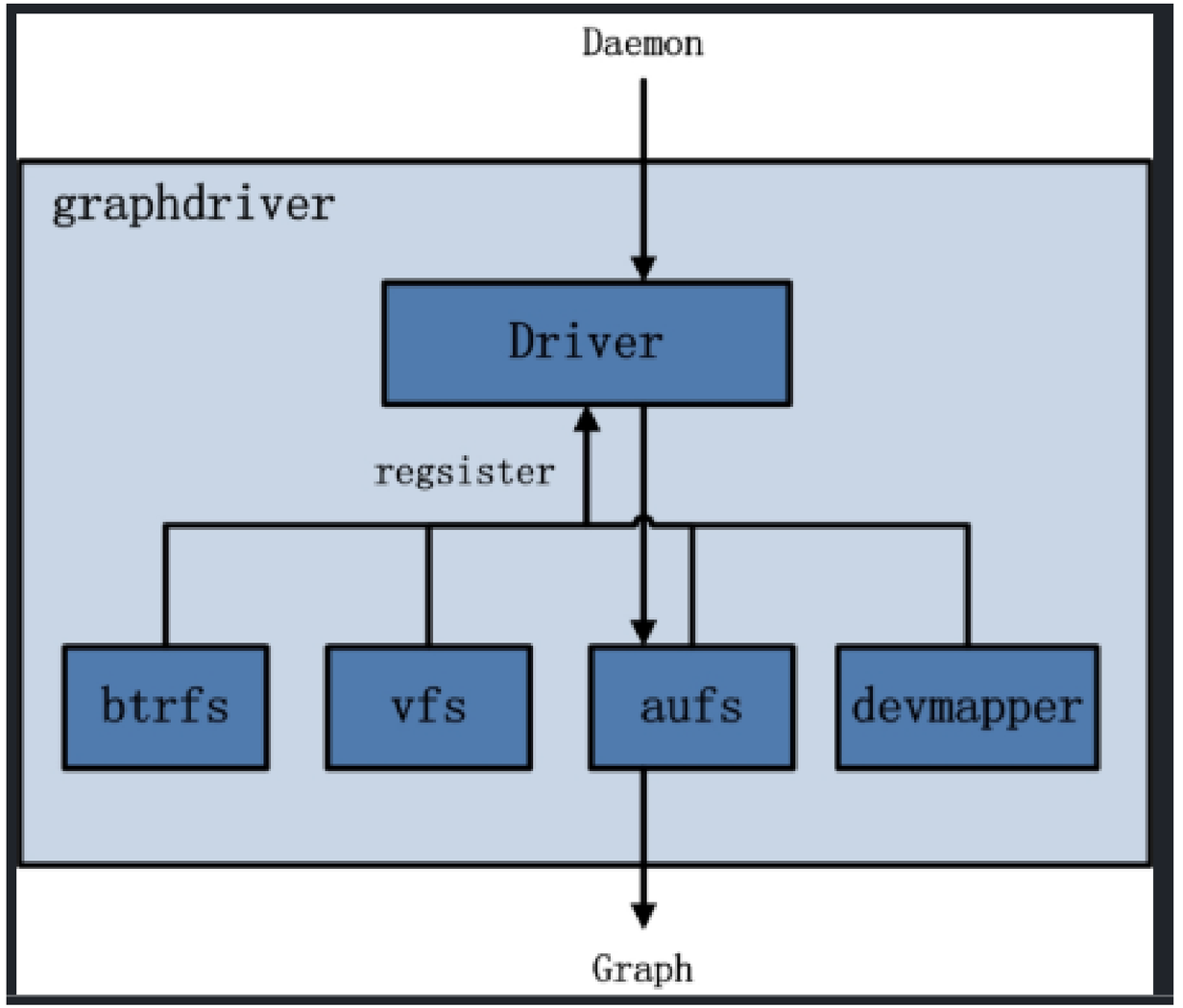
Graph driver supports:

- Build optimizations, such as caching and layer-sharing for faster build times
- Address content to ensure the filesystems were securely identified
- Change runtime requirements from LXC to runC

Graph Driver



Graph Driver



Source::https://www.google.com/imgres?imgurl=http%3A%2F%2Fblog.daocloud.io%2Fwp-content%2Fuploads%2F2014%2F12%2F005_graphdriver.jpg&imgrefurl=http%3A%2F%2Fprog3.com%2Fsbdm%2Fblog%2Fshlazww%2Farticle%2Fdetails%2F39178469&docid=WlQv0onf6gcXXM&tbnid=_w_oXWXwnzjsmM%3A&vet=10ahUKEwjngNqmpJbmAhUDeH0KHRDRDz8QMwhNKAQwBA..i&w=1105&h=955&bih=657&biw=1366&q=what%20is%20a%20graph%20driver%20in%20docker&ved=0ahUKEwjngNqmpJbmAhUDeH0KHRDRDz8QMwhNKAQwBA&iact=mrc&uact=8

Labels

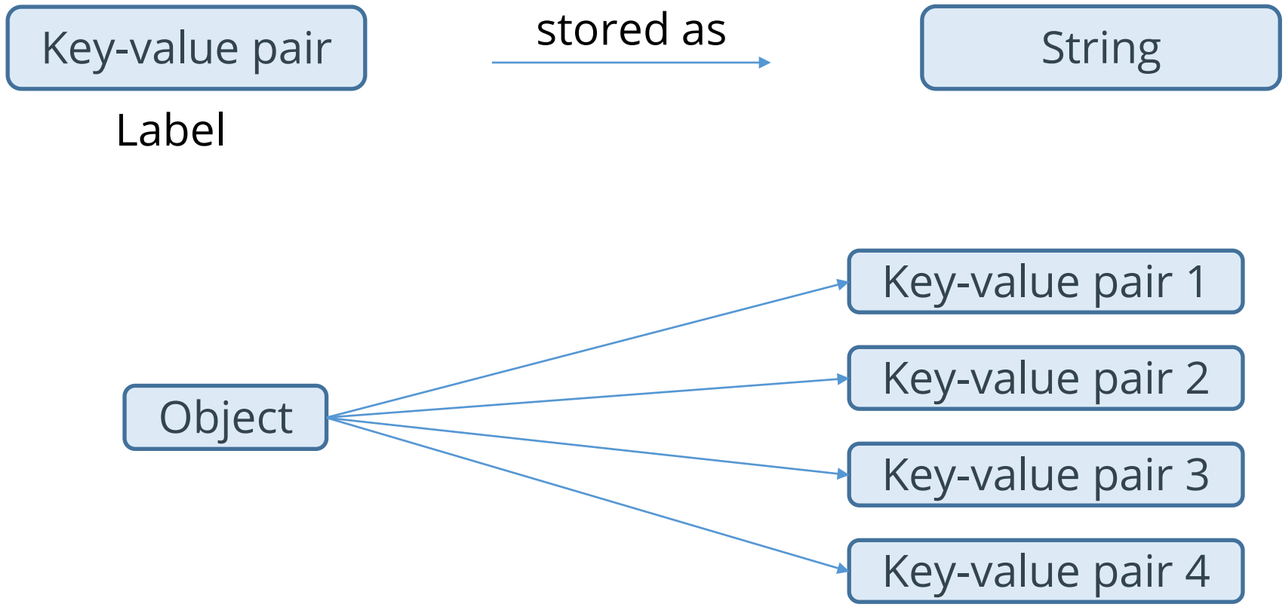
Label: Overview

It is a mechanism used for applying metadata to objects in Docker.

The Docker objects to which metadata is applied are:

- Images
- Containers
- Local daemons
- Volumes
- Networks
- Swarm nodes
- Swarm services

Label: Overview



Purpose of labeling:

- Organizing the images
- Recording the licensing information
- Annotating the relationships between containers, volumes, and networks

Label: Guidelines

Preventing duplication of labels:

Keys:

- Prefix each label key of the third-party tools with the reverse notation of the domain, such as *com.example.some-label*
- Refrain from using the domain in label key without the permission of the domain owner
- Refrain from using the namespaces which are reserved by Docker for internal use, such as *com.docker.**, *io.docker.**, and *org.dockerproject.**
- Begin and end the label keys with a lower-case letter and only use lower-case alphanumeric characters, period, and hyphen
- Refrain from using consecutive periods or hyphen
- Use period character (.) to separate the namespace *fields*

Values:

- Serialize the value to a string, for instance, use the *JSON.stringify()* JavaScript method to serialize JSON into a string
- Refrain from treating a JSON or XML document as a nested structure when querying or filtering by label value

Filtering Objects

Filtering flag: *-f* and *--filter*

Filters	Image	Container	Volume	Network	Swarm Node	Swarm Service
Dangling	Yes	-	Yes	-	-	-
Label	Yes	Yes	Yes	Yes	Yes	Yes
Before (Time)	Yes	Yes	-	-	-	-
Since (Time)	Yes	Yes	-	-	-	-
Driver	-	-	Yes	Yes	-	-
Name	-	Yes	Yes	Yes	Yes	Yes
Id	-	Yes	-	Yes	Yes	Yes
Scope	-	-	-	Yes	-	-
Type	-	-	-	Yes	-	-
Membership	-	-	-	-	Yes	-
Role	-	-	-	-	Yes	-
Mode	-	-	-	-	-	Yes

A complete list of filters for container object: *Label, before, since, name, id, exited, status, ancestor, volume, network, publish, expose, health, isolation,* and *is-task*

Apply Node Labels, Inspect the Labels, and Filter Swarm Nodes by Labels

Problem Statement: Your have been asked by your manager to add labels to the swarm cluster nodes so that you can inspect them later for specific labels and filter them based on labels.

Steps to Perform:

1. Apply labels to swarm cluster nodes.
2. Inspect the swarm nodes for labels.
3. Filter swarm nodes on the basis of labels.

ASSISTED PRACTICE

Quorum

Quorum of Managers

Manager nodes manage and maintain the swarm using the Raft Consensus Algorithm.

Quorum: It is formed by many managers that agree on updates proposed to the swarm.

Why does *Docker swarm mode* use a consensus algorithm?

The consensus algorithm makes sure that the same consistent state is stored by manager nodes which manage and schedule tasks in the cluster.

- Failures tolerated by Raft: **(N-1)/2**
- Members required in a quorum to agree on values proposed to cluster: **(N/2)+1**

Fault Tolerance

An odd number of managers must be maintained for fault tolerance.

How does an odd number of managers help during network partition?

This ensures that the quorum is available to process requests when the network is partitioned into two sets.

Swarm Size	Majority	Fault Tolerance
1	1	0
2	2	0
3	2	**1**
4	3	1
5	3	**2**
6	4	2
7	4	**3**
8	5	3
9	5	**4**

Recovering from Losing the Quorum

Loss of Quorum

What happens when swarm loses the quorum?

- The swarm tasks on the existing worker nodes will continue to run.
- After the loss of a quorum, the swarm nodes cannot be:

Added

Updated

Removed

- After the loss of a quorum, the new or existing tasks cannot be:

Started

Stopped

Moved

Updated

Quorum Recovery

The error encountered when an attempt is made to perform any management operation on the swarm after losing the quorum:

Error response from daemon: rpc error: code = 4 desc = context deadline exceeded

Different ways to recover a quorum:

- Bring the missing manager nodes back online
- Use the *--force-new-cluster* action from a manager node, that is:

  docker swarm init --force-new-cluster --advertise-addr node01:2377

Templates

Templates

Using the syntax provided by Go's template package, the templates are used for the following flags of *service create*:

- *--hostname*
- *--mount*
- *--env*

Placeholder	Description
.Service.ID	Service ID
.Service.Name	Service Name
.Service.Labels	Service Labels
.Node.ID	Node ID
.Node.Hostname	Node Hostname
.Task.ID	Task ID
.Task.Name	Task Name
.Task.Slot	Task Slot

The table above lists the description of different placeholders that are valid for Go template.

Usage of Templates

Problem Statement: You are asked to demonstrate the use of templates by creating a service from a template in docker.

Steps to Perform:

1. Set the template based on the service's name, node's ID, and the hostname.
2. Show the status of all the running tasks of the service created.
3. Inspect the service.

ASSISTED PRACTICE

Logs

Logs

Command	Description
docker logs	This command shows information logged by a running container.
docker service logs	This command shows information logged by all containers participating in a service.

Three streams of the Unix/Linux commands:

Stream	Description
STDIN	A *file handle*, which is read by the process to get information
STDOUT	A *file handle* on which information is written by the process
STDERR	A *file handle* on which the error information is written by the process

Logs

The following are the cases when the log command does not show useful information:

- The Docker logs may not show useful information when logging drivers are used to send logs.
- The application may send its output to the log files instead of *STDOUT* and *STDERR* when the image runs a non-interactive process.

Accessing Logs

Accessing the Docker container log requires:

1. Connecting to the Docker machine
2. Identifying the container ID
3. Examining the container logs
4. Investigating the Docker container logs

Accessing Logs

Log location on different operating systems:

Operating system	Location/Command
RHEL, Oracle Linux	*/var/log/messages*
Debian	*/var/log/daemon.log*
Ubuntu 16.04+, CentOS	Command: *journalctl -u docker.service*
Ubuntu 14.10-	*/var/log/upstart/docker.log*
macOS (Docker 18.01+)	*~/Library/Containers/com.docker.docker/Data/vms/0/console-ring*
macOS (Docker <18.01)	*~/Library/Containers/com.docker.docker/Data/com.docker.driver.amd64-linux/console-ring*
Windows	*AppData\Local*

Troubleshooting Services

Reasons for a service to go into a pending state:

- When all the nodes are drained
- When constraints are applied on the nodes
- When all the nodes in a swarm do not have enough memory to perform tasks

Steps for troubleshooting a service:

- Find all services using *docker service ls.*
- Check the service that is in pending state by using *docker service ps troub. troub* is the name of the service.
- Inspect the service tasks by using *docker inspect*.
- Find “Err” under “Status” to know the reason behind the pending state of the service.

Debugging

Swarm debugging:

Command:

docker node ls

Container and service debugging:

Commands for container:

docker container logs [OPTIONS] CONTAINER

docker container inspect [OPTIONS] CONTAINER [CONTAINER...]

Commands for service:

docker service logs [OPTIONS] SERVICE|TASK

docker service inspect [OPTIONS] SERVICE [SERVICE...]

Troubleshoot a Service that Is Unable to Deploy

Problem Statement: Your colleague is facing some issues while deploying a service on the swarm cluster. Help him in troubleshooting the service.

Steps to Perform:

1. Troubleshoot the running services.
2. Troubleshoot the tasks assigned for a service.
3. Troubleshoot the container associated with a service.

ASSISTED PRACTICE

Logging Drivers

Logging Drivers

Docker includes multiple logging mechanisms to assist users to obtain information from containers and services running. These mechanisms are called logging drivers.

Install the logging driver plugin

- To install, use **docker plugin install <org/image>** using the information provided by the plugin developer
- To list all installed plugins, use **docker plugin ls** and to inspect a specific plugin use **docker inspect**

Configure Logging Drivers

Configuration of the default logging drivers

1. Configure the Docker daemon to a specific logging driver:

- The user should set the log-driver value in the daemon.json file to the logging driver name.
- The file is 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"
}
```

Configure Logging Drivers

Configuration of the default logging drivers

2. The user can set the logging drivers in the **daemon.json** file as a JSON object with the key **log-opts** if they have configurable options.

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"
  }
}
```

Configure Logging Drivers

Configuration of the default logging drivers

3. If the user does not specify a logging driver, the default logging driver will be json-file. Thus, the default output for commands, such as docker inspect <CONTAINER> is JSON.

The user can find the current default logging driver for the Docker daemon, by:

- Running **docker info**
- Searching for **Logging Driver**

The user can use the following command on Linux, macOS, or PowerShell on Windows:

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

json-file
```

Configure Logging Drivers

Configuration of the logging driver for a container

1. The user can configure the container using different logging driver by using the **--log-driver flag** when the user starts a container.
   The user can set them using one or more instances of the **--log-opt <NAME>=<VALUE>** flag if the logging driver has configurable options.

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

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

Configure Logging Drivers

Configuration of the logging driver for a container

2. The user can 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
```

Configure Logging Drivers

Configuration of the delivery mode of log messages from container to log driver

Docker provides two modes for delivering messages from the container to the log driver:

- Direct blocking delivery from container to driver
- Non-blocking delivery stores log messages for driver consumption in an intermediate per-container ring buffer

The delivery mode for **non-blocking** messages prevents applications from blocking due to back pressure logging. When STDERR or STDOUT streams block, applications are under the risk of failing in unexpected ways.

Configure Logging Drivers

Field	Description
none	No availability of logs for the container and docker logs do not return any output
local	Stores logs in a custom format designed for minimal overhead
Json-file	Formats logs as JSON, the default logging driver for Docker
syslog	• Writes log messages to the syslog facility • Must be running on the host machine
journald	• Writes log messages to journald • Must be running on the host machine
gelf	Writes log messages to a Graylog Extended Log Format (GELF) endpoint, such as Graylog or Logstash

Configure Logging Drivers

Field	Description
fluentd	• Writes log messages to fluentd (forward input) • Must be running on the host machine
awslogs	Writes log messages to Amazon CloudWatch Logs
splunk	Writes log messages to splunk using the HTTP Event Collector
etwlogs	Writes log messages as Event Tracing for Windows (ETW) events
gcplogs	Writes log messages to Google Cloud Platform (GCP) Logging
logentries	Writes log messages to Rapid7 Logentries

Logging Drivers

Problem Statement: Your manager has asked you to configure the logging driver as per your requirement.

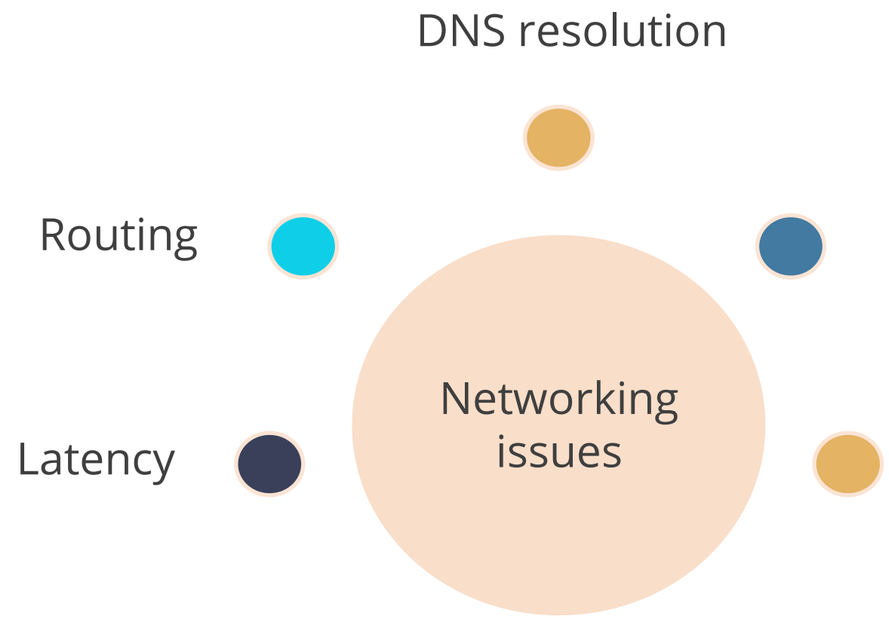
Steps to Perform:

1. Navigate to *daemon.json* and add *log driver* key to set default logging driver.
2. Restart the *Docker daemon* and check the default logging driver.
3. Configure the logging driver for a container in *run* command.
4. Configure the delivery mode of log messages by using *--mode* flag in run command.
5. Inspect a container for *log-config.*

ASSISTED PRACTICE

Network Troubleshooting

Networking Issues



Debugging

Network debugging:

Command:

docker network inspect [OPTIONS] NETWORK [NETWORK...]

netshoot container:

The *netshoot* container contains network troubleshooting tools to resolve Docker networking issues.

The *netshoot* container is able to:

- Get attached to any network
- Get placed in any network namespace of any container

Network Troubleshooting

Troubleshooting using *netshoot*:

- Container's Network Namespace
- Host's Network Namespace
- Network's Network Namespace

Problem: Networking issues with the application's container

Solution: Launch *netshoot* by using command

$ docker run -it --net container:<container_name> nicolaka/netshoot

Network Troubleshooting

Troubleshooting using *netshoot*:

- Container's Network Namespace
- **Host's Network Namespace**
- Network's Network Namespace

Problem: Networking issues on the host

Solution: Launch *netshoot* by using command

$ docker run -it --net host nicolaka/netshoot

Network Troubleshooting

Troubleshooting using *netshoot*:

- Container's Network Namespace
- Host's Network Namespace
- Network's Network Namespace

Problem: Networking issues on Docker network

Solution: Use *nsenter*

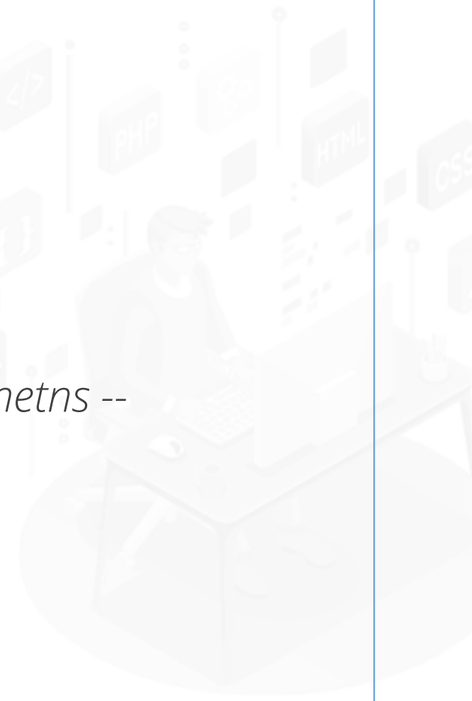
Network Troubleshooting

Troubleshooting using *netshoot*:

- Container's Network Namespace
- Host's Network Namespace
- Network's Network Namespace

Command for launching *netshoot* in privileged mode:

docker run -it --rm -v /var/run/docker/netns:/var/run/docker/netns --privileged=true nicolaka/netshoot



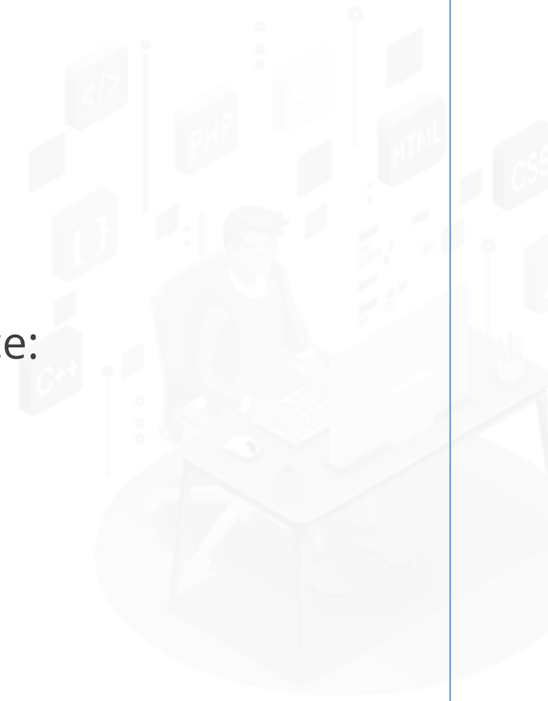
Network Troubleshooting

Troubleshooting using *netshoot*:

- Container's Network Namespace
- Host's Network Namespace
- Network's Network Namespace

Command for getting inside the network namespace:

nsenter –net /var/run/docker/netns/<networkid> sh



Network Troubleshooting

Troubleshooting using *netshoot*:

- Container's Network Namespace
- Host's Network Namespace
- Network's Network Namespace

Commands for assessing network after getting inside the network namespace:

ipconfig

brctl show

ip route show

ip –s neighbor show

ip –d link show

bridge fdb show

Troubleshooting Container Networking

Problem Statement: Your team lead wants you to troubleshoot containers and networks for issues by debugging containers and inspecting networks.

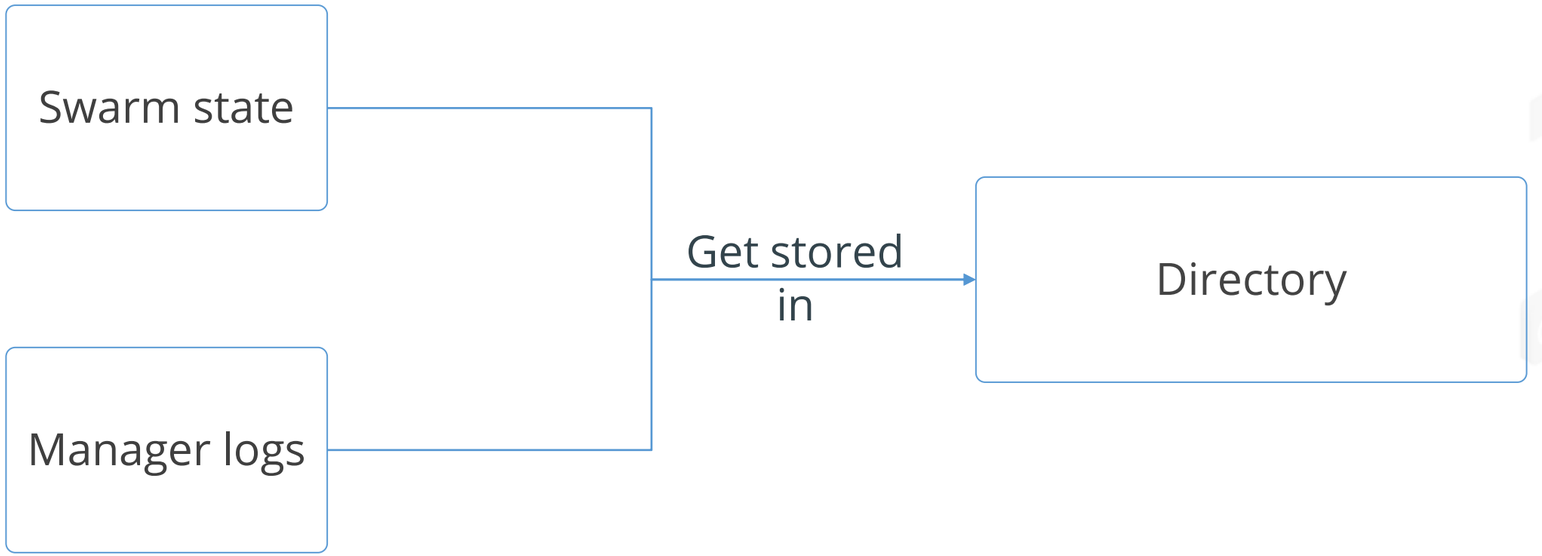
Steps to Perform:

1. Inspect a container and fetch the logs.
2. Inspect a network and perform container Debugging.
3. Use *nicolaka/netshoot* that contains all linux networking tools for container debugging.
4. Use *nsenter* to enter any namespace with netshoot running as a privileged container.

ASSISTED PRACTICE

Swarm: Backup

Swarm: Backup



Swarm: Backup

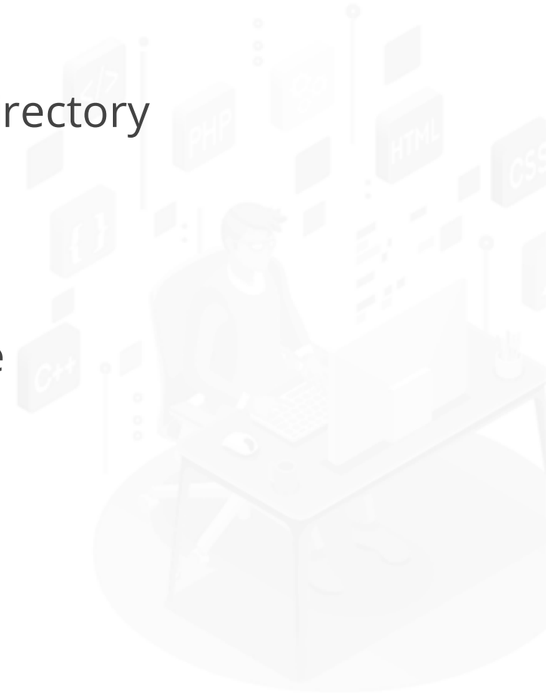
Procedure for swarm backup:

1. Retrieve the unlock key
2. Cease Docker on the manager
3. Back up the directory
4. Restart the manager

Swarm: Restore

Restoring the swarm backup:

1. Shutdown Docker
2. Remove the contents of the directory
3. Restore the directory along with the content
4. Start Docker on the new node
5. Check whether the swarm is in desired state by using *docker service ls*
6. Rotate the unlock key
7. Add manager and worker nodes
8. Restore the previous backup regimen



Set up a Backup Schedule

Problem Statement: Your manager wants you to set up a backup schedule so that the swarm data can be restored in case of an emergency.

Steps to Perform:

1. Stop the docker daemon on the manager node.
2. Create a backup by compressing the swarm folder.
3. Restore the backup in case of an emergency.

ASSISTED PRACTICE

Disaster Recovery

Swarm Disaster Recovery

Recover swarm from losing the quorum

Swarm is resilient to failures and can recover from any number of temporary node failures (machine reboots or crash with restart) or other transient errors.

These types of failures include data corruption, which are:

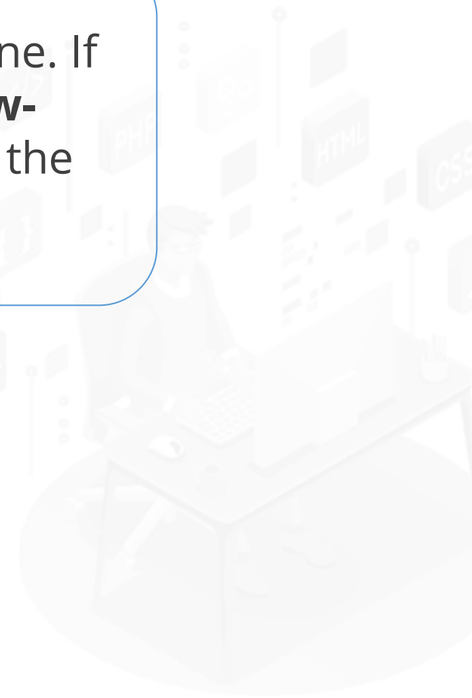
- If the user loses the quorum of managers, the user cannot administer the swarm.
- If the user has lost the quorum and attempts to perform any management operation on the swarm, an error occurs:

```
Error response from daemon: rpc error: code = 4 desc = context deadline exceeded
```

Swarm Disaster Recovery

Recover swarm from losing the quorum

The best way to recover from losing the quorum is to bring the failed nodes back online. If the user can't do that, the only way to recover from this state is to use the **--force-new-cluster** action from a manager node. This removes all managers except the manager the command was run from.



Key Takeaways

- One or more Docker Engines running together in a swarm mode forms a cluster. Swarm consists of multiple Docker hosts that act as either a manager or worker.
- A runtime instance of a docker image is known as a container. The way in which an application container runs in a swarm depends on the services.
- Volumes are not automatically deleted by Docker when the container is removed.
- The containers running on different Docker hosts communicate using the overlay network.

Dockerize a Legacy Application

Problem Statement: Your team is asked to dockerize a legacy application that has a Django application and a Postgres database. You must sync the Django application and the database so that the information stored can be accessed on demand. You are required to containerize the legacy system by creating docker images for these components using the Dockerfile. Connect these components using the docker-compose file.

Hint: This project requires a Dockerfile, a Python dependency file, and a docker compose.yml file.

Steps to Perform:

1. Create a docker image for Django application.
2. Set up a database connection for Postgres and Django.
3. Run docker-compose.yml to connect the Django and Postgres database image.

LESSON-END PROJECT

Running Replicated Service on a Swarm Cluster

Problem Statement:

Part 1: Your team has been repeatedly creating an image for a particular requirement in the production. To avoid multiple creations of the same file, your manager has asked you to create and push a docker image to the Docker Hub. The docker image is prepared from a Dockerfile that denotes the path of an index file.

Part 2: You are also asked to create a replicated service to check whether the tasks are run properly on different nodes of a swarm cluster.

Steps to Perform:

1. Create a docker image using the Dockerfile.
2. Tag the image and push it to the Docker Hub.
3. Create a swarm cluster and add manager and worker nodes to it.
4. Create a service using nginx image with three replicas to run them on three different nodes.

LESSON-END PROJECT