Docker Container

**Containers** are multiple isolated services that are run on a single control host (underlying infrastructure) and they access a single kernel.

***Key Benefits***

* **Improved performance**
* **portability**
* **eliminates environmental inconsistencies**

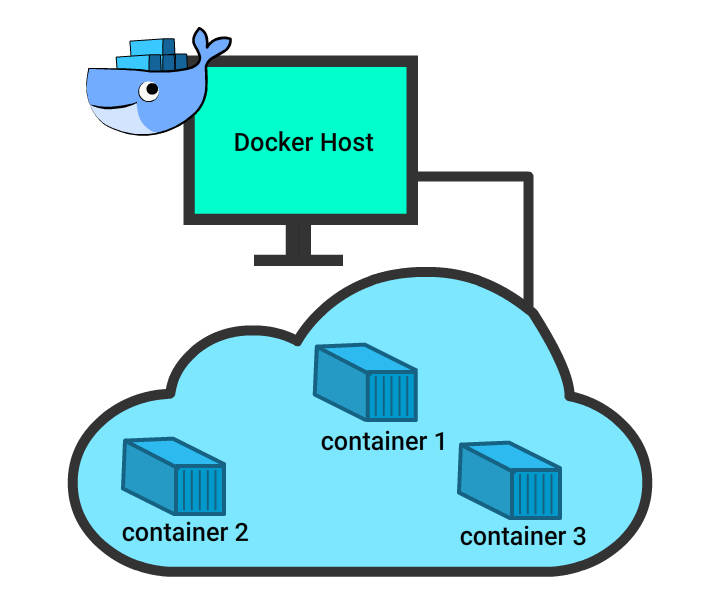
**Docker** is a **tool that uses open source Container technology** intended to make the process of **creating**, **deploying** and **running applications** easier by using **container based virtualization technology.**

##### Docker Components

Docker components include

* **Docker daemon** : Docker process responsible for *managing docker objects*
* **Docker client** : *Communicates with Docker daemon* through API calls
* **Docker Image** - *Read only template* that stores the application and environment.
* **Docker Container** - *Runtime instance* of a docker image
* **Docker File** - *Automates* Image construction
* **Docker registry** - Public and private repositories to *store images*

##### Docker Network



The concept of networking in Docker comes into picture when working with Docker in a **real time scenario at a large scale**.

**Docker Networking** helps us to **share data** across various containers.

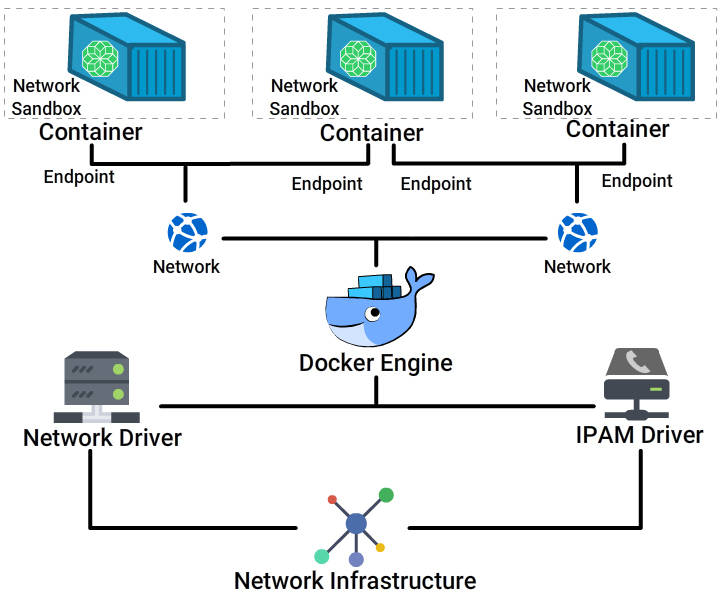
**Host and containers** in Docker are tied with **1:N relationship**, which means one host can command multiple containers.

Container Networking Model

* Container networking model is a standard defined for configuring network interfaces in Linux containers. This was proposed by Docker and later adopted by libnetwork (defines networking for docker) project.
* In this model, a ***set of network interfaces*** is assembled together to formalize the steps that are required to ***provide networking*** for containers.
* The ***fundamental goal*** of this model is to achieve ***application portability*** across various infrastructures.
* This model not only provides application portability but also utilizes the ***advantages of special features and capabilities of the infrastructure***.

2 of 9

Container Networking Model (CNM)



This block diagram represents the Container networking model.

Containers are ***interconnected*** to each other through a network established by connecting the ethernet port of the individual containers to the host system.

Every host system has a ***network infrastructure*** built in with a network driver and IPAM driver.

***Network driver*** - Device that enables network communication.

***IPAM Driver*** - Provides IP address management serives.

CNM core components include

* Sandbox
* Endpoint
* Network

##### CNM - Components

#### **Sandbox**

* Includes container's **network stack configuration**
* Manages **container's interfaces, routing table and DNS settings**
* Includes **multiple endpoints from multiple networks**

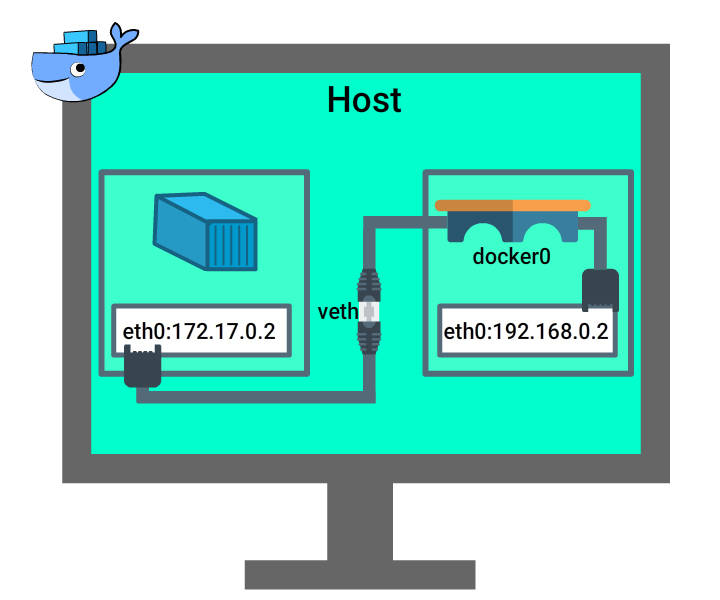
#### **Endpoint**

* Connects **Sandbox to Network**
* Abstracts **Network connection from the application**
* Provides **portability** to connect to different Network drivers

#### **Network**

* **Collection of endpoints** that can be connected to each other
* Implementation can be a **Linux bridge, VLAN** etc.

##### Behind the screens



#### **Docker Networking**

Refer to the image to understand how docker network is configured on a host machine.

##### Behind the screens

#### **Docker Networking**

* Every **regular host machine** (laptops/ Vm / Cloud machine) has an **ethernet interface**with an IP attached (eg. ip address 10.0.1.1).
* Once you install docker , within the host machine, **Docker creates a bridge**.
* Docker installs scripts that are clever enough to **interpret the networking on the host**and identify a space for its ip configuration.
* When you **start** a container, it **creates a virtual bridge called docker01**.
* **Default inet address** for Docker **172.17.42.1**.

Docker Native Network Drivers

Docker engine had ***in-built network drivers*** which are the core component that enables networking in containers.

Below are the different types of network drivers that can be used to ***configure networking in docker***.

* **Host:** Uses ***host's*** networking stack.
* **Bridge:** Creates a ***Linux Bridge on the host*** which is managed by Docker.
* **Overlay:** Creates an ***overlay network*** for ***multi-host*** networks.
* **MACVLAN:** Uses ***MACVLAN brige*** to connect container interfaces with parent host interface.
* **None:** Container has its ***own networking stack*** and completely ***isolated*** from the host network.

Network Modes

Various modes for networking is all about how we ***manage connections*** between containers using the ​network drivers.

* **Bridge mode Networking:** The ***default network*** will be bridge network.

Unless we specify the network option in docker run command, Docker daemon connects the container to this network.

* **Host Mode Networking:** Add the container to the network stack of the ***host***. Container in this mode will not create a new network configuration, where as ***share the network config of host***.

--net = host

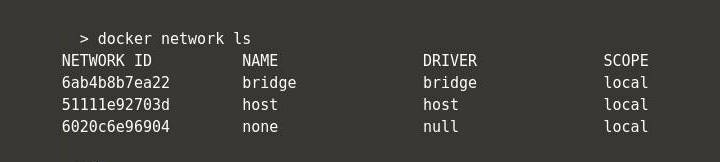
* **Container Mode Networking:** New container created will have the ***same config*** as that of the ***specified Container***.

--net=container:$comtainer2

* **No Networking:** Adds container to container network stack. Hence this ***lacks connectivity*** with the host.

--net=none

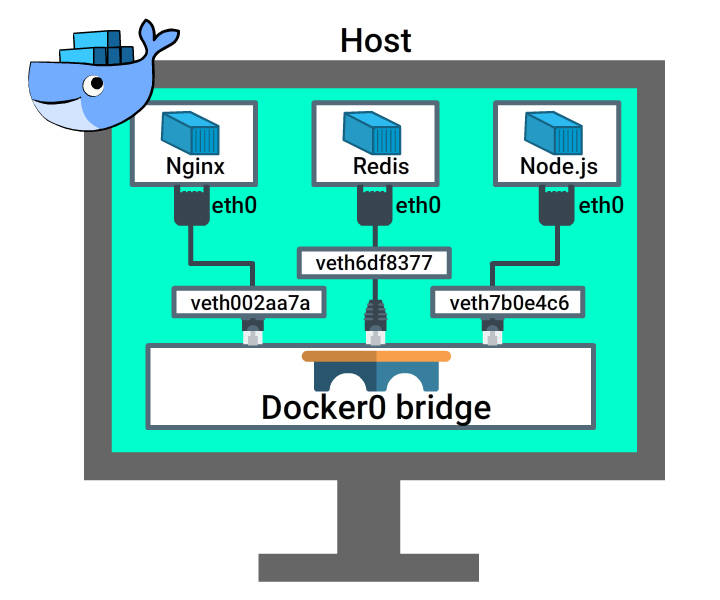
##### Network Drivers & Scope



On Docker installation, there are 3 networks that are automatically created.

This can be listed using command

docker network ls

This default network is **available in all docker hosts**. While creating containers, by default they get mapped to this bridge if you do not specify a different one.

Lets run **docker inspect** command to know more on this network.

docker network inspect bridge

This command lists

* Details on the **configured bridge network**
* **Containers running** in the network

##### Bridge Network (Contd1...)

Lets create couple of containers using below command. [Click here](https://www.katacoda.com/courses/docker/playground) to try this out in Katacoda Docker playground.

docker run -itd --name=container1 busybox

docker run -itd --name=container2 busybox

Now we have 2 containers running in the **default bridge network**.

Lets run **docker inspect** command.

docker network inspect bridge

Now we can see these containers listed under this bridge network.

***Disconnect*** from the bridge network

docker network disconnect bridge container1

##### User Defined Network

Users can create 2 kinds of network

* **Bridge network**
* **Overlay network**

Docker command to create a **Bridge network**

docker network create <<network name>>

e.g. docker network create myNetwork

##### Create a User Defined Network

Lets create our new bridge network and learn how to connect containers to the network.

Step 1:

docker network create -d bridge new\_bridge

-d --> to specify the driver type (in this case its is bridge network)

Step 2:

Verify if the new network is created

docker network ls

This command will list the new bridge along with the other default bridges.

Run the below command to inspect the new network created.

docker network inspect new\_bridge

Step 3:

Add container to the network

Lets create a new container using image training/postgres to run a PostgreSQL db and tag this to the new network created.

docker run -d --net=new\_bridge --name db training/postgres

Step 4: Verify if its connected to the new network.

Run the below command to get the network details on where this container is connected.

docker inspect --format='{{json .NetworkSettings.Networks}}' db

##### Create a User Defined Network (Contd 2...)

Step 5:

Lets run the python web app using image training/webapp without specifying any network settings.

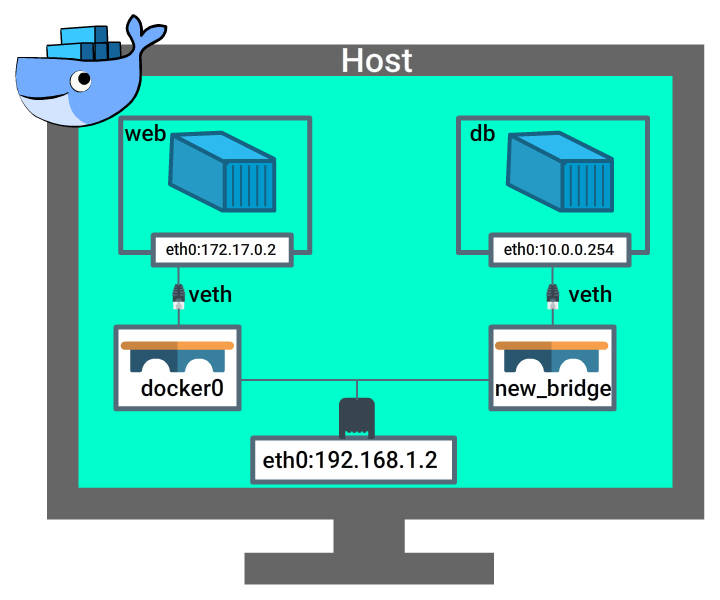
This connects the container to the default bridge network.

docker run -d --name web training/webapp python app.py

Step 6:

Verify if the web app container is connected to the default bridge network.

docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' web



##### Create a User Defined Network (Contd 4...)

Step 7:

Check Connectivity between containers.

Lets try to ping the web app container from the db container using the below command.

docker exec -it db bash

Now lets enter the ping command in the db running container.

ping 172.17.0.2

Note: ip address should be as that of the web app container. Refer to Step 6 to get the ip address.

Press CTRL-C to end this ping and you will find that the ping command failed. This is because the web app and the db container is not connected to the same network.

##### Create a User Defined Network (Contd 5...)

Step 8:

Connect web container to new\_bridge.

You can open an additional terminal in Katacoda playground. Let's execute the below commands in this new terminal.

Let's now connect web container new\_bridge network.

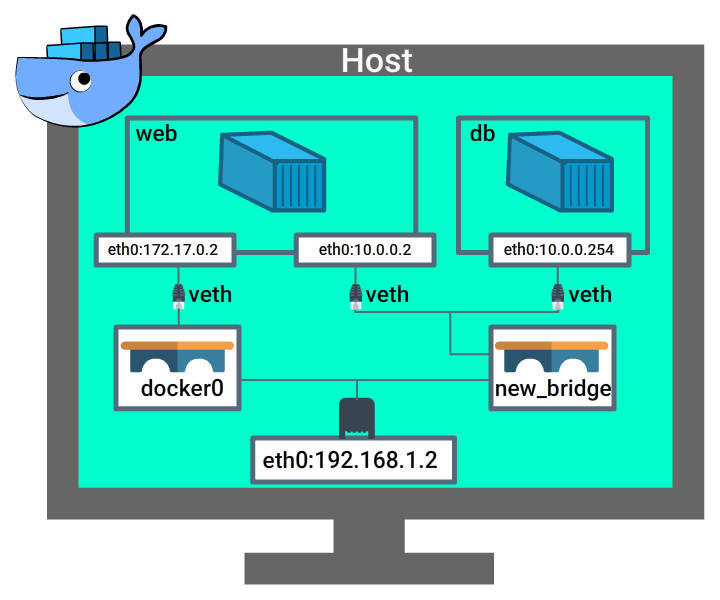
docker network connect new\_bridge web

Step 9:

Repeat Step 7 commands to ping web container from db container. Get the ipaddress by running below command again. This time you will find 2 ipaddress, the second one is for the connectivity with the new\_bridge.

docker inspect --format='{{json .NetworkSettings.Networks.new\_bridge}}' web

This ping command is successful now since we have established connectivity between 2 containers.



Refer to this image for the overview of the networking that we have done so far.

##### Network Commands

[Click here](https://www.katacoda.com/courses/docker/playground) to try this out in Katacoda Docker playground.

#### **Create Network**

Command to **create new bridge network**.

docker network create <<network name>>

e.g docker network create myNetwork

#### Connect Container to Network

docker network connect <<network name>> <<Container name>>

e.g. Create a container busybox and connect this to myNetwork.

docker run -itd --name=container1 busybox

docker network connect myNetwork container1

##### Network Commands (Contd 1...)

#### **List all Networks**

docker network ls

#### **Inspect a Network**

We can inspect a network to list the configuration and containers connected to using command

docker network inspect <<network name>>

e.g.

docker network inspect myNetwork

#### **Disconnect a container from a Network**

docker network disconnect <<network name>> <<container name>>

e.g.

docker network disconnect isolated\_nw container5

##### Network Commands (Contd 2...)

#### **Remove network**

Command to remove unused network

docker network rm <<network name>>

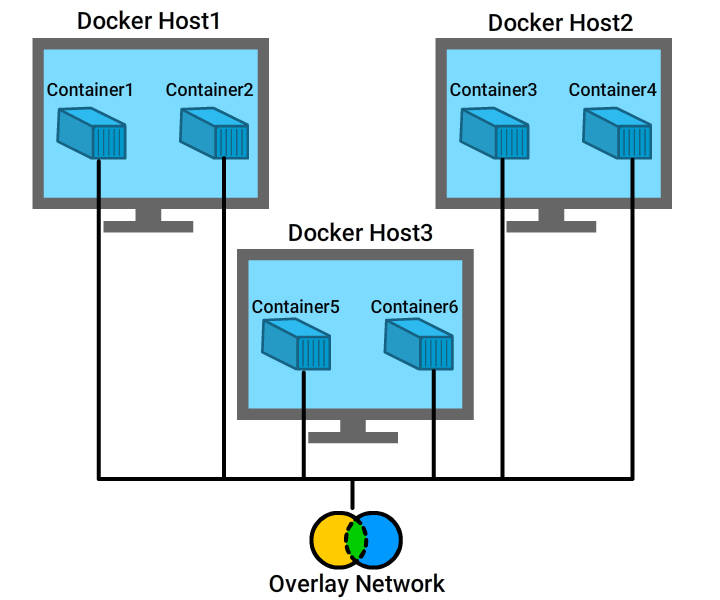
e.g. docker network rm Mynetwork

##### Multi-Host Networking

Docker Engine provides **out-of-the-box** support to **multi-host networking** using **overlay network driver**.

A **bridge network** is used when we run a relatively **small network** on a **single host**.

An **overlay network** is used when we have a significantly **larger network** involving **multiple host**.



Refer to the Overlay network block diagram. This network connects multiple containers from different hosts.

##### Overlay Network

There are 2 ways of creating an overlay network.

* Overlay networking with an **external key-value store**
* Overlay networking in **swarm mode**

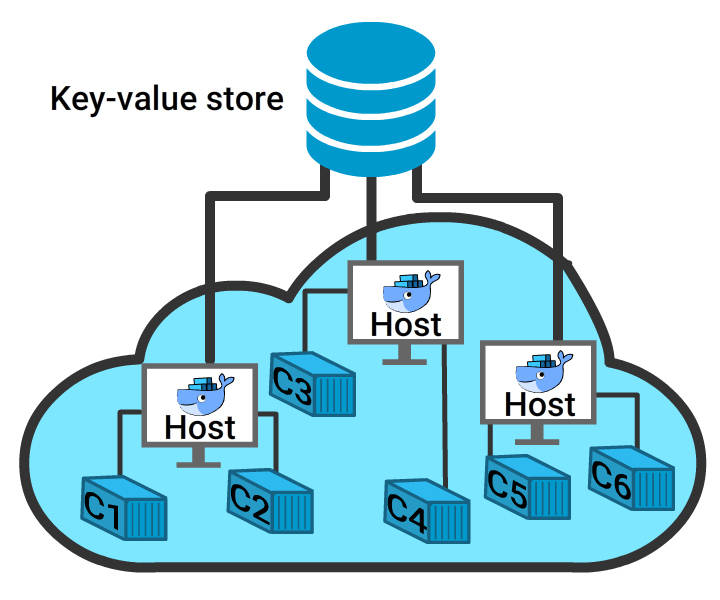
##### Overlay networking with an external key-value store

We need a valid **key-value store service** to create a Overlay network.

Before creating a network in this way, you must **install and configure** your chosen key-value store service.

Pre-requisites:

* A key-value store (Docker supports **Consul, Etcd, and ZooKeeper**)
* A **cluster of hosts** that connects to key-value store
* Host with Docker Engine **configured properly**
* Host within cluster must have a **unique** hostname because key-value store uses host name to **identify cluster members**



In this block diagram, containers from various host systems communicate through key-value store.

##### Overlay networks in swarm mode

What is Swarm?

A **swarm** is a cluster of **Docker engines, or nodes**, where you deploy services.

The **cluster management** and **orchestration** features embedded in the Docker Engine are built using **SwarmKit**.

Docker engines participating in a cluster are running in **swarm mode**.

Source: docker.com

You can initialize a swarm or join an already existing swarm.

##### Overlay Network in Swarm Mode

The swarm nodes exchange overlay network information using a **gossip protocol**.

By default the nodes **encrypt and authenticate** information they exchange via gossip using the **AES algorithm in GCM mode**.

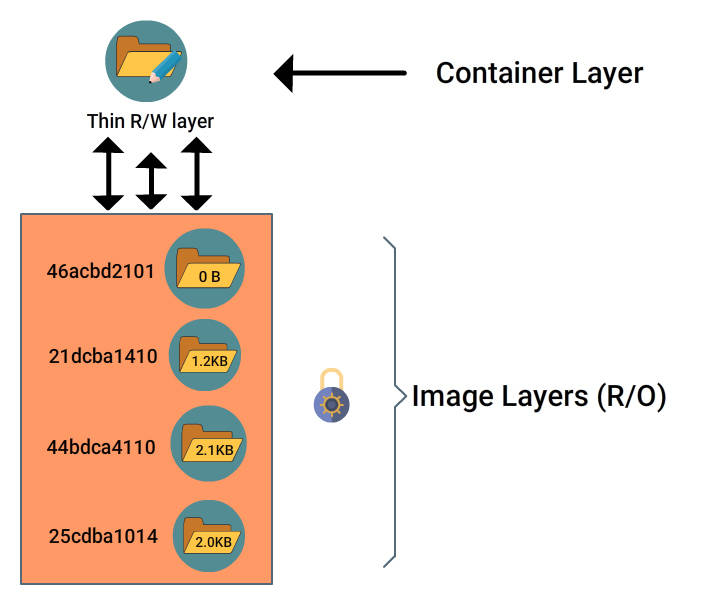
Manager nodes in the swarm rotate the key used to encrypt gossip data every 12 hours.

##### How Images are stored ?

Lets assume that we pull **nginx image** using docker pull command as below.

docker pull nginx

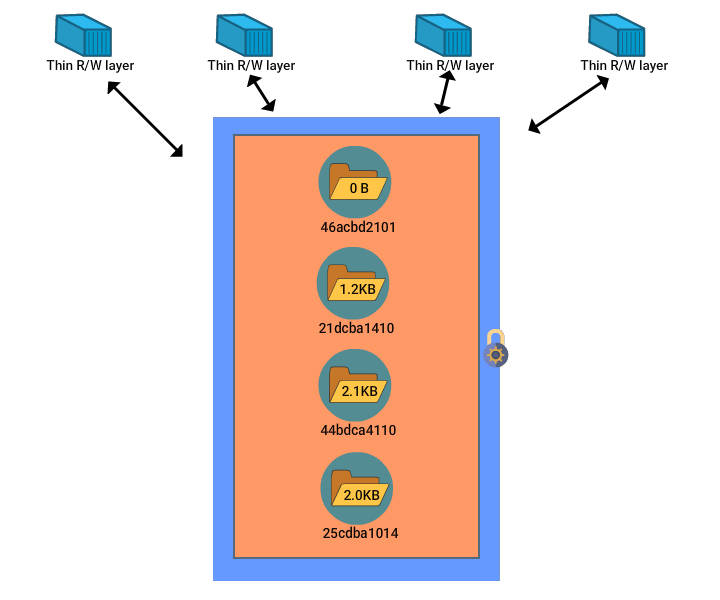
Docker will download this image from the docker hub to host directory which is managed by docker engine running in the host machine.



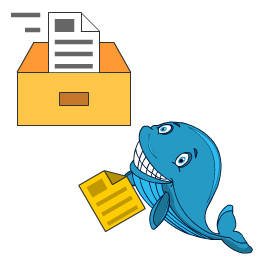
**Docker Images** have a series of in-build layers within. All the subsequent layers except for the last one is **read- only**.

When we create a container from the image built, a **new layer** which is writable is added on top. This is called **container layer**.

Changes done on the running container gets stored in the **thin writable layer**.

The underlying image layer is shared between containers whereas the thin writable layer is separate for every individual containers.

##### Storage Drivers



Storage drivers manage the contents of the image layers as well as the thin writable container layer.

Storage drivers supported for Ubuntu systems are

* aufs
* devicemapper
* overlay2
* overlay
* zfs

##### Storage Driver (Contd...)

Lets run the following command in Katacoda Docker playground to view the Storage driver used.

docker info

Lets consider only the below set of information in the terminal output to understand the current storage driver configured.

Server Version: 1.13.1

Storage Driver: overlay

Backing Filesystem: extfs

##### Modify Storage Driver

Current Storage driver configured can be modified as below.

Step:1

Stop the docker service.

service docker stop

Step 2:

In a Ubuntu machine, add the following to daemon.json file in /etc/docker/ directory to modify the driver to 'devicemapper'

{

"storage-driver": "devicemapper"

}

If you do not find this file, please go ahead and add it.

##### Modify Storage Driver (Contd...)

Step 3:

Once you add it, restart docker service using command

service docker start

Note:

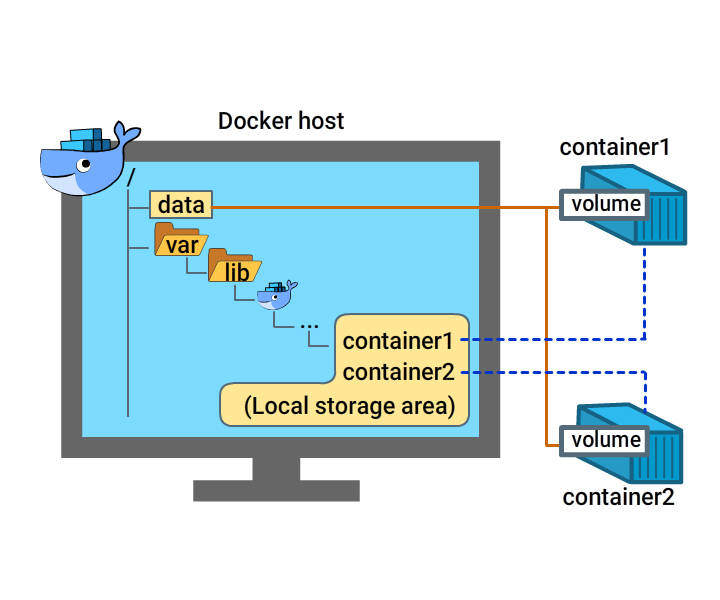
This cannot be tried out in Katacoda playground since this requires a root access.

Step 4:

You can verify using command

docker info

Docker Volumes



Docker Volumes are directories that ***store data outside of the container’s filesystem*** in the host machine.

Data stored on Volumes are ***reusable and are shareable*** even when the ***container is stopped***.

***Data in the volumes*** can be ***reused*** by the same service on redeployment, or ***shared*** with other services.

In Docker Cloud, you can define one or more data volumes for a service.

Containers directly ***read and write*** data on to the volumes.

***Storage drivers*** do not have any control on the data volumes.

##### Docker Volumes - Commands

**Add a Data Volume:**

docker run -d -P --name web -v /webapp training/webapp python app.py

**Note**: You can add **one or more volumes** to a **single container**

**Locate a volume:**

docker inspect web

"Mounts": [

{

"Type": "volume",

"Name": "dbfa40d8080a2e467c91567109ce124683747afcf1511562f77d6dd006733f4f",

"Source": "/var/lib/docker/volumes/dbfa40d8080a2e467c91567109ce124683747afcf1

511562f77d6dd006733f4f/\_data",

"Destination": "/webapp",

"Driver": "local",

"Mode": "",

"RW": true,

"Propagation": ""

}

],

Mounts section of the output trace displays the source and destination as well as Read/write mode if enabled or not.

##### Data Volumes - Commands

**Create a Volume:**

docker volume create --name new\_volume

**Display detailed information on Volumes:**

docker volume inspect new\_volume

**List volumes:**

docker volume ls

**Remove unused volumes**

docker volume prune

This will remove 'new\_volume'.

**Remove one or more volumes**

docker volume create --name new\_volume1

docker volume rm new\_volume1

##### Backup/ Restore/ Migrate Data Volumes

**Command to Back up Volumes:**

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

**Command to Restore Volumes:**

**Create new container**

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

docker run --rm --volumes-from dbstore2 -v $(pwd):/backup ubuntu bash -c "cd /dbdata && tar xvf /backup/backup.tar --strip 1"

##### Create and Mount Data Volume Container

**Data Volume Container** is created to share persistent data between containers.

docker create -v /dbdata --name dbstore training/postgres /bin/true

Here we have mounted a volume '/dbdata' using image 'training/postgres'.

Command to mount '/dbdata' volume to other containers using '--volumes-from' keyword.

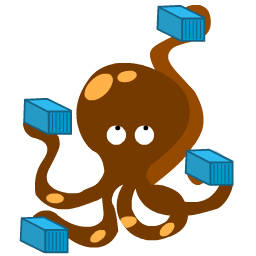
docker run -d --volumes-from dbstore --name db1 training/postgres

docker run -d --volumes-from dbstore --name db2 training/postgres

##### Docker Volumes - Points to remember

* Data volumes can be **reused** and **shared** amongst containers
* Data volume changes are **independent** of the image update
* Even when containers are **deleted**, data volumes **persist**
* Data volumes get **initialized** when we **create a container**

Below mentioned types can be mounted as a Data volume.

* **Host directory**
* **Shared storage volume**
* **Host file**
* Docker Compose
* 
* ***Docker Compose*** is a tool used to ***define and run applications*** containing ***multiple containers***.
* Why ***docker Compose***?
* When an application is built to have ***many services/containers*** , its difficult to manage them separately. Every container will have its own ***dockerfile*** and to ***bring it up*** individual ***run commands*** are to be executed. Whereas with Docker compose, we can bring ***all services up with a single command***.
* Docker compose simplifies the process to run services on a real time large scale applications.

##### Installing Docker Compose

**Installation on Ubuntu**:

Docker compose version : 1.14.0

Run the following command as **super user** (Run sudo -i command to execute as super user)

curl -L https://github.com/docker/compose/releases/download/1.14.0/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose

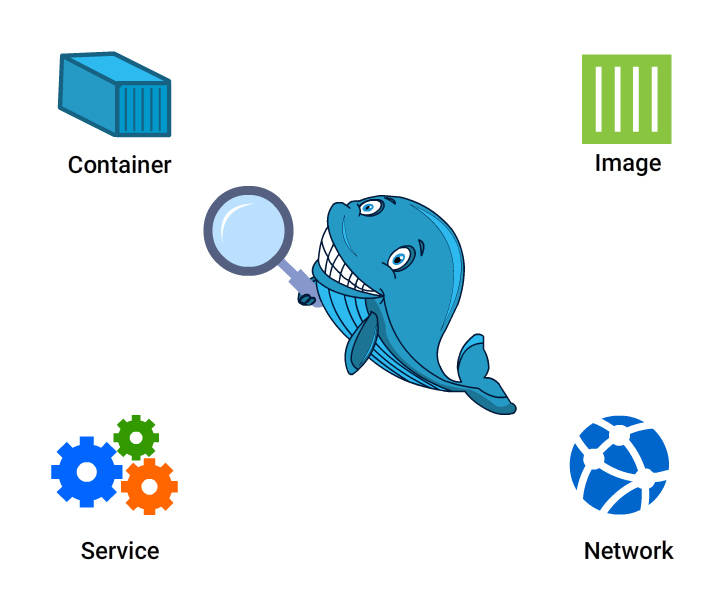
Applying **executable permission** to binary.

sudo chmod +x /usr/local/bin/docker-compose

Verify by running version command.

docker-compose --version

Docker Inspect



This docker command is used to display ***low-level information*** on docker objects such as ***Images, Containers, volume, network, node, service, and task***.

These objects can be identified by either ***object name or ID***.

The default output will be a ***JSON array***.

##### Docker Inspect - Syntax

docker inspect [--help] [-f|--format[=FORMAT]] [-s|--size] [--type] NAME|ID [NAME|ID...]

**--help:** prints usage statement

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

**-s, --size** : Display total file sizes if the type is container

**--type**: container|image|network|node|service|task|volume

Return JSON for specified type, permissible values are "image", "container",

"network", "node", "service", "task", and "volume"

##### Docker Inspect - Example

Let's **pull** an image from **docker hub** and run **Inspect** command.

docker pull tomcat

**Docker inspect** command to display the **information on the Image**.

docker inspect tomcat

When there is a **name conflict** with the image name and the corresponding container name lets add the **type option** and specify the object as **image or container**.

docker inspect --type=image tomcat

##### Display Container Information

We already have a Tomcat image pulled from docker hub. Let's run the same Image as below.

docker run -d --name tomcatContainer tomcat

Let's now run **docker inspect** command on the **new container** created.

docker inspect tomcatContainer

##### Display Size information on a container

docker inspect -s tomcatContainer

Highlighted below the **size section** of the output.

"SizeRw": 37433,

"SizeRootFs": 292398179

##### Formatting Docker Inspect Output

Let's understand more on formatting looking into the below example.

Display **IP address** of Container instance

docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' tomcatContainer

##### More Examples

Display **MAC address** of the container

docker inspect --format='{{range .NetworkSettings.Networks}}{{.MacAddress}}{{end}}' tomcatContainer

Display **log path** of the container

docker inspect --format='{{.LogPath}}' tomcatContainer

Display **Image name** of the container

docker inspect --format='{{.Config.Image}}' tomcatContainer

Display a **subsection in JSON format**

docker inspect --format='{{json .Config}}' tomcatContainer

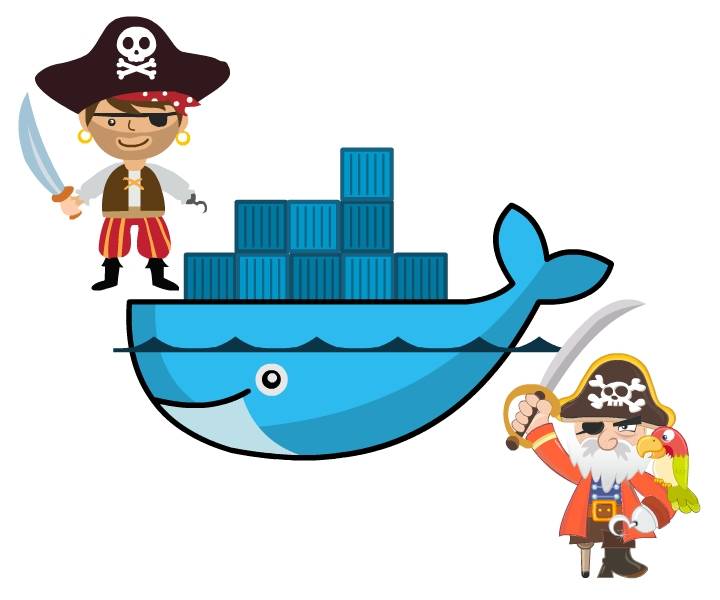
Docker Security



Creating services using docker is ***gaining traction*** when compared to the ***virtual machine set up.***

To use ***services effectively and securely***, We need to be aware of ***potential threats and security issues*** to docker.

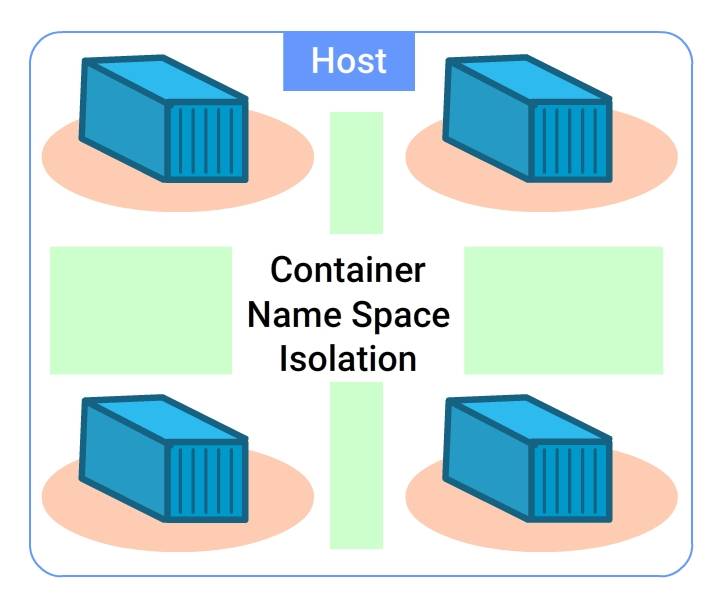
##### Security threats to Docker Container



Let's look into few of the security issues.

* **Risk of privilege escalation via containers.**

For example, if an attacker gets inside a **containerized app**, it becomes easy to gain **root access**to the **host system**.

* A potential attack on **one container** can compromise data or resources used by a **different container**. This can happen even without root access.
* **Simple DoS attacks** where one container gets all available **system resources** which stop other containers from functioning properly.
* **Pulling insecure or invalidated images** from the public repository.
* Container Isolation - Kernal Namespaces
* 
* ***LXC containers*** are similar to ***Docker Containers*** and hence the ***security features*** remain similar.
* Docker creates a ***namespace and control groups*** for the container when it is started.
* ***Namespace*** provide a ***isolation*** between containers running on the same host machine.
* Every container owns an ***individual network stack***. Containers interact with rest others as if they interact with ***external hosts***. Every container is like a ***physical machine*** connected to rest through a ***common Ethernet switch***.
* **Control groups** are responsible for **accounting and limiting of resources**.
* They ensure that all containers receive **fair share of available resources** (CPU, memory and disk space)
* **Prevent Container based Dos attacks** (Denial of Service). i.e. it ensures that one container cannot exhaust all the resources available which might bring the whole system down.

##### Docker Daemon Attack

**Only trusted users** should be given access to control the docker daemon since this user will have **root privilege to the host** system to run docker daemon.

Also, Docker gives the **flexibility to share files/ directories** between the **host system and the container**.

By this way, a **running container** can directly **access the host directory** and modify the files.

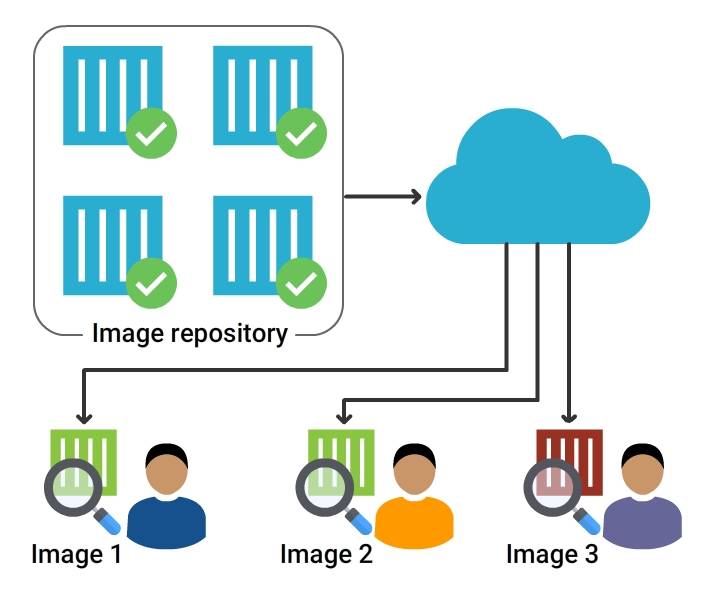
##### Docker Daemon Attack (Contd...)

**Parameter checking** is very important when we use the web server to **provision containers**since **malicious users** can easily create **arbitrary containers**.

**Rest API endpoint** was modified to use **UNIX socket instead of TCP socket** since the TCP sockets are prone to **cross site forgery attacks**.

When we expose the **REST API over HTTP**, we should ensure that we have **enough security**using client stunnel, SSL certificates, HTTPS and certificates.

Securing Images



***Poisoned images*** cause a ***serious problem*** since the attacker can take control over the complete host system as well as the data.

With Docker 1.3.2, images are extracted in ***chrooted subprocess***. This is a major step towards ***privilege separation***.

With Docker 1.10.0, ***cryptographic checksums*** of their Image contents are used to store and access them. This prevents the attacker from ***colliding with the existing images***.

##### Linux Kernal Capabilities

**Bunch of processes** listed below needs **root access** on Linux system to run. But the **infrastructure around the container** takes care of almost all of these tasks.

* SSH
* Cron
* Log Management
* Hardware Management
* Network Management

Containers do **not need root privileges** in most of the cases. Hence containers will have a **limited** root privilege rather than the complete root access.

##### Linux Kernal Capabilities

Docker follows a **whitelist approach** by allowing only the capabilities listed.

Docker also supports **addition and removal of capabilities**. With the reduction in capabilities, docker becomes more secure and with the addition of capabilities, it becomes less secure.

The complete list of capabilities is​ listed here.

http://man7.org/linux/man-pages/man7/capabilities.7.html

##### Docker Features - User Namespace

With Docker 1.10, **User Namespaces** are supported by the Docker daemon.

User Namespaces allows users to work on **daemon commands** on containers with **root access**but **without root privilege on the host**.

The container root user is **mapped** to another non uid-0 outside the container. This will **reduce the risk of Container breakout**.

##### Why User Namespace?

Let consider the below **example**.

User 100 in container 1 is mapped to User 501 outside the container.

User 150 in container 2 is mapped to User 502 outside the container.

This is something similar to **network port mapping**. This mapping is done to enable the administrator to **set privilege** to the users. Admin can set uid 0 (root) to a user inside container 1 without giving root access to the host system

##### Secure Docker

To summarize,

* Add **-u flag** to Docker run command while **starting a container** to run as a **normal user**instead of **root user**
* Remove **SUID flags** from container images to make **privilege escalation** attack **harder**
* Add **Docker Control groups** and configure them to set limits on **resource usage** for a container
* Add **user namespace** in Docker to create an isolation between the containers as well as the container and the host

##### Secure Docker (Contd...)

* Use only **trusted images** or images from **official source** from Docker repository
* Use software for **static analysis of vulnerabilities** in application containers e.g.**Clair**
* Apart from the capabilities, leverage systems like **TOMOYO, AppArmor, SELinux, GRSEC,**etc. with docker

##### Other Security features

Apart from the capabilities, systems like **TOMOYO, AppArmor, SELinux, GRSEC,** etc. can be leveraged with docker.

Running a kernel with **GRSEC and PAX**, will add more **security** at both **compile and run time**.

**Security model templates** that work with docker can be used.

For e.g. **AppArmor and Red Hat** offers **SELinux** policies for Docker. These provide **additional security** on top of capabilities.

Users can themselves define policies with their access control mechanism.

**DOCKER – BEST PRACTICES**

Avoid **storing data** in containers.

Use **Volumes** to store data if needed, since a container can be **stopped** or application running can be **re-version-ed**.

**Avoid creating large images which is difficult to manage across multiple containers.**

Always create **multi-layered Image**.

It is easier to **manage and distribute** across multiple containers.

Always use **Dockerfile** to create an image.

This process is **reproducible** and changes to this file can be **tracked** through any version control.

**Tag images** with correct version.

**Avoid** generic tags like **'Latest'** to the image.

Always run **one process per containers**.

This will be easier to **manage, trouble shoot issues**and update processes individually.

Use **Environment variables** to store credentials.

Do not **store directly** in the image.

Run processes with **non-root user id**.

Communicate between containers using **environmental variables**.

Do not use **IP address**, since it **changes** for every container restart.

Keep **Container interactions minimal**.

Do not accept any connection on exposed ports through any network interface.

Use tools to **monitor and identify vulnerabilities** in images.

Wherever feasible, Run containers in Read -only mode.

Do not install **unnecessary packages**.

This improves **performance and reduces complexity**.