NobleProg

The World's Local Training Provider

About me

Lukasz Zajączkowski
zreigz@gmail.com

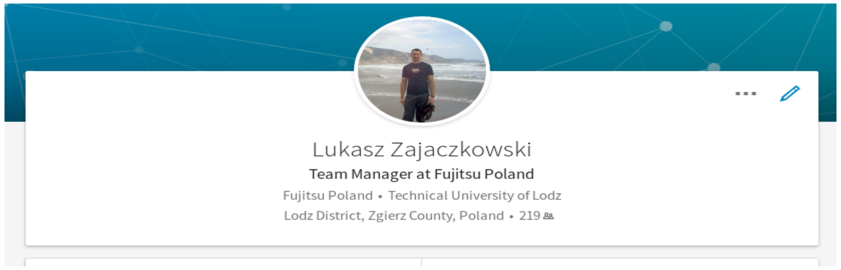
Lukasz
Zajaczkowski
zreigz

Add a bio

GDC Fujitsu Poland
Lodz
lukasz.zajaczkowski@ts.fujitsu...

Organizations

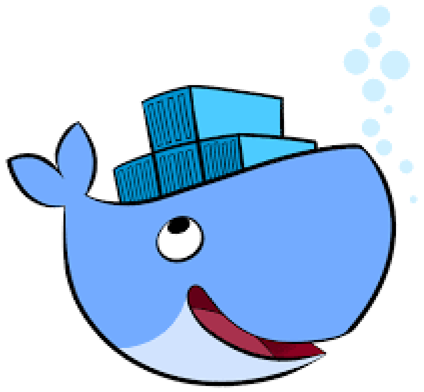
https://github.com/zreigz


Lukasz Zajaczkowski
Team Manager at Fujitsu Poland
Fujitsu Poland • Technical University of Lodz
Lodz District, Zgierz County, Poland • 219

https://www.linkedin.com/in/zreigz/

NobleProg

Introduction to docker



- docker concepts, tools, ecosystem and what it’s used for.

- what are containers and their history and stating their applications in real life and what are the famous implementations of containers.

- quick view of the docker vocab,architecture, techniques behind it and a comparison between container and Virtual Machines.

NobleProg

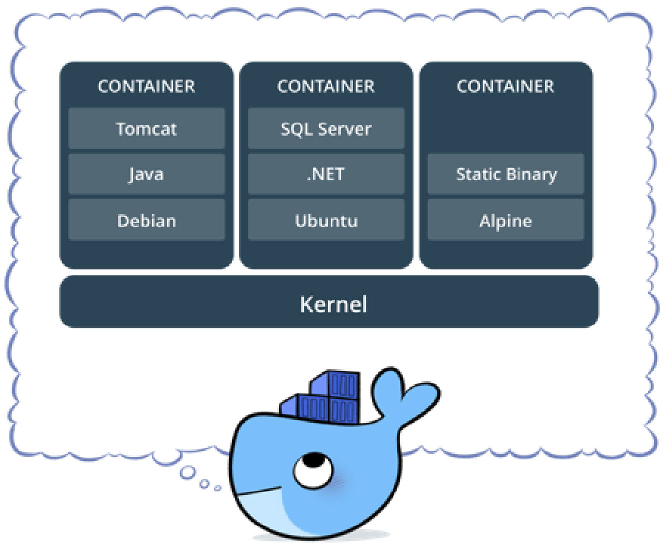
What are Containers ?

Containers is a light weight virtualization which is also known as Operating system level virtualization. As a definition its a middle-ware and echo system that allows sharing the Operating system between multiple instances that are called containers.

Each container is separated from other containers and could have a different Linux flavor than other container and the base operating system.

What are Containers ?

Instead of shipping around a full operating system and your software (and maybe the software that your software depends on), you simply pack your code and its dependencies into a container that can then run anywhere



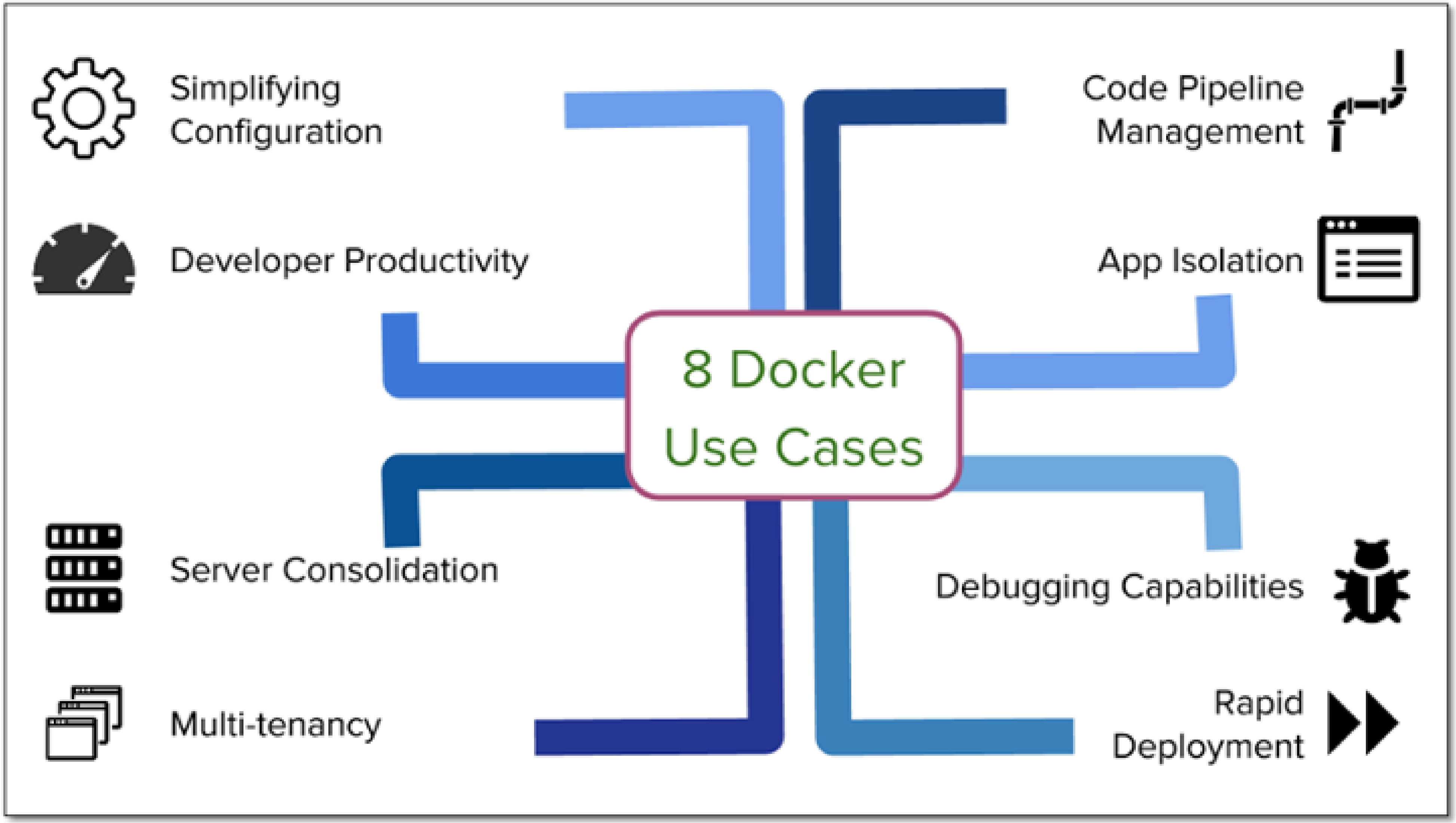
History Of Containers

- 1979 chroot
- 2000 jails
- 2004 Solaris Containers
- 2005 Open VZ
- 2006 Google process containers
- 2007 HP-UX Containers
- 2008 LXC (Linux Containers)
- 2013 Docker
- 2013 lmctfy (Let Me contain that for you) was release by Google to compete with Docker
- 2014 Google Announces that it has used containers as a concept since 2006 and released its open source project kubernetes that allows container clustering.
- 2015 OPEN CONTAINER INITIATIVE annoced that the Docker lib containers is standardized as the base for a Linux kernel feature.

Containers in Reallife

- Highly performant, hyper scale deployments of hosts
- Micro-Services architecture
- Building a multi-user Platform-as-a-Service (PAAS) infrastructure
- Sharing Environments between Developers
- Testing Environments
- Research Validation

Containers in Reallife



What is docker

Docker is “a platform for developers and sysadmins to develop, ship, and run applications”, based on containers

Docker is open-source, mainly created in Go and originally on top of libvirt and LXC - later replaced by a unifying library, libcontainer, written in Go as well

Docker simplifies and standardizes the creation and management of containers, also providing a simple and elegant Remote API to perform queries and actions

Build, Ship and Deploy

Build your application the way you always have, with the tools you've always used.
Deploy

Design the entire cycle of application development, testing and distribution, and manage it consistently and securely within your organization across teams.

Deploy Docker containers across private and a variety of public clouds.
Run

Docker Vocab

Docker Image and Container
A Docker image is template or a class that has all the necessary file to create a container. The difference between an image and a container is like the difference between the class and an object. A class is a manuscript of what should be there without any physical existence and this is just what an image is an image contains what should be there without any actual access to any resources (e.g. CPU time, memory and storage). On the other hand a container is the object it has a physical existence on the host with an IP, exposed ports and real Processes.

Docker Engine
The docker command run in daemon mode. This turns a Linux system into a Docker server that can have containers deployed, launched, and torn down via a remote client.

Docker Client
The docker command used to control most of the Docker workflow and talk to remote Docker servers.

Docker Hub (Registry)
The Storage component of docker that allows you to save, use and search for images.

Docker is not

- Docker is not an Automation or configuration management tool (Puppet, Chef, and JUJU)
- Docker is not a Hardware Virtualization Solution (VMware, KVM, and XEN)
- Docker is not a Cloud Platform (Open Stack, and Cloud Stack)
- Docker is not a Deployment Framework (Capistrano, Fabric, etc.)
- Docker is not a Workload Management Tool (Mesos, Fleet, etc.)
- Docker is not Development Environment (Vagrant, etc.)

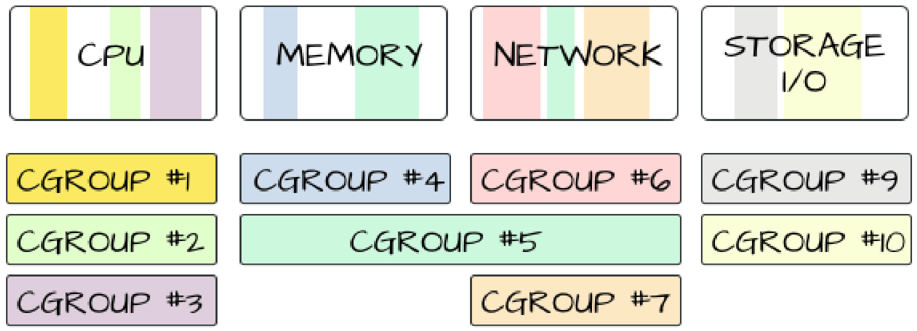
Techniques Behind Docker

The Techniques behind docker includes:

- Control Groups
- Kernel Name Spaces
- Union File system
- Capabilities
- Copy on Write

Techniques Behind Docker

Control Groups



It's a method to put processes into groups by allowing the Following:

- Resource limitation: groups can be set to not exceed a configured memory limit, which also includes the file system cache.
- Prioritization: some groups may get a larger share of CPU utilization or disk I/O throughput.
- Accounting: measures how much resources certain systems use, which may be used, for example, for billing purposes.
- Control: freezing the groups of processes, their checkpointing and restarting

Techniques Behind Docker

Kernel Namespaces

A kernel namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource.

Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes.

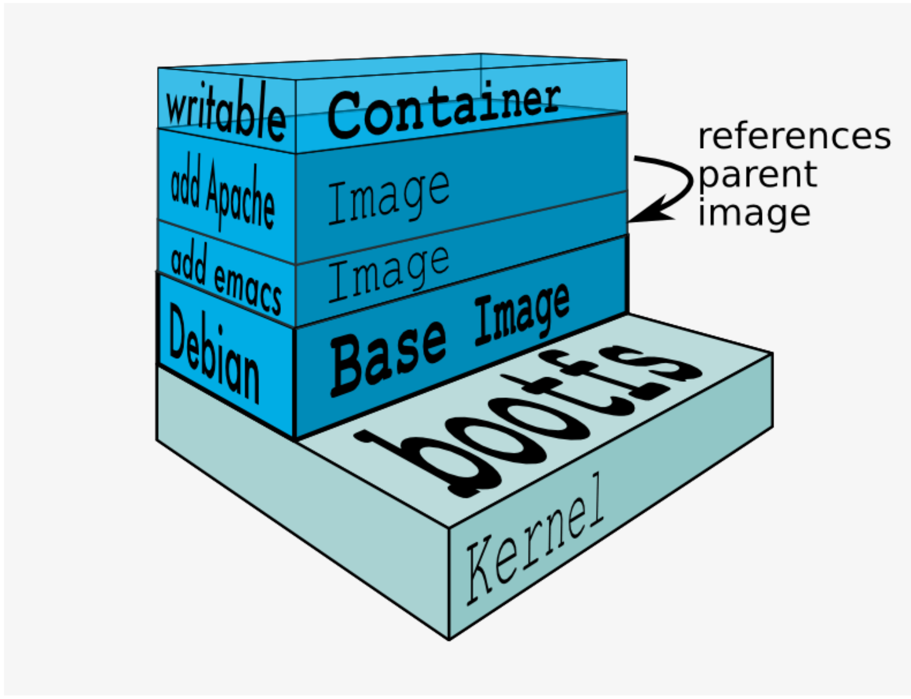
There are 6 Name Spaces:

- **PID** namespace provides isolation for the allocation of process identifiers (PIDs), lists of processes and their details. While the new namespace is isolated from other siblings, processes in its "parent" namespace still see all processes in child namespaces—albeit with different PID numbers.
- **Network** namespace isolates the network interface controllers (physical or virtual), iptables firewall rules, routing tables etc. Network namespaces can be connected with each other using the "veth" virtual Ethernet device.
- "**UTS**" namespace allows changing the hostname.
- **Mount** namespace allows creating a different file system layout, or making certain mount points read-only.
- **IPC** namespace isolates the System V inter-process communication between namespaces.
- **User namespace** isolates the user IDs between namespaces.

Techniques Behind Docker

Union File System

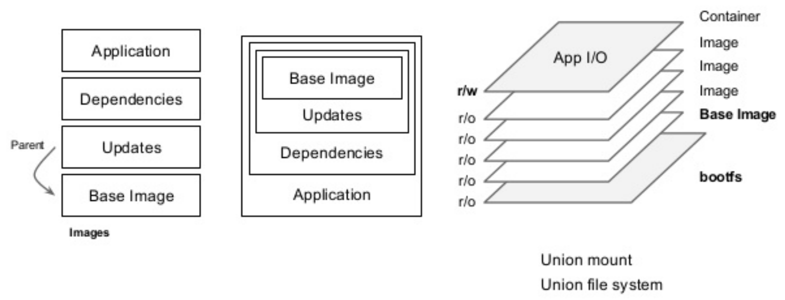
Union file system represents file system by grouping directories and files in branches. A Docker image is made up of filesystems layered over each other (i.e. Branches that is stacked on top of each other) and grouped togeather. At the base is a boot filesystem, bootfs, which resembles the typical Linux/Unix boot filesystem.



Techniques Behind Docker

Union File System

Image Anatomy



- Not changed files are shared for each containers
- Storage and memory are saved
- Faster deployment of containers

Techniques Behind Docker

Capabilities

Each Linux process has four sets of bitmaps. By Default each bitmap is 32 bit for 32 different capability.

Effective (E):

The effective capability set indicates what capabilities are effective The Process can use now

Permitted (P):

The process can have capabilities set in the permitted set that are not in the effective set. This indicates that the process has temporarily disabled this capability. A process is allowed to set a bit in its effective set only if it is available in the permitted set. The distinction between effective and permitted makes it possible for a process to disable, enable and drop privileges

Inheritable (I):

Indicates what capabilities of the current process should be inherited by the program executed by the current process

Bset:

Determine forbidden capabilities

Techniques Behind Docker

Copy On Write

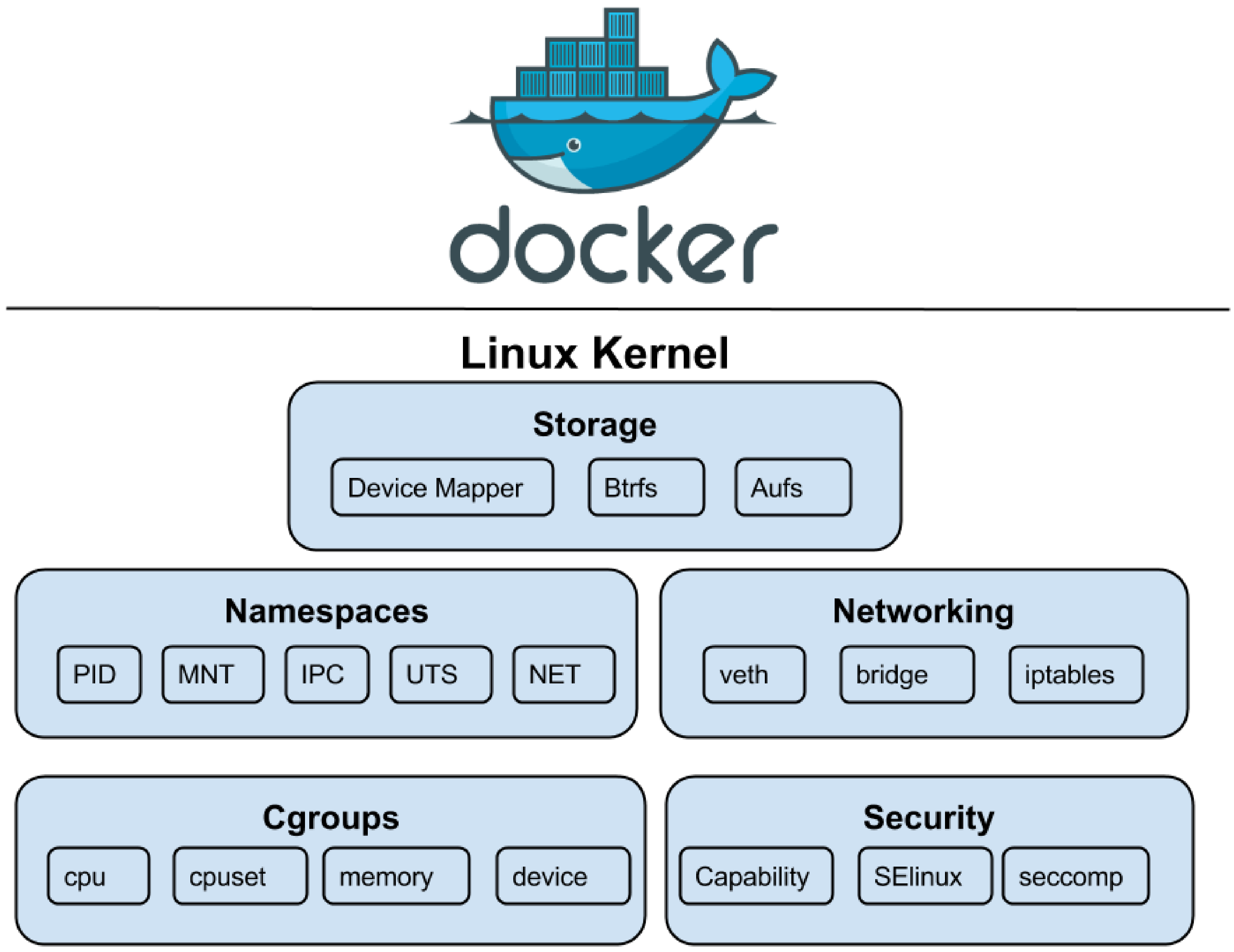
The Middleware is responsible for creating and managing the container. When a container is started it appears to have its own Linux files system that you're using. The fact is you don't; at the start of the container it links to files in the base kernel that all containers shared, and the way it handle the changes is via the copy on write model, where each change to the file system is copied to an in memory version of the base file with the changes.

This results two main effects:

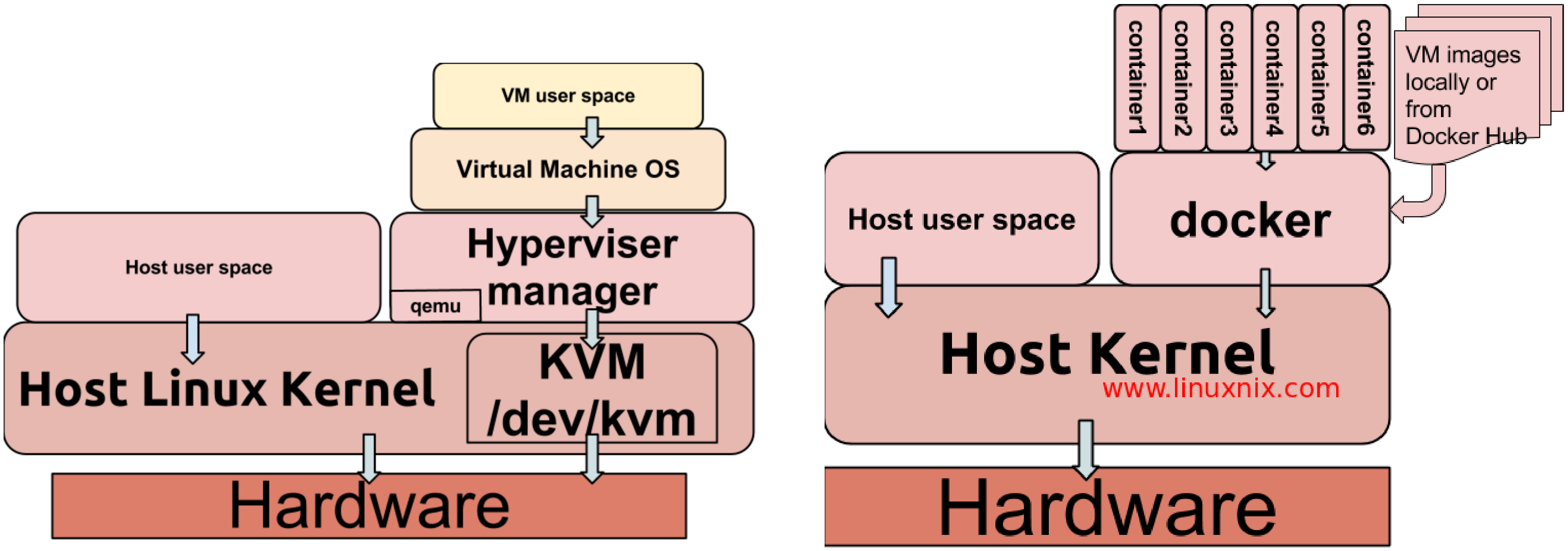
- The container will result a small footprint.
- changes in memory are faster than writing to storage.

This works in conjunction with UNION File system. Where, each image layer is read-only; this image never changes. When a container is created, Docker builds from the stack of images and then adds the read-write layer on top. That layer, combined with the knowledge of the image layers below it and some configuration data, form the container.

Techniques Behind Docker

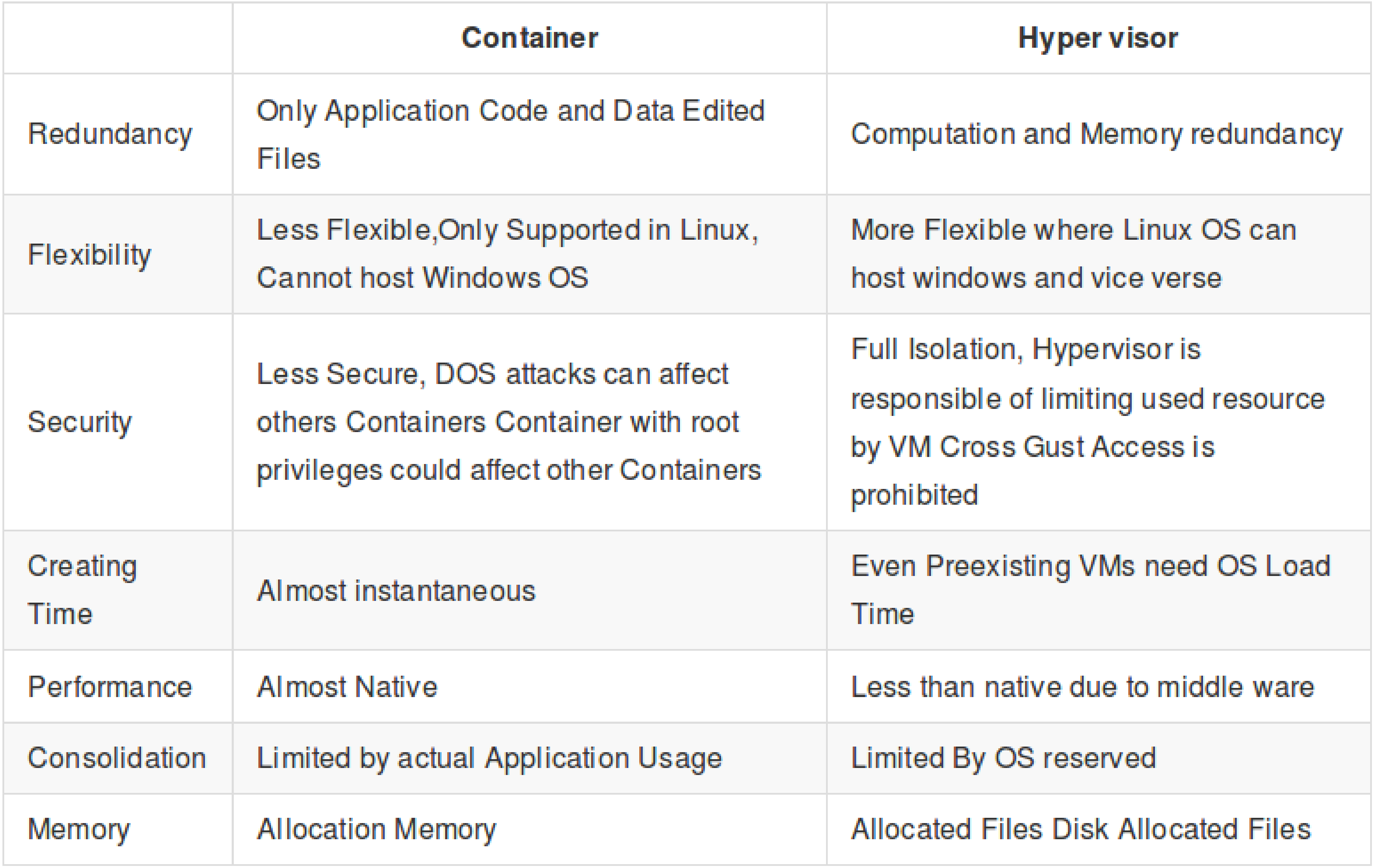


Docker vs Virtual Machines



For a single application(Like Apache or DNS application etc) to run we are creating whole virtual machine which require up many virtual resources like CPU, RAM and hard disk. This is not an efficient way of utilizing resources and to just run one application we are creating whole Operating system.

Docker vs Virtual Machines



	Container	Hyper visor
Redundancy	Only Application Code and Data Edited Files	Computation and Memory redundancy
Flexibility	Less Flexible,Only Supported in Linux, Cannot host Windows OS	More Flexible where Linux OS can host windows and vice verse
Security	Less Secure, DOS attacks can affect others Containers Container with root privileges could affect other Containers	Full Isolation, Hypervisor is responsible of limiting used resource by VM Cross Gust Access is prohibited
Creating Time	Almost instantaneous	Even Preexisting VMs need OS Load Time
Performance	Almost Native	Less than native due to middle ware
Consolidation	Limited by actual Application Usage	Limited By OS reserved
Memory	Allocation Memory	Allocated Files Disk Allocated Files

Docker Architecture

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.

Docker Architecture

The Docker daemon

The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker client

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

Docker registries

A Docker registry stores Docker images. Docker Hub and Docker Cloud are public registries that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry. If you use Docker Datacenter (DDC), it includes Docker Trusted Registry (DTR).

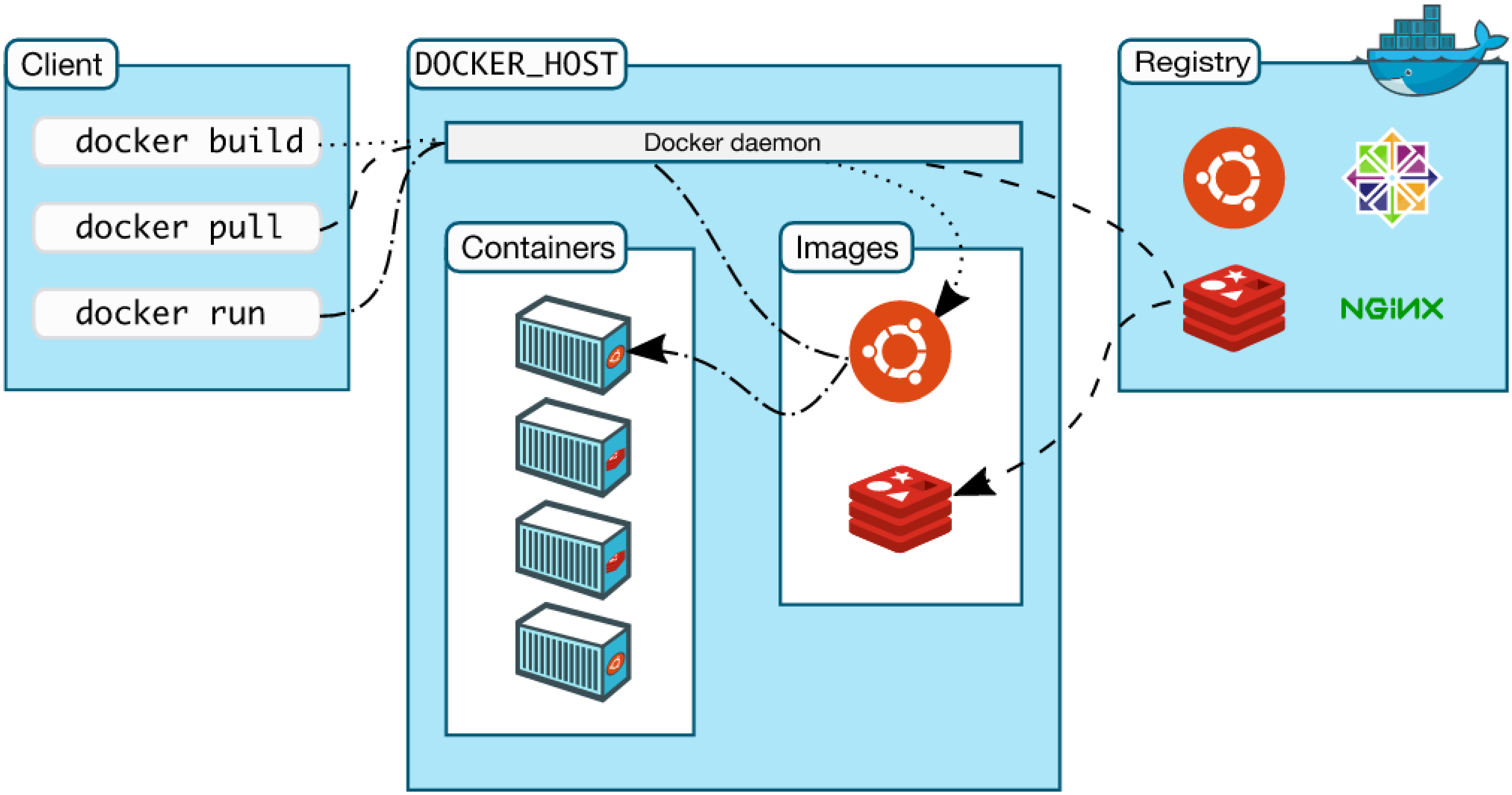
When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

Docker store allows you to buy and sell Docker images or distribute them for free. For instance, you can buy a Docker image containing an application or service from a software vendor and use the image to deploy the application into your testing, staging, and production environments. You can upgrade the application by pulling the new version of the image and redeploying the containers

Docker objects

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

Docker Architecture



The EcoSystem

The docker ecosystem is rich and allows multiple types of orchestration and clustering tools.

The tools that are built for the docker ecosystem and included in the book are :

- Kubernetes
- Mesos Marathon
- CoreOS
- Flocker

Also the docker integrate with tools like:

- Puppet
- Chef
- Ansible

Installation

Open github project to get information how to install Docker

https://github.com/zreigz/docker-workshop/tree/master/installation

Container Control Commands

The docker have some commands that allows the some control over a container.

- Create
- Start
- Stop
- Restart
- Execute
- Copy
- Attache
- Inspect
- Delete
- Logs

Docker Create

The docker create command creates container that allows setting configuration you can set in run command the only difference that the created container is never started and in order to start a container run the start command.

```
$ sudo docker create [OPTIONS] IMAGE [COMMAND] [ARG...]
```

Example:

```
$ docker create -t -i fedora bash
6d8af538ec541dd581ebc2a24153a28329acb5268abe5ef868c1f1a261221752
$ docker start -a -i 6d8af538ec5

bash-4.2#
```

Docker Start, Stop and Restart

These three commands are used to change the state of a container from running to stopped or restart it.

Stop: When a user issues this command, the Docker engine sends SIGTERM (-15) to the main process, which is running inside the container.

The SIGTERM signal requests the process to terminate itself gracefully. Most of the processes would handle this signal and facilitate a graceful exit. However, if this process fails to do so, then the Docker engine will wait for a grace period. Even after the grace period, if the process has not been terminated, then the Docker engine will forcefully terminate the process. The forceful termination is achieved by sending SIGKILL (-9).

The SIGKILL signal cannot be caught or ignored, and so it will result in an abrupt termination of the process without a proper clean-up.

```
$ docker stop | start| restart [OPTIONS] CONTAINER [CONTAINER...]
```

```
[Options]
-t, --time=10      Seconds to wait for stop before killing it

[Container]
Control the container either by Name or ID
```

Docker Execute

Run a command in a running container

```
docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
```

```
-d, --detach=false         Detached mode: run command in the background
-i, --interactive=false    Keep STDIN open even if not attached
-t, --tty=false            Allocate a pseudo-TTY
-u, --user=                Username or UID (format: <name|uid>[:<group|gid>])
```

```
$ docker exec -d ubuntu_bash touch /tmp/execWorks
```

This will create a new file /tmp/execWorks inside the running container ubuntu_bash, in the background.

Docker Copy

Copy data [From|To] container, It copies data from and to running containers.

```
$ docker cp [OPTIONS] CONTAINER:PATH LOCALPATH|-
```

```
$ docker cp [OPTIONS] LOCALPATH|- CONTAINER:PATH
```

Copy Data from Container to host

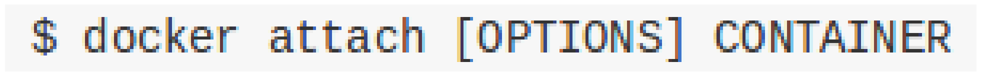
```
$ sudo docker cp testcopy:/root/file.txt .
```

Copy Data from host to container

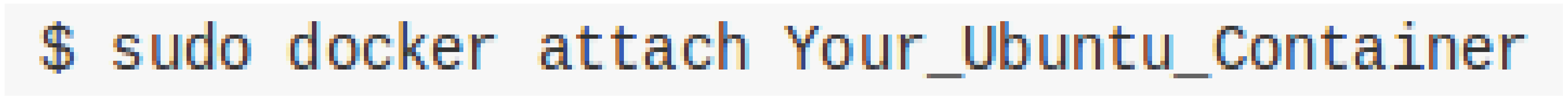
```
$ sudo docker cp host.txt testcopy:/root/host.txt
```

Docker Attach

This commands brings a container to the foreground. The container must be running to be attached.



```
$ docker attach [OPTIONS] CONTAINER
```



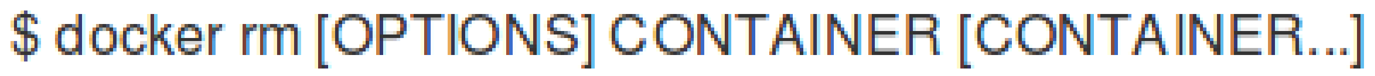
```
$ sudo docker attach Your_Ubuntu_Container
```

Docker Inspect

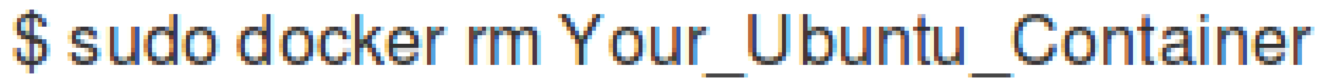
This command is used to check for the current configuration of a container from network to drivers and name. It prints the information in the form of JSON File

Docker delete

This command is used to delete any container either by name or ID. Put Make sure that the container is stopped before you remove it.



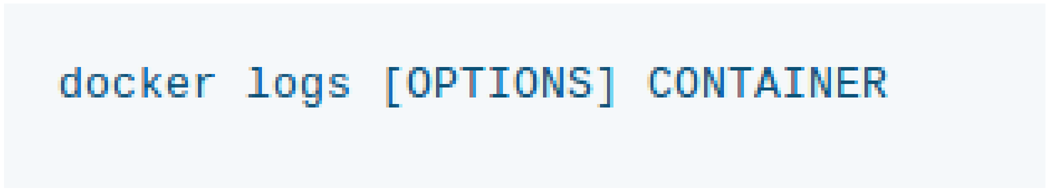
```
$ docker rm [OPTIONS] CONTAINER [CONTAINER...]
```



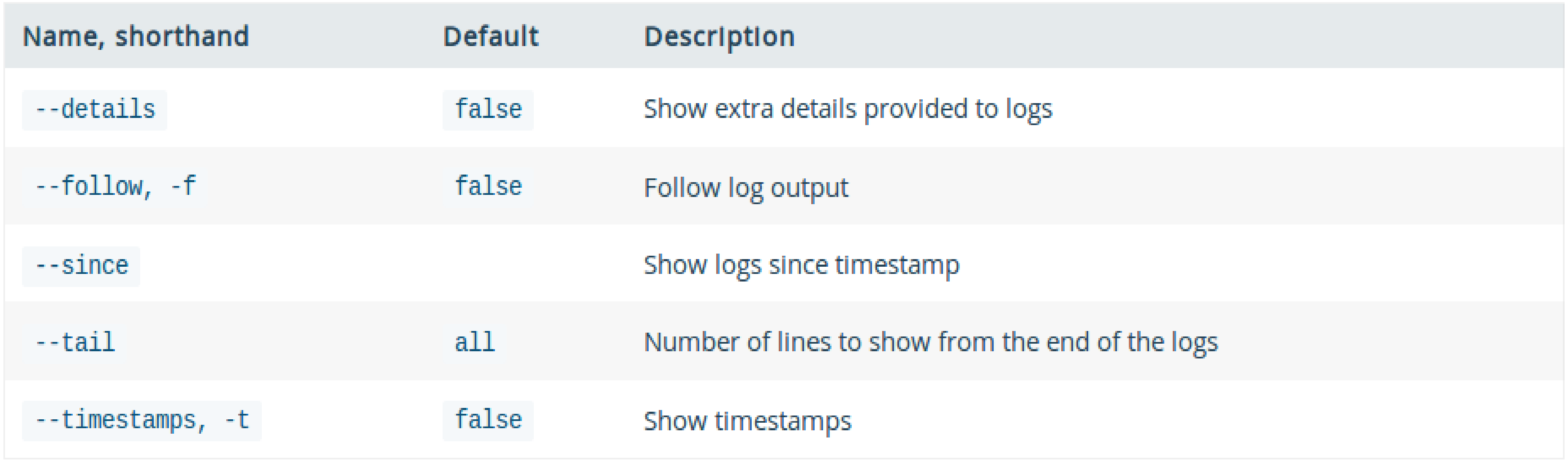
```
$ sudo docker rm Your_Ubuntu_Container
```

Docker Logs

Fetch the logs of a container



```
docker logs [OPTIONS] CONTAINER
```



Name, shorthand	Default	Description
`--details`	`false`	Show extra details provided to logs
`--follow, -f`	`false`	Follow log output
`--since`		Show logs since timestamp
`--tail`	`all`	Number of lines to show from the end of the logs
`--timestamps, -t`	`false`	Show timestamps

Examples

Go to github:

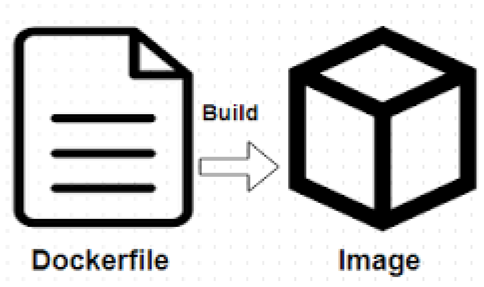
https://github.com/zreigz/docker-workshop/tree/master/commands

Clone this project:

```
$ git clone https://github.com/zreigz/docker-workshop.git
$ cd docker-workshop/commands
```

Using Dockerfiles to Automate Building of Images

Each Dockerfile is a script, composed of various commands (instructions) and arguments listed successively to automatically perform actions on a base image in order to create (or form) a new one. They are used for organizing things and greatly help with deployments by simplifying the process start-to-finish.



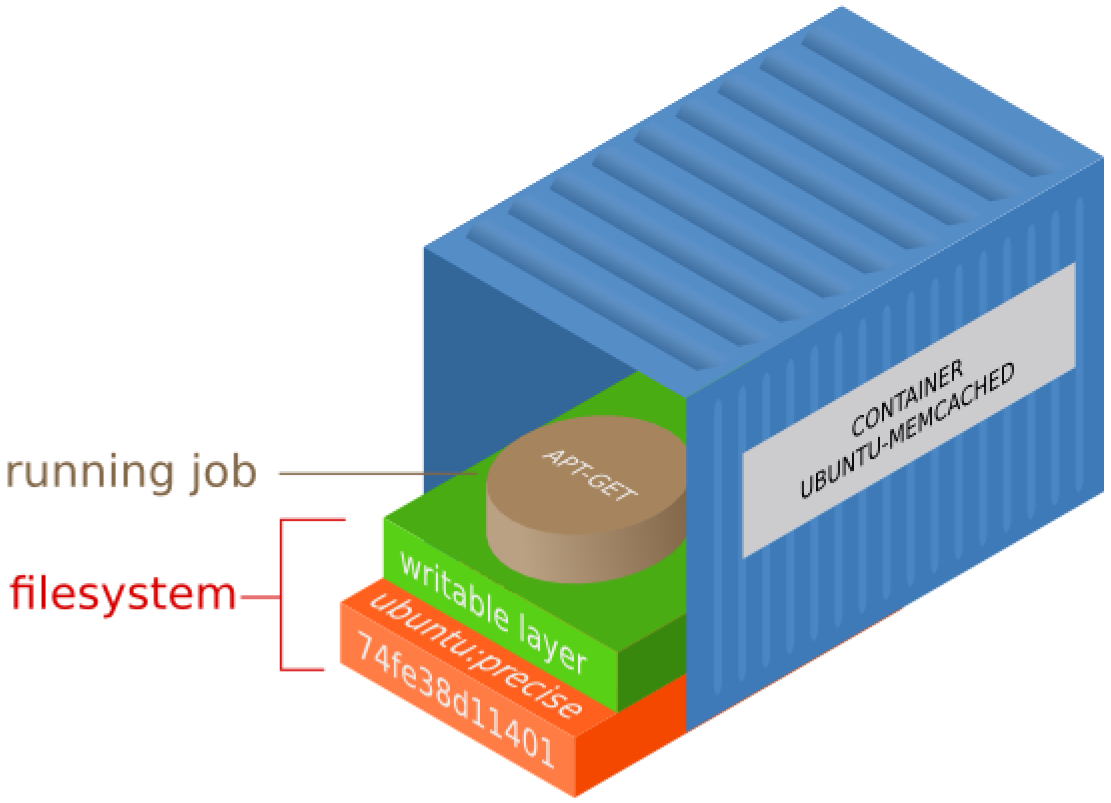
Using Dockerfiles to Automate Building of Images

Some of the benefits Dockerfiles provide are:

- Easy versioning: Dockerfiles can be committed and maintained via version control to track changes and revert any mistakes.
- Predicbility: Building images from a Dockerfile helps remove human error from the image creation process.
- Accountability: If you plan on sharing your images, it is often a good idea to provide the Dockerfile that created the image as a way for other users to audit the process.

Using Dockerfiles to Automate Building of Images

- ADD
- CMD
- ENTRYPOINT
- ENV
- EXPOSE
- FROM
- MAINTAINER
- RUN
- USER
- VOLUME
- WORKDIR



FROM

- Sets the base image for subsequent instructions
- Usage: FROM <image>
- Example: FROM ubuntu
- Needs to be the first instruction of every Dockerfile

- TIP: find images with the command: docker search

RUN

- Executes any commands on the current image and commit the results
- Usage: RUN <command>
- Example: RUN apt-get install –y memcached

```
FROM ubuntu
RUN apt-get install -y memcached
```

is equivalent to:

```
$ docker run ubuntu apt-get install -y memcached
$ docker commit ID
```

ENTRYPOINT & CMD

Both CMD and ENTRYPOINT instructions define what command gets executed when running a container. There are few rules that describe their co-operation.

1. Dockerfile should specify at least one of CMD or ENTRYPOINT commands.
2. ENTRYPOINT should be defined when using the container as an executable.
3. CMD should be used as a way of defining default arguments for an ENTRYPOINT command or for executing an ad-hoc command in a container.
4. CMD will be overridden when running the container with alternative arguments.

EXPOSE

- Sets ports to be exposed to other containers when running the image
- Example: EXPOSE 80

ENV

The ENV command is used to set the environment variables (one or more). These variables consist of “key = value” pairs which can be accessed within the container by scripts and applications alike. This functionality of docker offers an enormous amount of flexibility for running programs.

Example:

```
# Usage: ENV key value
ENV SERVER_WORKS 4
```

WORKDIR

The WORKDIR directive is used to set where the command defined with CMD is to be executed.

Example:

```
# Usage: WORKDIR /path
WORKDIR ~/
```

ADD

The ADD command gets two arguments: a source and a destination. It basically copies the files from the source on the host into the container's own filesystem at the set destination. If, however, the source is a URL (e.g. http://github.com/user/file/), then the contents of the URL are downloaded and placed at the destination.

Example:

```
# Usage: ADD [source directory or URL] [destination directory]

ADD /my_app_folder /my_app_folder
```

COPY

Although ADD and COPY are functionally similar, generally speaking, COPY is preferred. That’s because it’s more transparent than ADD. COPY only supports the basic copying of local files into the container, while ADD has some features (like local-only tar extraction and remote URL support) that are not immediately obvious. Consequently, the best use for ADD is local tar file auto-extraction into the image, as in ADD rootfs.tar.xz /.

Example:

```
# Usage: COPY [source directory] [destination directory]

COPY /my_app_folder /my_app_folder
```

HEALTHCHECK

The HEALTHCHECK instruction tells Docker how to test a container to check that it is still working. This can detect cases such as a web server that is stuck in an infinite loop and unable to handle new connections, even though the server process is still running.

Example:

```
HEALTHCHECK --interval=5m --timeout=3s \
  CMD curl -f http://localhost/ || exit 1
```

Examples

https://github.com/zreigz/docker-workshop/tree/master/dockerfile

Best Practices for working with Dockerfiles

Containers should be ephemeral

The container produced by the image your Dockerfile defines should be as ephemeral as possible. By “ephemeral,” we mean that it can be stopped and destroyed and a new one built and put in place with an absolute minimum of set-up and configuration. Should fulfill 12 Factor app methodology to get a feel for the motivations of running containers in such a stateless fashion.

Best Practices for working with Dockerfiles

Avoid installing unnecessary packages

In order to reduce complexity, dependencies, file sizes, and build times, you should avoid installing extra or unnecessary packages just because they might be “nice to have.” For example, you don’t need to include a text editor in a database image.

Best Practices for working with Dockerfiles

Minimize the number of layers

You need to find the balance between readability (and thus long-term maintainability) of the Dockerfile and minimizing the number of layers it uses. Be strategic and cautious about the number of layers you use.

Sort multi-line arguments

Whenever possible, ease later changes by sorting multi-line arguments alphanumerically. This will help you avoid duplication of packages and make the list much easier to update.

Here’s an example from the buildpack-deps image:

```
RUN apt-get update && apt-get install -y \
  bzr \
  cvs \
  git \
  mercurial
```

Best Practices for working with Dockerfiles

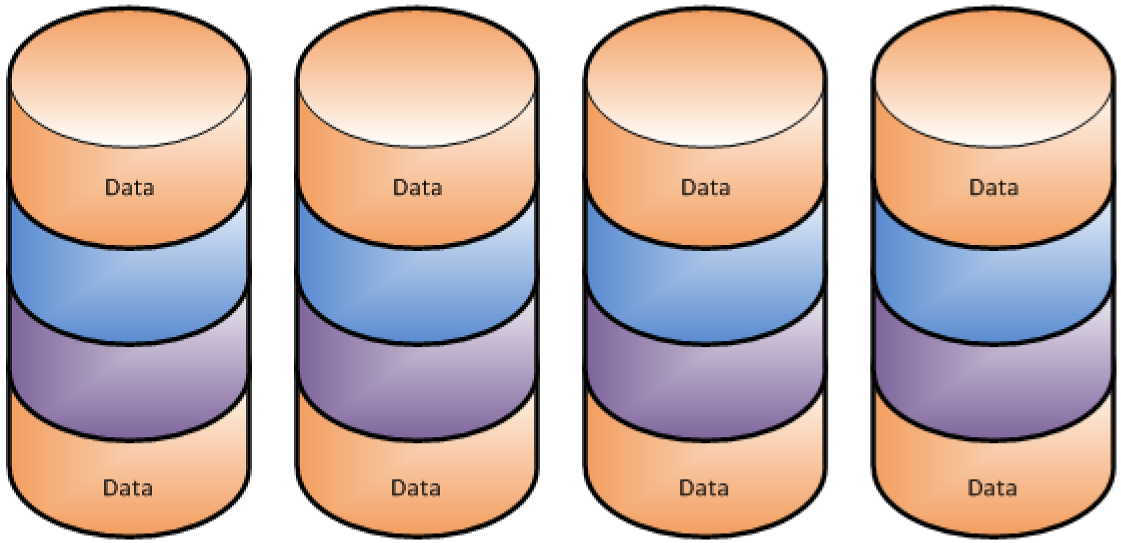
Use multi-stage builds

- One of the most challenging things about building images is keeping the image size down. Each instruction in the Dockerfile adds a layer to the image.
- With multi-stage builds, you use multiple FROM statements in your Dockerfile. Each FROM instruction can use a different base, and each of them begins a new stage of the build. You can selectively copy artifacts from one stage to another, leaving behind everything you don’t want in the final image.

```
FROM golang:1.7.3 as builder
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go    .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .


FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /go/src/github.com/alexellis/href-counter/app .
CMD ["./app"]
```

Data Volumes



Data Volumes

There are 2 ways in which you can manage data in Docker:

- Data Volumes
- Host Data Volumes

Data Volumes

- A data volume is a specially designed directory in the container.
- It is initialized when the container is created. By default, it is not deleted when the container is stopped. It is not even garbage collected when there is no container referencing the volume.
- The data volumes are independently updated. Data volumes can be shared across containers too. They could be mounted in read-only mode too.

Data Volumes

Mounting a Data volume

A “data volume” is a marked directory inside of a container that exists to hold persistent or commonly shared data. Assigning these volumes is done when creating a new container.

You can add a data volume to a container using the -v flag in conjunction with the create or run command. You can use the -v multiple times to mount multiple data volumes.

```
$ docker run -d -it -v /data --name container1 busybox

$ docker volume ls
DRIVER              VOLUME NAME
local          7286369b084a1c4222123e0b52a4d8137dbeb0781
```

Data Volumes

Mounting a Data volume

Now, let us inspect the container and see what Docker did when we started this container.

Give the following command:

```
$ docker inspect container1

"Volumes": {
 "/data":
"/mnt/sda1/var/lib/docker/volumes/af0e7c1547fbcb42e8d5a5
14252e47cb754c8adf701e21b13b67a640d7a77883/_data"
 },
 "VolumesRW": {
 "/data": true
 },
```

Data Volumes

You can create and manage volumes outside the scope of any container.

Create volume:

```
$ docker volume create my-vol
$ docker run -d -v my-vol:/data busybox
```

List volumes:

```
$ docker volume ls
DRIVER              VOLUME NAME
local               my-vol
```

Host Data Volume

Mount a host directory as a data volume

```
$ docker run -d -P --name web -v /src/data:/data busybox
```

This command mounts the host directory, /src/data, into the container at /data. If the path /data already exists inside the container’s image, the /src/data mount overlays but does not remove the pre-existing content.

Host Data Volume

Mounting Individual Host Files

```
$ docker run -d --name test -v /tmp/file.txt:/data/file.txt busybox
```

The -v flag used so far can target a single file instead of entire directories from the host machine. This is done by mapping the specific file on each side of the container.

Volumes

Creating Dedicated Data Volume Containers

A popular practice with Docker data sharing is to create a dedicated container that holds all of your persistent shareable data resources, mounting the data inside of it into other containers once created and setup.

1. You first create a Data volume container
2. Create another container and mount the volume from the container created in Step 1.

```
$ docker run -it -v /data --name container1 busybox
$ docker run -it --volumes-from container1 --name container2 busybox
```

Docker Volume Plugins

These allow you to use third-party container data management solutions to persist data on your hosts or across multiple hosts. With these plugins it’s much easier to run stateful applications inside containers than before.

Example: Local Persist Volume Plugin for Docker

“This is great for creating standalone volumes and easily connecting them to different directories in different containers as a way to share data between multiple containers.”

Data Volumes - Examples

https://github.com/zreigz/docker-workshop/tree/master/volumes

Docker container networking

When you install Docker, it creates three networks automatically. You can list these networks using the command:

```
$ docker network ls
```

```
$ docker network ls
NETWORK ID          NAME                DRIVER
7fca4eb8c647        bridge              bridge
9f904ee27bf5        none                null
cf03ee007fb4        host                host
```

These three networks are built into Docker. When you run a container, you can use the --network flag to specify which networks your container should connect to.

Docker container networking

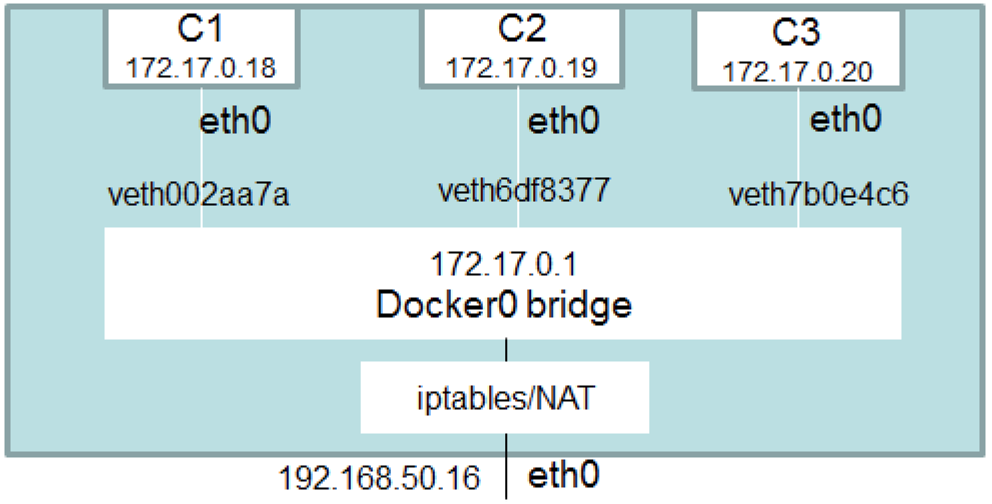
Docker0 bridge

- Default network automatically created when no additional options “--net“ or “-P” are specified
- Each container is addressed by a static IP address assigned by Docker
- Similar to what we have as default with KVM or VirtualBox
- Host can reach container with IP on the bridge
- But, outside traffic cannot reach the container

Docker container networking

Docker0 bridge

The docker0 bridge is virtual interface created by docker, it randomly chooses an address and subnet from the private range



```
$ docker network inspect bridge
```

Docker container networking

Host networking

Using host networking option during docker run command, the container can directly use the network of the host system. In other words, container will not create a seperate networking stack for itself. Container will still use process stack of its own, but not the network.

If you are using this host networking option, exposing ports etc with -p option does not make any sense. This is because, any services running in the container, will be directly on the host network. So everything is already exposed.

Host networking is enabled by using --net=host option with docker run command.

Docker container networking

User-defined networks

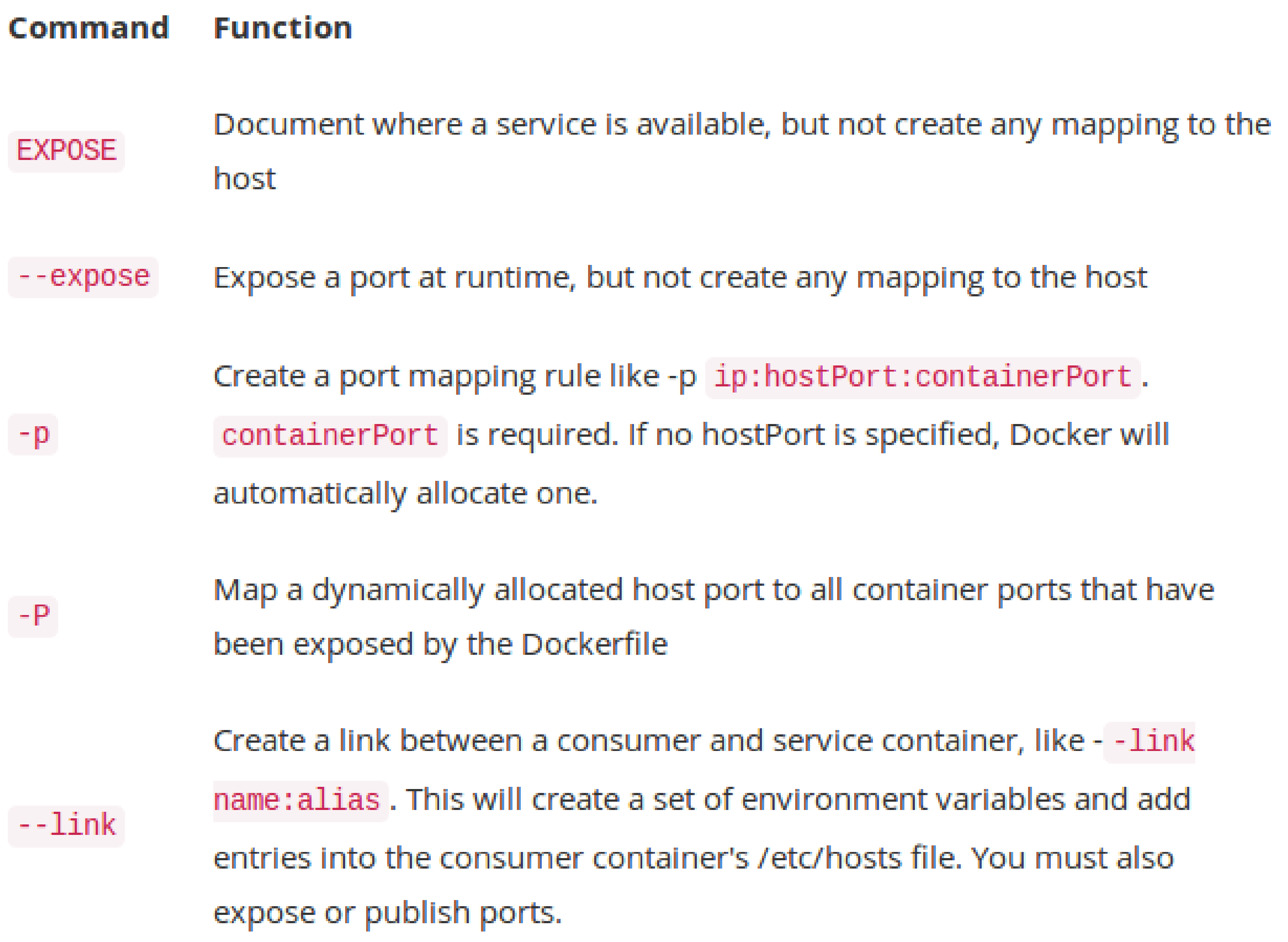
It is recommended to use user-defined bridge networks to control which containers can communicate with each other, and also to enable automatic DNS resolution of container names to IP addresses. Docker provides default network drivers for creating these networks.

Bridge networks

A bridge network is the most common type of network used in Docker. Bridge networks are similar to the default bridge network, but add some new features and remove some old abilities. The following examples create some bridge networks and perform some experiments on containers on these networks.

- $ docker network create --driver bridge isolated_nw
- $ docker run --network=isolated_nw -itd --name=container3 busybox

Connecting Docker Containers

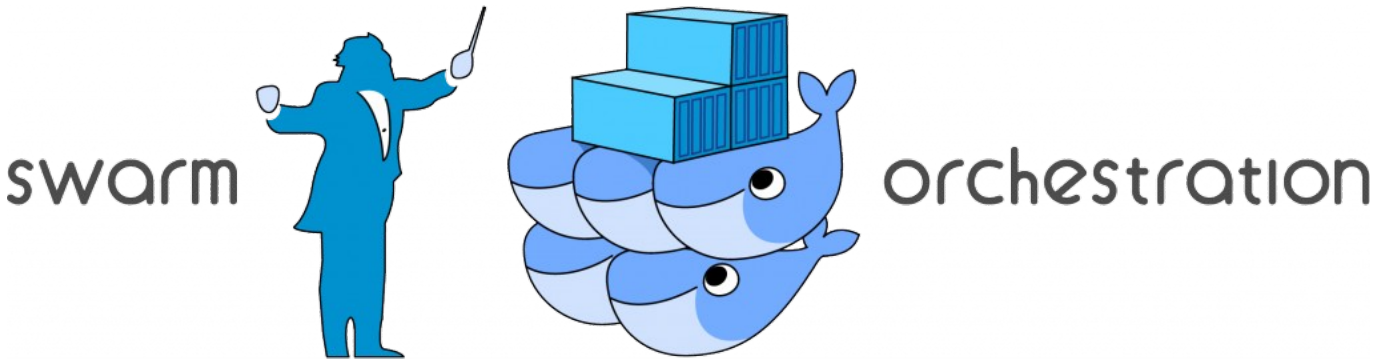


Command	Function
`EXPOSE`	Document where a service is available, but not create any mapping to the host
`--expose`	Expose a port at runtime, but not create any mapping to the host
`-p`	Create a port mapping rule like -p `ip:hostPort:containerPort`. `containerPort` is required. If no hostPort is specified, Docker will automatically allocate one.
`-P`	Map a dynamically allocated host port to all container ports that have been exposed by the Dockerfile
`--link`	Create a link between a consumer and service container, like -`-link name:alias`. This will create a set of environment variables and add entries into the consumer container's /etc/hosts file. You must also expose or publish ports.

Examples

https://github.com/zreigz/docker-workshop/tree/master/networking

Docker Swarm



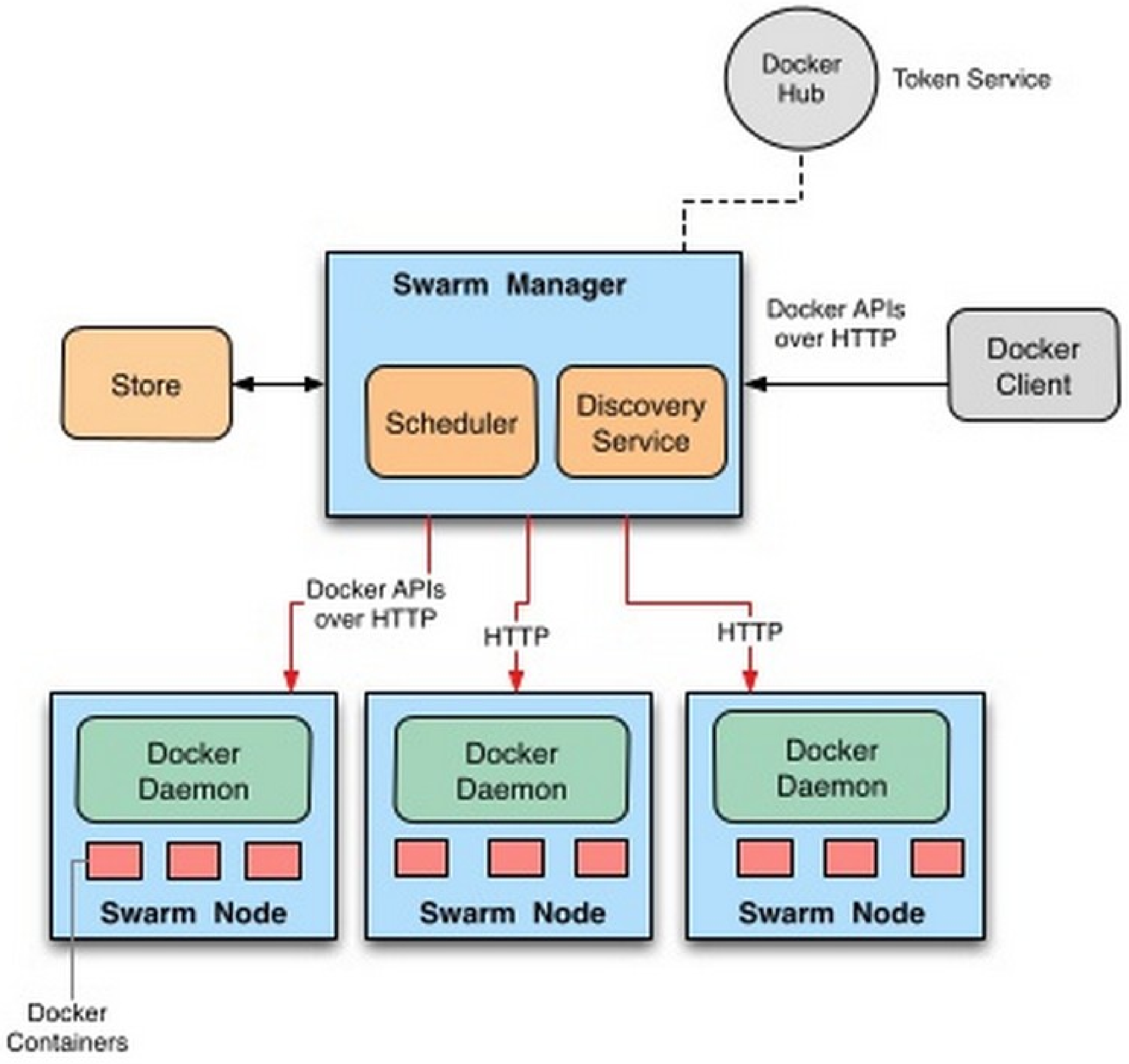
Swarm is the native clustering for Docker.

Docker Swarm is designed around four core principles: simple yet powerful with a “just works” user experience, resilient zero single-point-of-failure architecture, secure by default with automatically generated certificates, and backwards compatibility with existing components.

Docker Swarm

The Docker Swarm follows a decentralised design where nodes can handle any role in the cluster. The node specialisations to managers and workers are chosen at runtime. As the cluster must have at least one manager, the first node initializing the cluster is assigned as such. Subsequent nodes joining the cluster are usually added as workers but can be assigned as either. The flexibility means that the entire swarm can be built from a single disk image with little differentiation.

Docker Swarm



Docker Swarm

Create a new swarm cluster with the initialization command below. Replace the <manager private IP> with the private IP of the host you are creating the cluster on.

```
docker swarm init --advertise-addr <manager private IP>
```

When the swarm starts successfully, you can see a docker swarm join command in the output like the example below. Use the command on your other nodes to join them to the swarm.

```
docker swarm join \
--token <SWMTKN token> \
<manager private IP>
```

docker vs kubernetes

Easy and fast to install and configure

Installing Docker is as simple as any application available on the package manager system of your OS. With Swarm, deploying a node and telling it to join the cluster is all that is required. Along with the ease of use, the Swarm also provides flexibility by allowing any new node to join an existing cluster as either a manager or a worker and seamlessly promote or demote nodes between the two roles.

Takes some work to get up and running

Kubernetes requires a number of manual configurations to tie together its components such as etcd, flannel, and the docker engine. Installation instructions differ from OS to OS and provider to provider. Kubernetes also needs to know much of the cluster configuration in advance like the IP addresses of the nodes, which role each node is going to take, and how many nodes there are in total.

docker vs kubernetes

Functionality is provided and limited by the Docker API

The Swarm API provides much of the familiar functionality from Docker itself but does not fully encompass all of its commands. It supports many of the tools that work with Docker, however, if Docker API lacks a specific operation there is no easy way around it using Swarm.

Client, API and YAML definitions are unique to Kubernetes

Kubernetes uses its own client, API and YAML definitions which each differ from that of the standard Docker equivalents. In other words, you cannot use Docker CLI nor Docker Compose to define containers. When switch platforms, commands and YAML definitions will need to be rewritten.

docker vs kubernetes

Quick container deployment and scaling even in very large clusters

Docker Swarm is able to deploy containers faster than Kubernetes even in very large clusters and high cluster fill stages allowing fast reaction times to scaling on demand. New replicas can be started with a single update command.

Provides strong guarantees to cluster states at the expense of speed

Kubernetes is in comparison more of an all-in-one framework for distributed systems. Its complexity stems from offering a unified set of APIs and strong guarantees about the cluster state, which slows down container deployment and scaling.

docker vs kubernetes

Pros

- Easy and fast setup
- Works with other existing Docker tools
- Lightweight installation
- Open source

Cons

- Limited in functionality by what is available in the Docker API
- Limited fault tolerance

Pros

- Open source and modular
- Runs well on any operating systems
- Easy service organisation with pods
- Backed by years of expert experience

Cons

- Laborious to install and configure
- Incompatible with existing Docker CLI and Compose tools