Docker Internals

Anthony Ramirez
anthony@nebulaworks.com

nebulaworks

The Agenda

Day 1

- Linux Containers (namespaces, cgroups, CoWFS)
- Container Management and Runtime Systems
- Docker Engine

Day 2

- Docker Filesystem
- Images
- Storage Drivers
- Volume Plugins
- Hardening of Container Infrastructure
  - (daemon configuration, host security, trusted Images/content)

The Agenda

Day 2 continued

- Reimagining Infrastructure
  - Artifact
  - Application logistics
  - Containerization process
  - Platform characteristics

Day 3

- Considerations when adopting container tools
- Continuous deployment
- System Testing
- Troubleshooting containers
- Containerize your applications
- BYOLS
- Ecosystem

ANATOMY OF LINUX CONTAINERS

module 01

What features do containers have that other technologies don’t?

Section 1.1 | Containers don’t exist

Red container, pixarprinting.eu

At a high level, is it a VM?

- Get shell access?
- Own process stack
- Own network stack
- Can be root
- Install packages
- Run services
- Manipulate iptables

At a low level, is it a super-chroot?

- OS level virtualization
- Boot different OS
- Separate kernel mods?
- Need a service manager as PID1?
- Require its own system services?

- Processes are transparent to the host
  - VMs are opaque

Okay let’s see how the kernel sees them!

- Search the kernel source @ kernel.org
- Search for lxc == nada
- Containers are isolation abstractions
- General search for containers == ref to ACPI containers
- No actual code in the kernel for containers!

How can we construct an understanding of a container?

The container recipe for a Docker Linux container:

Ingredients

- 1 portion CGROUPS
- 1 portion NAMESPACES
- 1 portion Copy on Write (COW) FS

- What were motivations for containers?
- Didn’t we have isolation without Docker?

- Filesystem isolation with chroot (1979)
- FreeBSD Jails built up functionality such as namespaces for process IDs (2000)
- Solaris extended features (2004)
- Cgroups adopted some ideas that existed as a part of these technologies (2007)
- LXC (2008), LMCTFY, Docker, Rkt

Section 1.2 | cgroups v1

CGROUPS ===

- In 2006 Google engineers started a project under the name process containers
  - Vision to isolate (Memory, CPU, disk, I/O, network)
- Merged into kernel 2.6.24 in 2008
- Each subsequent controller has its own maintainers
- Primary controllers were cpu, memory and blkio
- Networks left to iptables and tc (traffic control)
- cgroup v2 release April, 2016 (only 3 controllers)

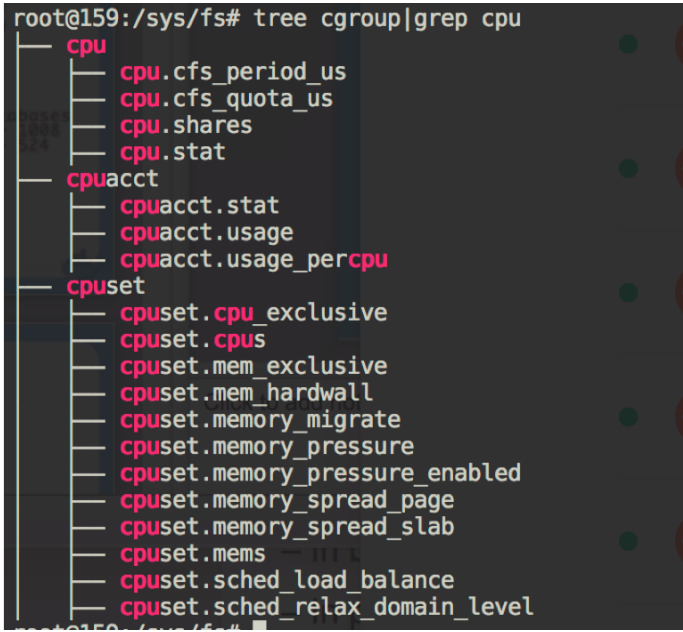
How do they work?

- cgroups provides a file interface for accessing cgroup functions
  - New procfs entries generated
    - For each process `/proc/pid/cgroup`
    - System-wide `/proc/cgroups`
- In cgroup v1 each subsystem has an independent hierarchy
- Each node in the tree == group of processes
- sys/fs/cgroup/

```
root@159:/sys/fs# tree cgroup|grep cpu
├── cpu
│   ├── cpu.cfs_period_us
│   ├── cpu.cfs_quota_us
│   ├── cpu.shares
│   ├── cpu.stat
├── cpuacct
│   ├── cpuacct.stat
│   ├── cpuacct.usage
│   ├── cpuacct.usage_percpu
├── cpuset
│   ├── cpuset.cpu_exclusive
│   ├── cpuset.cpus
│   ├── cpuset.mem_exclusive
│   ├── cpuset.mem_hardwall
│   ├── cpuset.memory_migrate
│   ├── cpuset.memory_pressure
│   ├── cpuset.memory_pressure_enabled
│   ├── cpuset.memory_spread_page
│   ├── cpuset.memory_spread_slab
│   ├── cpuset.mems
│   ├── cpuset.sched_load_balance
│   ├── cpuset.sched_relax_domain_level
```

How do they work?

- cgroup core found in kernel/cgroup.c (~6k lines of code)
  - Same for version 1 & 2
- cgroup controllers
  - 12 controllers in v1
- To use cgroups they must be mounted as any other filesystem is. Most systems based on the systemd runtime default this to /sys/fs/cgroups as the mount point



Example of mounting a controller (net_prio):
mount -t cgroup -o net_prio none /sys/fs/cgroup/net_prio

List of controllers

Name	Kernel Object name	Module
blkio	io_cgrp_subsys	block/blk-cgroup.c
cpuacct	cpuacct_cgrp_subsys	kernel/sched/cpuacct.c
cpu	cpu_cgrp_subsys	kernel/sched/core.c
cpuset	cpuset_cgrp_subsys	kernel/cpuset.c
devices	devices_cgrp_subsys	security/device_cgroup.c
freezer	freezer_cgrp_subsys	kernel/cgroup_freezer.c
hugetlb	hugetlb_cgrp_subsys	mm/hugetlb_cgroup.c
memory	memory_cgrp_subsys	mm/memcontrol.c
net_cls	net_cls_cgrp_subsys	net/core/netclassid_cgroup.c
net_prio	net_prio_cgrp_subsys	net/core/netprio_cgroup.c
perf_event	perf_event_cgrp_subsys	kernel/events/core.c
pids	pids_cgrp_subsys	kernel/cgroup_pids.c

Memory cgroup >> accounting

- Keeps track of pages used by each group:
  - file (read/write/mmap from block devices)
  - anonymous (stack, heap, anonymous mmap)
  - active (recently accessed)
  - inactive (candidate for eviction)
  - 4kb memory pages to store a reference

- Pages may be shared across multiple groups*
- Shared pages are only “charged” to one group

Memory cgroup >> limits

- Limits on a group may be hard or soft (optional)
- Soft limits are not enforced*
- Hard limits will trigger per-group OOM (out of memory) killer events
- The OOM event action can be customized (advanced)
  - Use controller Freeze on group/processes
  - Notify user space and kill process
  - Modify the limits and then move container back
  - unfreeze
- Limits may be applied to > physical, kernel, total memory (including swap)

Memory cgroup >> usage examples

```
mkdir /sys/fs/cgroup/memory/group0
```

- The tasks entry that is created under group0 is empty (processes are called tasks in cgroup terminology).

```
echo 0 > /sys/fs/cgroup/memory/group0/tasks
```

- The pid of the current bash shell process is moved from the memory controller in which it resides into group0 memory controller group.

```
echo 40M > /sys/fs/cgroup/memory/group0/memory.limit_in_bytes
```

You can disable the out of memory killer with memcgca:

```
echo 1 > /sys/fs/cgroup/memory/group0/memory.oom_control
```

Memory cgroup >> the fine print

- When the kernel issues or removes a page from a process it updates the counters metering the usage.
  - This adds some overhead
  - This is a system-wide setting enabled/disabled at boot time
- When multiple groups use the same page only the first process is “charged”*

(* Should the process terminate, the “charge” is moved to the other group using it)

cpu cgroup >> accounting

- Track user/system CPU time
  - May also track this at a per cpu level
- Track usage by CPU
- Limits on the CPU can not be set

cpu cgroup >> management

- Weights for a process may be given to influence scheduled time
- Processes may be pinned to specific CPUs
- Reserve CPUs for specific processes
- Prevent processes from being bounced between CPUs
  - Design to minimize NUMA (non-uniform memory access) system impacts
- Additional characteristics are tweakable
  - per zone memory pressure, process migration costs…

cpu cgroup >> examples

```
mkdir /sys/fs/cgroup/cpu/group0
```

- The tasks entry that is created under group0 is empty (processes are called tasks in cgroup terminology).

```
echo 512 > /sys/fs/cgroup/cpu/group0/cpu.shares
```

- This provides hints to the cfs for when CPU is under pressure.
- Ranges from 1 to 1024

```
cat /sys/fs/cgroup/cpuacct/cpuacct.usage_percpu
```

- view the usage on a cpu by cpu basis in the form of cpu time (in nanoseconds)

blkio cgroup >> accounting

- Tracks IO per group
  - Per block device
  - Segments reads and writes
  - Segments sync vs. async
- Tracks IO wait time
- Tracks IO queue depth
  - Also tracks the average queue depth
- Tracks block device idle time
- Tracks empty time
- Certain counters require “CONFIG_DEBUG_BLK_CGROUP=y” parameter to be configured (boot parameter in grub)

blkio cgroup >> management

- Set throttle (limits) on a group basis
  - Bytes per sec for reads and writes
  - Operations per sec for reads and writes
- Weights may also be applied on a group basis
  - By device type
  - Default weight for a device is 500: range is between 100-1000

blkio cgroup >> examples

```
<host>:~# mkdir -p /cgroup/blkio/test1/
```

- Creates a new blkio group

```
<host>:~# echo 1000 > /cgroup/blkio/test1/blkio.weight
```

- Set the weight for the group as a whole

```
<host>:~# cat /sys/fs/cgroup/blkio/test1/blkio.io_queued
```

- View outstanding IOs in the queue

Net_cls and net_prio cgroup >> special

- Automatically set traffic priority for traffic generated by processes in the group
- Only works for egress traffic
- Net_cls will assign traffic to a class
  - class then matched with tc/iptables, otherwise traffic just flows normally
- Net_prio will assign traffic to a priority
  - priorities are used by queuing disciplines

Devices cgroup >> special

the device cgroup is more of an access controller (ACL) than a resource controller

- Controls what the group can do on device nodes
- Does not use metering
- Permissions include read/write/mknod
- Typical use:
  - allow /dev/{tty,zero,random,null} …
  - deny everything else
- A few interesting nodes:
  - `/dev/net/tun` (network interface manipulation)
  - `/dev/fuse` (filesystems in user space)
  - `/dev/kvm` (VMs in containers, yay inception!)
  - `/dev/dri` (GPU)

The other cgroups available >>>

- HugeTLB (Translation Lookaside Buffer)
  - Controls the amount of "huge pages" usable by a process
  - Safe to ignore if you don’t know what huge pages are =)
  - More info: http://linuxgazette.net/155/krishnakumar.html
- Freezer
  - Allows to freeze/thaw a group of processes
  - Similar to mass SIGSTOP/SIGCONT
  - Caveat is that processes are completely unaware of the state change
  - Doesn’t impede ptrace/debugging
  - Specific use cases
    - cluster batch scheduling
    - process migration (CRIU project is a great example)
      - Checkpoint/Restore In Userspace: https://criu.org/
      - Experimental integration with Docker, but very cool: https://criu.org/Docker

The other cgroups available >>> ….cont'd

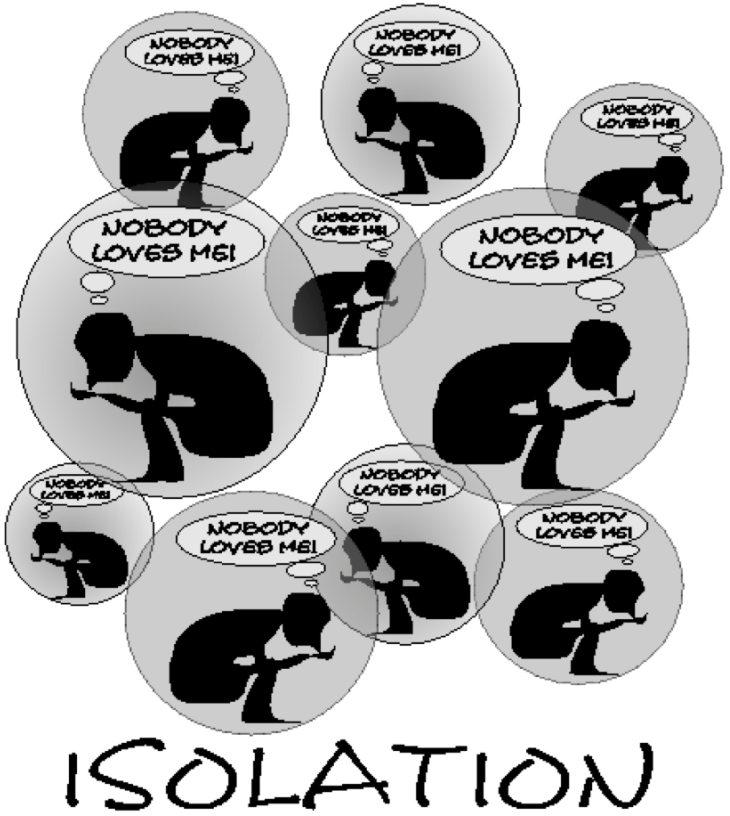
- perf_event
  - Allows the grouping of processes and threads to be monitored all at once with a perf tool
  - Groups in the hierarchy may possess only the commonly tunable parameters*
- PIDs
  - The anti-fork bomb control group
  - Provides the ability to limit the processes that maybe forked in a group
  - System wide limit on PIDs ~4million**
  - View current system specific limit `>> cat /proc/sys/kernel/pid_max`

Subtleties

- PID 1 is placed at the root of each hierarchy
- New processes are placed in their parent's groups
- Groups are materialized by one (or multiple) pseudo-fs
  - typically mounted in /sys/fs/cgroup
- Groups are created by mkdir in the pseudo-fs
- To move a process:
  `echo $PID > /sys/fs/cgroup/.../tasks`
- The cgroup wars: systemd vs. cgmanager (LXC) vs. ...

Section 1.3 | namespaces

Namespaces



Namespaces, a brief history >>>

- Dev started in 2002 and spanned more than a decade. Work began with 2.4.19 in 2002 and completed in 2012 with merge into 3.8
- There are currently 6 namespaces available:
  - pid
  - net
  - mnt
  - uts
  - ipc
  - user
- A process must be placed in each space to achieve full isolation
- A process can be created in Linux by the fork(), clone() or vclone() system calls.
- user namespace was added as the (currently) final namespace piece, and merged into 3.8 kernel in 2012.
- Docker was launched around March, 2013, once that was finalized.

Namespaces a brief history >>> …cont'd

Clone flag	Kernel Version	Required capability
CLONE_NEWNS	2.4.19	CAP_SYS_ADMIN
CLONE_NEWUTS	2.6.19	CAP_SYS_ADMIN
CLONE_NEWIPC	2.6.19	CAP_SYS_ADMIN
CLONE_NEWPID	2.6.24	CAP_SYS_ADMIN
CLONE_NEWNET	2.6.29	CAP_SYS_ADMIN
CLONE_NEWUSER	3.8	No capability is required

Namespaces usage

- In order to support namespaces, 6 flags (CLONE_NEW*) were added. These flags (or a combination of them) can be used in clone() or unshare() system calls to create a namespace.

Namespaces API consists of these 3 system calls:

- clone() - creates a new process and a new namespace; the newly created process is attached to the new namespace.
  - The process creation and process termination methods, fork() and exit(), were patched to handle the new namespace CLONE_NEW* flags.
- unshare() – gets only a single parameter, flags. Does not create a new process; creates a new namespace and attaches the calling process to it.
- setns() - a new system call, for attaching the calling process to an existing namespace; prototype: int setns(int fd, int nstype);

pid (process) namespace

- Processes within a PID namespace only see processes in the same PID namespace
- Each PID namespace has its own numbering starting at 1
- When PID 1 goes away, the whole namespace is killed
  - Sending SIGKILL signal does not kill system process 1, regardless of in which namespace the command was issued
    - (Need to send the kill to the "true" PID of the process.)
  - When a process dies, all its orphaned children will now have the process with PID 1 as their parent (child reaping)
- Those namespaces can be nested; up to 32 levels
  - A process ends up having multiple PIDs: one per namespace in which it's nested

net (network stack) namespace

In theory >

- A network namespace is logically another copy of the network stack, with its own routing tables, firewall rules, and network devices.
- Processes within a given network namespace get their own private network stack, including:
  - network interfaces (including lo)
  - routing tables
  - iptables rules
  - sockets (ss, netstat)
- You can move a network interface from a netns to another:
  - ip link set dev eth0 netns PID

net (network stack) namespace

In practice>

- Typical use-case:
  - use veth pairs (two virtual interfaces acting as a cross-over cable)
  - eth0 in container network namespace
  - paired with vethXXX in host network namespace
  - all the vethXXX are bridged together (Docker calls the bridge docker0)
- But also: the magic of *--net=host*
  - Attaches a container to localhost (and more!)
- Each newly created network namespace includes only the loopback device.

mnt (mount points, filesystems) namespace

- Processes can have their own root fs (à la chroot)
- Processes can also have “private” mounts
  - `/tmp` (scoped per user, per service...)
  - masking of `/proc`, `/sys`
  - NFS auto-mounts (why not?)
- Mounts can be totally private, or shared
- No easy way to pass along a mount from a namespace to another

uts (hostname) namespace

- The UTS namespace provides a way to get information about the system with commands like uname or hostname.
- UTS namespace was the most simple one to implement.
  - It's a simple namespace, but how we can name different ones, and lend logical/functional naming, which gives us our concept of it being a "separate" host/VM/container.
  - This is one of the things that helps us build our conceptual models of containers!
- The uts_ns object includes an object (new_utsname struct) with 6 members:
  - sysname
  - nodename
  - release
  - version
  - machine
  - domainname
- Available functions: gethostname / sethostname

ipc (system v ipc) namespace

- The IPC namespace is used for isolating System V IPC objects, and POSIX message queues.
- Each IPC namespace has its own message queue filesystem, which is only visible to processes residing in the same IPC namespace.
- Similar in nature to the uts namespace this namespace allows for isolation of process communication within the space.
- Allows a process (or group of processes) to have own:
  - IPC semaphores
  - IPC message queues
  - IPC shared memory

user (UIDs) namespace

- Allows to map UID/GID; e.g.:
  - UID 0→1999 in container C1 is mapped to
  - UID 10000→11999 on host;
  - UID 0→1999 in container C2 is mapped to
  - UID 12000→13999 on host; etc.
- Avoids extra configuration in containers
- UID 0 (root) can be squashed to a non-privileged user
- Security improvement……but the devil is in the details

Namespace manipulation

- Namespaces are created with the clone() system call
  i.e. with extra flags when creating a new process
- Namespaces are materialized by pseudo-files
  `/proc/<pid>/ns`
- When the last process of a namespace exits, it is destroyed but can be preserved by bind-mounting the pseudo-file
- It is possible to "enter" a namespace with setns()
  exposed by the nsenter wrapper in util-linux

Section 1.4 | CoW filesystem

CoW ==

- density
- speed
- space

How do Copy on Write File Systems Benefit Containers?

- Container creation is fast
- Storage (filesystem) keeps track of changes
- There are a lot of options:
  - AUFS, overlay, overlay2 (file level)
  - device mapper thinp (block level)
  - BTRFS, ZFS (FS level)
- Considerably reduces footprint and “boot” times

aufs

Advanced Multi-layered Unification Filesystem
Another Union File System

AUFS

- Implements a union mount for Linux file systems.
- Unites several directories into a single virtual filesystem.
- Directories are branches. Combine multiple branches in a specific order
- You generally have:
  - at least one read-only branch (at the bottom)
  - exactly one read-write branch (at the top)

(But other fun combinations are possible too!)

https://en.wikipedia.org/wiki/Aufs

AUFS

When opening a file -

- With O_RDONLY - read-only access:
  - look it up in each branch, starting from the top
  - open the first one we find
- With O_WRONLY or O_RDWR - write access:
  - look it up in the top branch; if it's found here, open it
  - otherwise, look it up in the other branches; if we find it, copy it to the read-write (top) branch, then open the copy

That "copy-up" operation can take a while if the file is big!

AUFS

When deleting a file –

- A whiteout file is created
  (if you understand the concept of "tombstones" as related to distributed date stores, this is similar)

```
# docker run ubuntu rm /etc/shadow
# ls -la /var/lib/docker/aufs/diff/$(docker ps --no-trunc -lq)/etc
total 8
drwxr-xr-x 2 root root 4096 Jan 27 15:36 .
drwxr-xr-x 5 root root 4096 Jan 27 15:36 ..
-r--r--r-- 2 root root    0 Jan 27 15:36 .wh.shadow
```

AUFS >> under the hood

- To see details about an AUFS mount:
  - look for its internal ID in /proc/mounts
  - look in `/sys/fs/aufs/si_.../br*`
  - each branch (except the two top ones) translates to an image

```
dockerhost# grep c7af /proc/mounts
none /mnt/.../c7af...a63d aufs rw,relatime,si=2344a8ac4c6c6e55 0 0

dockerhost# grep . /sys/fs/aufs/si_2344a8ac4c6c6e55/br[0-9]*
/sys/fs/aufs/si_2344a8ac4c6c6e55/br0:/mnt/c7af...a63d=rw
/sys/fs/aufs/si_2344a8ac4c6c6e55/br1:/mnt/c7af...a63d-init=ro+wh
/sys/fs/aufs/si_2344a8ac4c6c6e55/br2:/mnt/b39b...a462=ro+wh
/sys/fs/aufs/si_2344a8ac4c6c6e55/br3:/mnt/615c...520e=ro+wh
```

AUFS

- Creation of AUFS mount() is fast resulting is speedier container creation

- Scenarios where AUFS is slow:
  - Modifying very large files at lower levels require a write to top
  - with many layers + many directories in PATH (dynamic loading, anyone?)
- When starting the same container 1000x, the data is loaded only once from disk, and cached only once in memory (but entries will be duplicated)

device mapper

Device Mapper

- Kernel based framework that leverages **thin provisioning** and snapshotting
- Written by RedHat to support docker
- Operates at the block level
- RAID
- encrypted devices
- snapshots (i.e. copy-on-write)

Device Mapper >>> details

- Copy-on-write happens on the block level (instead of the file level)
- Performance slightly slower than AUFS
- Each container and each image gets its own block device
- At any given time, it is possible to take a snapshot:
  - of an existing container (to create a frozen image)
  - of an existing image (to create a container from it)
- If a block has never been written to:
  - it's assumed to be all zeros
  - it's not allocated on disk (hence "thin" provisioning)

Device Mapper >>> details

- Two storage areas are needed: one for data, another for metadata
- "data" is also called the "pool"; it's just a big pool of blocks (Docker uses the smallest possible block size, 64 KB)
- "metadata" contains the mappings between virtual offsets (in the snapshots) and physical offsets (in the pool)
- Each time a new block (or a copy-on-write block) is written, a block is allocated from the pool
- When there are no more blocks in the pool, attempts to write will stall until the pool is increased (or the write operation aborted)

Device Mapper >>> performance

- By default, Docker puts data and metadata on a loop device backed by a sparse file
- This is great from a usability point of view (zero configuration needed)
- But terrible from a performance point of view:
  - each time a container writes to a new block,
  - a block has to be allocated from the pool,
  - and when it's written to,
  - a block has to be allocated from the sparse file,
  - and sparse file performance isn't great anyway

btrfs Binary Tree File System

BTRFS >>> notes

- Pronunciation:
  - Btrfs (B-tree file system, pronounced as "butter F S", "better F S", "b-tree F S", or simply by spelling it out) is a file system based on the copy-on-write (COW) principle, initially designed at Oracle Corporation for use in Linux.
  - Details/background at https://en.wikipedia.org/wiki/Btrfs
- Do the whole "copy-on-write" thing at the filesystem level
- Create* a "subvolume" (imagine mkdir with Super Powers)
- Snapshot* any subvolume at any given time
- BTRFS integrates the snapshot and block pool management features at the filesystem level, instead of the block device level

BTRFS >>> notes

- It should be present even if the container is not running
- Data is not written directly, it goes to the journal first (in some circumstances*, this will affect performance)
- BTRFS works by dividing its storage in chunks
- A chunk can contain data or metadata
- You can run out of chunks (and get "No space left on device") even though df shows space available (because the chunks are not full)
- Quick fix:
  - # btrfs filesys balance start -dusage=1 /btrfs_device

BTRFS >>> tuning

- Not much to tune
- Keep an eye on the output of btrfs filesys show!

This filesystem is doing fine:

```
# btrfs filesys show
Label: none  uuid: 80b37641-4f4a-4694-968b-39b85c67b934
        Total devices 1 FS bytes used 4.20GiB
        devid    1 size 15.25GiB used 6.04GiB path /dev/xvdc
```

BTRFS >>> tuning

This one, however, is full (no free chunk) even though there is not that much data on it:

```
# btrfs filesys show
Label: none  uuid: de060d4c-99b6-4da0-90fa-fb47166db38b
        Total devices 1 FS bytes used 2.51GiB
        devid    1 size 87.50GiB used 87.50GiB path /dev/xvdc
```

overlayfs

Overlayfs >>> notes

This is just like AUFS, with minor differences:

- only two branches (called "layers”)
- but branches can be overlays themselves

Overlayfs >>> notes

- You need kernel 3.18
- On Ubuntu*:
  - go to http://kernel.ubuntu.com/~kernel-ppa/mainline/
  - locate the most recent directory, e.g. v3.18.4-vidi
  - download the linux-image-..._amd64.deb file
  - dpkg -i that file, reboot, enjoy

Overlayfs >>> notes

- Images just have a root subdirectory (containing the root FS)
- Containers have:
  - `lower-id` → file containing the ID of the image
  - `merged/` → mount point for the container (when running)
  - `upper/` → read-write layer for the container
  - `work/` → temporary space used for atomic copy-up

Overlayfs >>> notes

- Implementation detail: identical files are hardlinked between images (this avoids doing composed overlays)
- Not much to tune at this point
- Performance should be slightly better than AUFS:
  - no stat() explosion
  - good memory use
  - slow copy-up, still (nobody's perfect)

VFS >>> notes

- Creates an entire copy, not copy on write
- No copy on write.
- Doesn't rely on those fancy-pesky kernel features
- Space inefficient, slow
- Might be useful for production setups
  (If you don't want / cannot use volumes, and don't want / cannot use any of the copy-on-write mechanisms!)

Section 1.5 | Lab

Lab Objectives

- Create a linux “container” using only basic linux utilities
  (i.e. no help from a container runtime; well not a lot anyway)
- Play with cgroups and perform basic function tests
- Launch container with a simple bash script

git clone https://ghe.nebulaworks.com/nebulaworks/docker-internals-lab.git
User: training
PW : docker2018

Further info:

Much of this info in this first module is drawn from:

- "Cgroups, namespaces, and beyond: what are containers made from?"
  - Lecture by Jérôme Petazzoni, Docker Inc.
  - https://youtu.be/sK5i-N34im8 (55 minutes)

CONTAINER VS. CONTAINER

module 02

Section 2.1 | container management and runtime systems

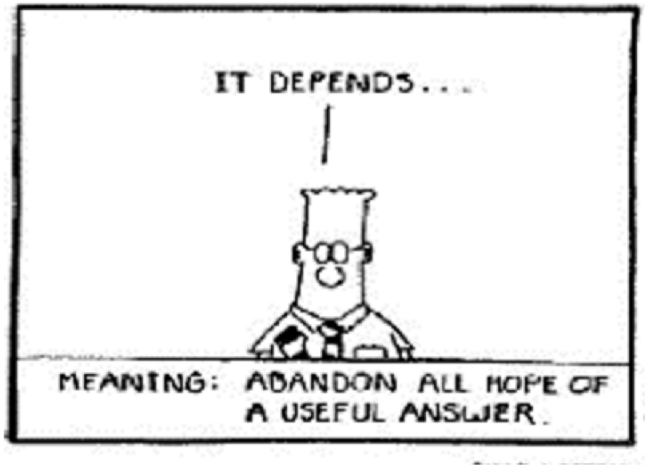
What is a container runtime?

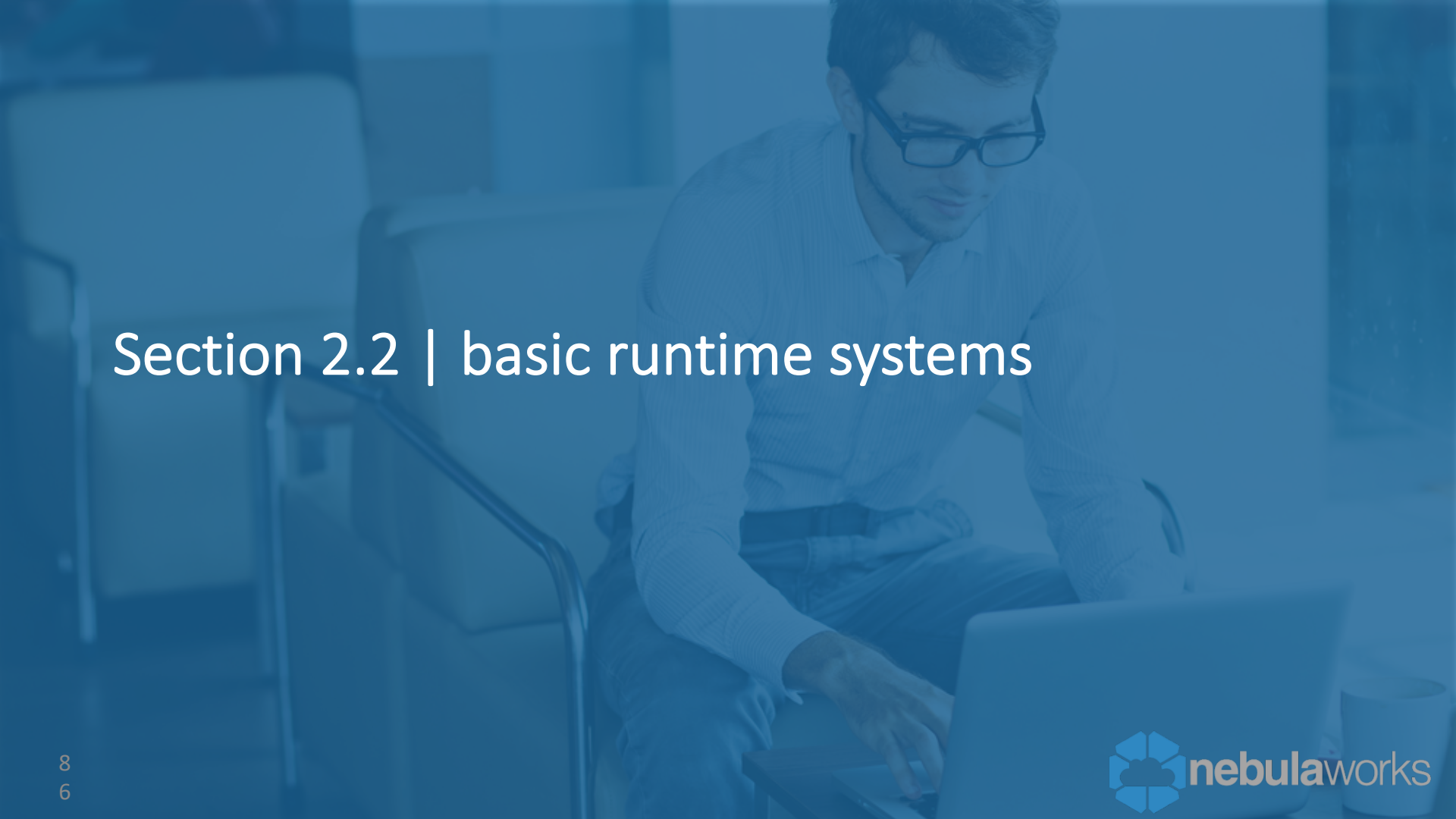
- Create containers from linux primitives
- Manipulate containers it created
- Tool based. Should support Unix design philosophy:
  - composability (modularity/working well with other tools)
  - simplicity (do one thing, and do it well)
- Should be open
- Generally adhere to some industry recognized specification

What is a container management system???

- Perform all the functions associated with the runtime
- Manage the entire container lifecycle
- Manage basic container support service: networking, volumes, images
- Act as a service manager for containers

Ok, ok, which one is better?!? It depends…





Section 2.2 | basic runtime systems

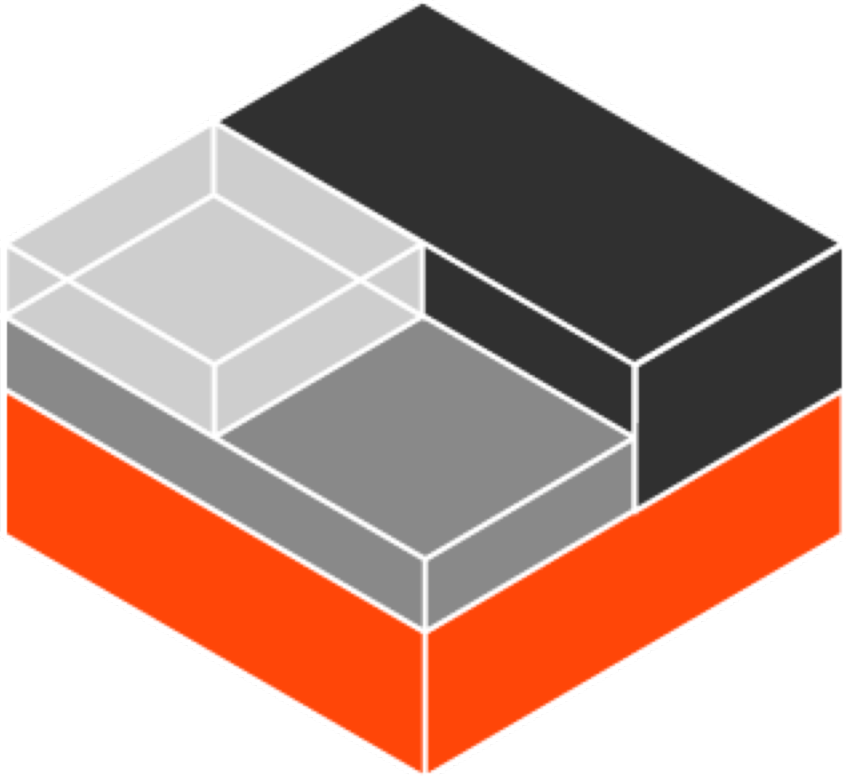
Contenders in the space >>>

- LXC v1
  - Development in Aug 2008, stable release v 1.1.5
  - Written in C, python, lua, shell
- Rkt
  - Development began Dec 2014
  - Written in Go
- runC
  - Development began Jun 2015
  - Written in Go

LXC

“LXC is a userspace interface for the Linux kernel containment features.
Through a powerful API and simple tools, it lets Linux users easily create
and manage system or application containers.”

- https://linuxcontainers.org/lxc/introduction/



LXC the granddaddy of linux container runtimes

LXC >>> key goals, features and facts:

- Provides a set of command line utilities to assist in the creation and manipulation of “linux containers”
- Use a VM-like approach to containers to facilitate ops management process
- Works with just the mainline linux kernel (no third-party patches required)
- Once upon a time was the backend for the docker engine

LXC >>> key terms

- operating-system-level virtualization
  - a server virtualization method in which the kernel of an operating system allows the existence of multiple isolated user-space instances, instead of just one.
- os-level container
  - Also known as “full system containers”. This type of container differ from docker type containers in that they do not employ a CoW filesystem to host the rootfs of the chrooted mount.

LXC – container life cycle

- lxc-create: sets up a container (obtains root fs and creates a startup config)
- lxc-start: boots the container (will drop you into a console by default)
- lxc-console: attach to the console of a container running in the background
- lxc-stop: shuts down a running container
- lxc-destroy: destroys the container and related file system objects created by lxc-create

rkt

“rkt (pronounced "rock-it") is a CLI for running app containers on Linux. rkt is designed to be secure, composable, and standards-based.”

https://github.com/coreos/rkt

rkt – the stage 0 cli

rkt >>> key goals, features and facts:

- Modularity – rkt is architected in stages (image fetching, cgroup and networking setup, and execution) that can have different implementations, providing separation of privileges as well as concerns
- Composability – rkt is not a daemon, is not the parent process of all your containers and is composable with other tools. Natively runs appc images built with acbuild
- Security – It has Intel Clear Container*, SELinux and TPM support, as well as image signature validation
- Cross Container Compatible – able to run Docker images

rkt >>> key terms

- Pod
  - appc defines the pod as the basic unit of execution. A pod is a grouping of one or more app images (ACIs), with some additional metadata optionally applied to the pod as a whole. The images in a pod execute with a shared context, including networking.*
- ACI
  - The image format defined by appc or Application Container Image (ACI). An ACI is a simple tarball bundle of a rootfs and an Image Manifest, which defines things like default execution parameters and default resource constraints.

rkt – container life cycle

- Prepare (stage 0) | rkt fetch: creates/prepares the “pod” filesystem
  - Fetching the specified ACIs Generating a Pod UUID
  - Generating a Pod Manifest
  - Creating a filesystem for the pod
  - Setting up stage 1 and stage 2 directories in the filesystem
  - Unpacking the stage 1 ACI into the pod filesystem
  - Unpacking the ACIs and copying each app into the stage2 directories
- Run (stage 1/2) | rkt run: creating the necessary cgroups, namespaces and mounts and combine with stage 0 “pod” file system to launch the pod
  - Read the Image and Pod Manifests.
  - Create and enter network namespace
  - Set up any external volumes
  - Launch PID 1 process

rkt – container life cycle

- Garbage Collection (stage -1) | rkt gc: after container reaches an exited status have background garbage collection services cleanup pod and return resources to the system.
  - Pass 1: mark
    All directories found in $var/run are tested for exited status by trying to acquire a shared advisory lock on each directory. When a directory's lock cannot be acquired, the directory is skipped as it indicates the pod is currently executing.
  - Pass 2: sweep
    The sweep operation takes the time the pod was marked as the beginning of the "dwell" grace period, and discards exited pods at the expiration of that period. This grace period currently defaults to 30 minutes.

runC

“runC is a CLI tool for spawning and running containers according to the OCP (Open Container Project) specification.”

- http://runc.io/

runC – a lightweight universal runtime container

runC >>> key goals, features and facts:

- Embeddable: Containers are started as a child process of runC and can be embedded into various other systems without having to run a Docker daemon
- Battle Hardened: runC is built on libcontainer, the same container technology powering millions of Docker Engine installations
- Compatible with Docker: Docker images can be run with runC
- Maintained by the Open Container Project foundation (OCP) which is managed by the linux foundation
- “The goal of runC is to make standard containers available everywhere”
  - Solomon Hykes

runC >>> key terms

- filesystem bundle (OCI bundle)
  - A Standard Container bundle contains all the information needed to load and run a container. This MUST include the following artifacts:
    - config.json : contains configuration data
    - A directory representing the root filesystem of the container. This directory MUST be referenced from within the config.json file.

Side note >>> The principles of software plumbing

THOU SHALT

I

Re-use and improve existing plumbing.

II

Make new plumbing easy to re-use and improve.

III

Follow the unix principles: make small simple tools, not big complicated ones.

IV

Define standard interfaces for assembling larger systems.

runC – container life cycle

- Create filesystem bundle: need to obtain/create a rootfs and a config.json spec file
  - Easiest method to obtain a rootfs is to use docker to export a rootfs from an existing docker container with the docker export command
  - Then use the runc spec command to obtain a generic config.json for the filesystem bundle to be complete
- Start container: *runc start* - start up a container based on a filesystem bundle and drop the user into an interactive session
- Kill container: *runc kill* - sends a SIGTERM kill to a container's init process that is being run by runC
- Delete container: *runc kill* - deletes any resources held by a container often used with detached containers

Section 2.3 | complex container management systems

Contenders in the space >>>

- systemd-nspawn
  - Development began Apr 2012
  - Written in C
- rkt with systemd
  - Development began
  - Written in C and Go Dec 2014
- docker engine
  - Development began Jan 2013
  - Written in Go

systemd-nspawn | When lxc is just too hard for you!

“We believe a modern system and service manager should natively know the concept of a container. Containers should be a central facet of server management and the concept of it should transcend the layers of the OS stack, all the way from the application layer down to the kernel. As the glue between that all, it’s a responsibility for systemd to integrate containers into the OS.”

- Lennart Poettering

systemd-nspawn >>> strengths

- Availability: is part of the systemd init system and so will be part of most mainline flavors of linux within the next couple of years
- Simplicity: designed to be as simple to use as chroot, ie it should require no special configuration to run or understanding of underlying system architecture
- Familiarity: leverages existing tool sets already in use to manage system VMs
- Integration: as a part of the init system nspawn is already integrated into system service like logging, security policies, etc

systemd-nspawn >>> weaknesses

- Limited functionality –
  - no use of cgroups with nspawn native containers
  - No API
  - No CoW
  - No native image format
- Dependent on init service
- Not meant for production

rkt w/ systemd | power up to a full management “systemD”

“I believe in the rkt model…Integrating container and service management, so that there’s a 1:1 mapping between containers and host services is an excellent idea. Resource management, introspection, life-cycle management of containers and services – all that tightly integrated with the OS; that’s how a container manager should be designed.”

- Lennart Poettering
  https://coreos.com/blog/rkt-hits-1.0.html

rkt w/ systemd >>> strengths

- Compatibility: relying on rkt's strength to run multiple container image formats. Will work on any modern linux system with the systemd init system as the system init manager
- Security: designed with a security first principle and for multitenant use case. Also uses signed images format and system to ensure image authenticity
- Composable: follows the Unix tool philosophy by delivering a single purpose scriptable binary that allows the user to rely on standard system utilities to manage containers

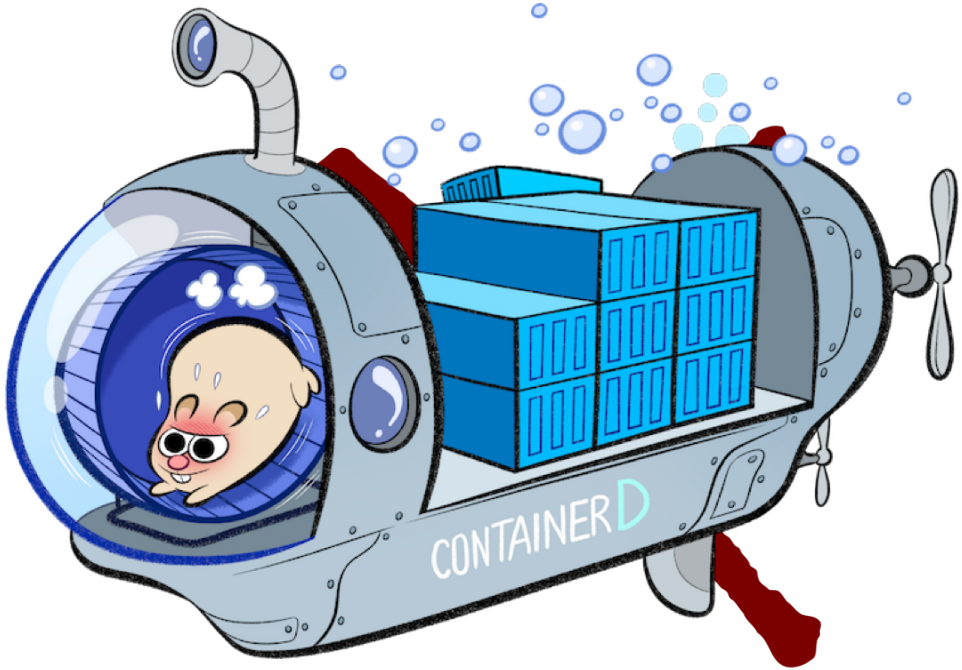
rkt w/ systemd >>> weakness

- Complexity: rkt by itself is only a runtime and requires an init system to run containers as a service. This requires additional knowledge of configuring systemd unit services. The default unit of operation with rkt is not an app container but a pod.
- Incompatibility: by default rkt uses a non-docker image format for its container images known as an aci. While CoreOS provides the tooling to convert images, differences in the implementation may mean features and meta information may not be fully translated.
- Optimization: Designed best to work with the CoreOS operating system. Running it as the primary runtime on other flavors of linux are not currently commercially supported.

~~docker engine~~ …containerD

“Containerd is a daemon to control runC, built for performance and density. It includes a daemon with a command line client to manage containers on one machine.”

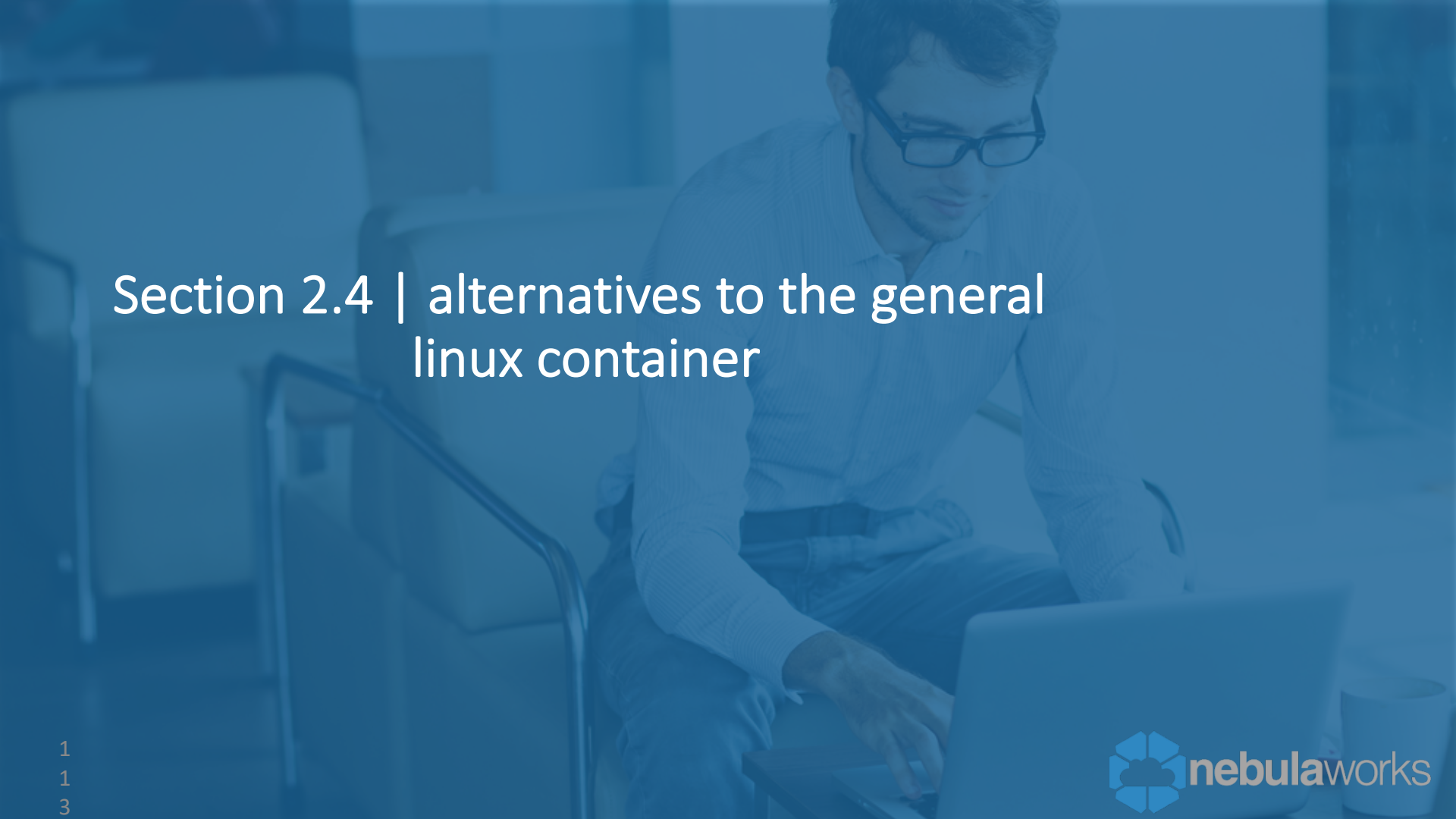
- https://containerd.tools/



containerD >>> strengths

- First Mover: containerD is fully based on the core docker engine bits and thus is currently the most widely adopted container management system available.
- Community: large community of developer are constantly adding and testing the code. Their code base acts as the foundation of the "neutral" open container initiative.
- Portability: the containerD management system is be adapted to run multi-platform containers such as windows native containers and IBM power series containers, etc
- Simplicity: by far the most easy to use container management system for developers

	systemd-nspawn	rkt w/ systemd	containerD
established API	X	Y	Y
plugin architecture	X	X	Y
unix way	Y	Y	Y
std image format	X	Y/X	Y
runs as service	Y	X	Y
img: pull	X	Y	X
img: push	X	X	X
img: build	X	X	X



Section 2.4 | alternatives to the general linux container

OpenVZ

“OpenVZ (Open Virtuozzo) is an operating system-level virtualization technology for Linux. It allows a physical server to run multiple isolated operating system instances, called containers, virtual private servers (VPSs), or virtual environments (VEs.) OpenVZ is similar to Solaris Containers and LXC.” - https://en.wikipedia.org/wiki/OpenVZ

OpenVZ –

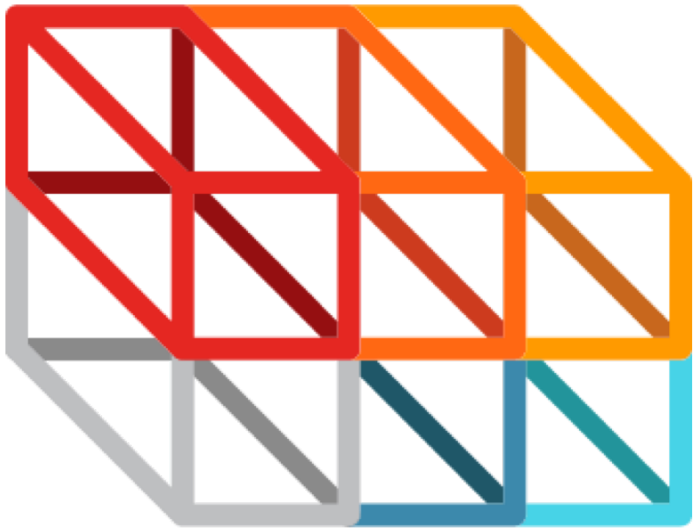
key facts >>>

- Initial release – 2005
- Written in C
- Runs on a patched kernel that provides virtualization, isolation, resource management, and checkpointing
- Uses a 2-lvl cpu scheduler to allowing finer control of cpu constraints
- Allows overcommit of physical memory
- Integrate live migration functionality
- Behave more like a full system container
- Commercial version available from virtuozzo

Solaris Zones aka solaris containers*

“Elastic Container Infrastructure
Simple and proven. Securely deploy and operate containers with bare metal speed on container-native infrastructure, your cloud or ours.”

- https://www.joyent.com/



~~Solaris Zones~~ → triton containers

key facts >>>

- Initial release ~ Jun 2015
- Written in C
- Provides 3 flavors of zones (containers): linux/Unix(LX), docker, cross-platform(KVM)
- Based on the illumos type 1 hypervisor which builds on the open Solaris and linux kvm virtualization stacks
- Zones are defined in a 3 step process
  - Configure: Defines the networking and privilege sets for the zone.
  - Install packages: Constructs, installs, and initializes packages for the zone.
  - Run: Boots the zone and starts services.

Hyper-V container and Windows Containers

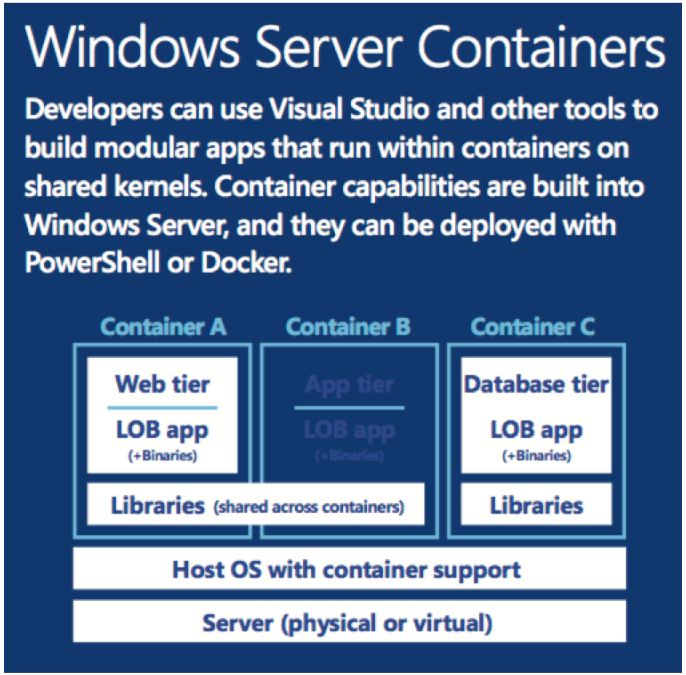
- @jessfraz docker core maintainer

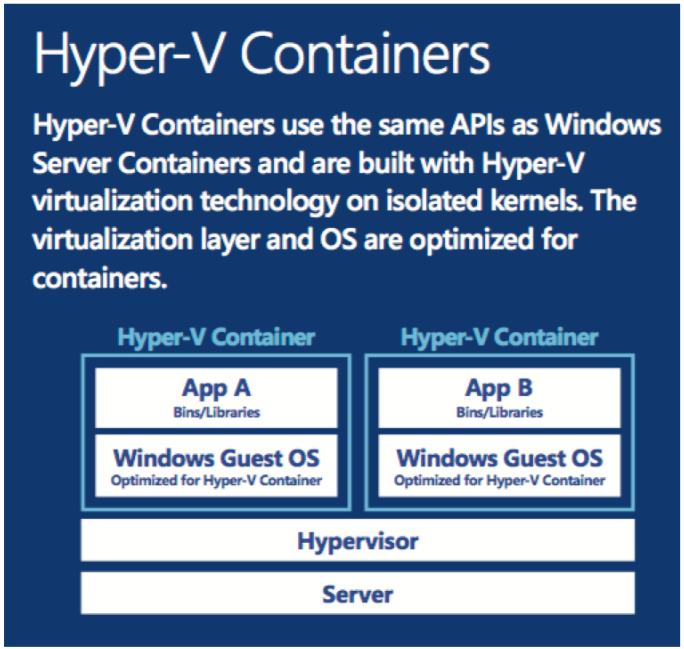
Hyper-V container and Windows Containers

key facts >>>

- Initially released as part of the 2016 server release
- Likely written in C
- Currently available as part of the windows server 2016
- Two container flavors: windows container, hyper-v container
- Technology has actively been a part of Azure since at least 2014
- Compatible with Docker tools and provides facilities to run both linux and windows based containers

Hyper-V container and Windows Containers





Unikernel – virtual library operating system

“What are unikernels?...

Unikernels are specialized, single-address-space machine images constructed by using library operating systems.”

- http://unikernel.org/

Section 2.5 | best container platform to use? …it depends

nebulaworks

Performance

Same kernel primitives <

Same system schedulers <

Shared kernel <

Same hardware <

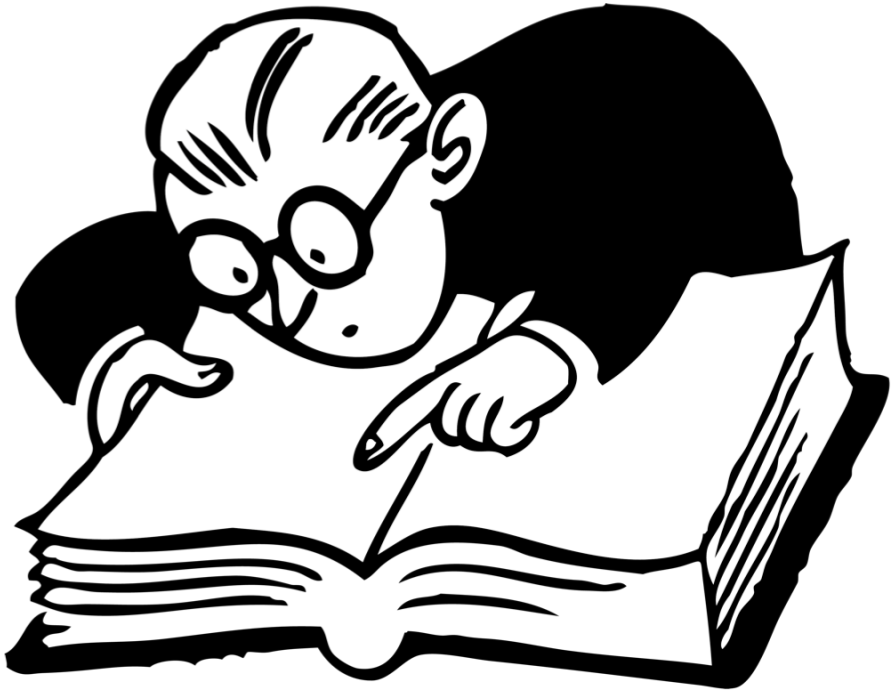
NEBULAWORKS

ease of use

- picking a view point
- how it fits into a workflow
- ease of sharing the results
- ease of maintenance

documentation

- does it exist?
- does it keep pace with the code?
- can you find answers?
- is there a user community?



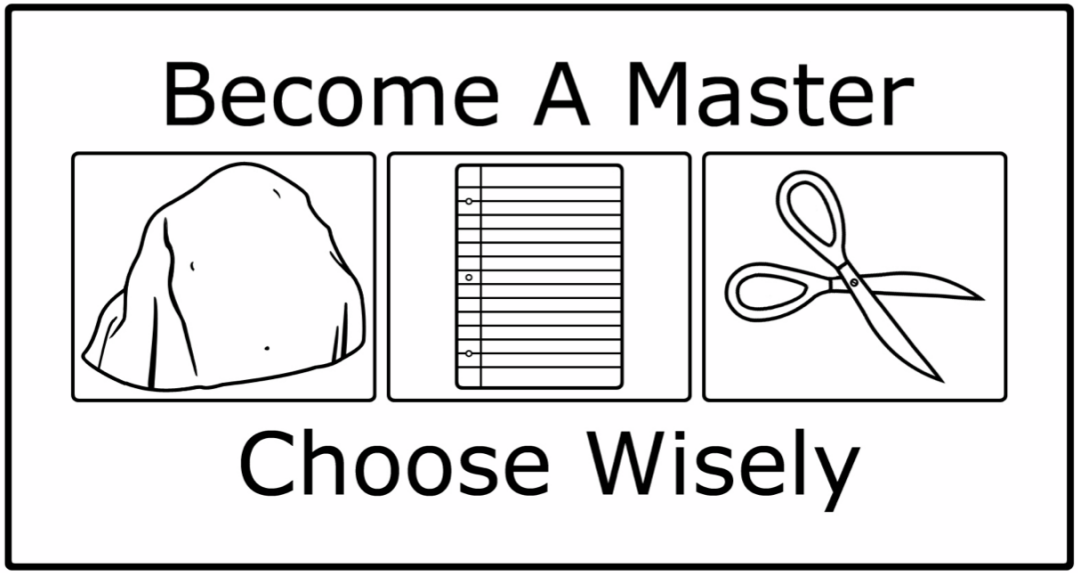
devs

- simple to develop micro services
- simple to build container images
- large image library

ops

- simple to deploy in production
- built-in security
- easy integration with monitoring and logging systems

conclusions



Section 2.6 | lab

Lab Objectives

- Run various runtimes: containerD, lxc, rkt

IN-DEPTH WITH DOCKER ENGINE

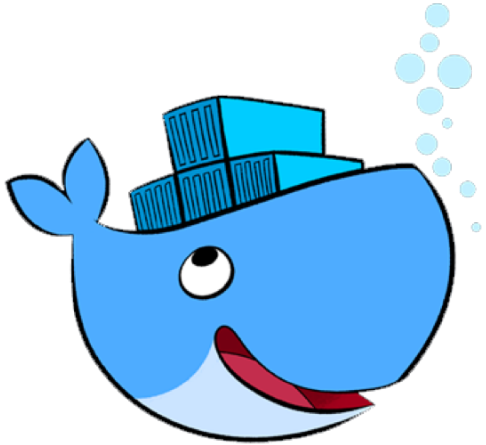
module 03

Section 3.1 | engine overview

this year docker turned five, and the whale that started the container revolution continues to evolve at an insane pace. since its start in 2013 it has >

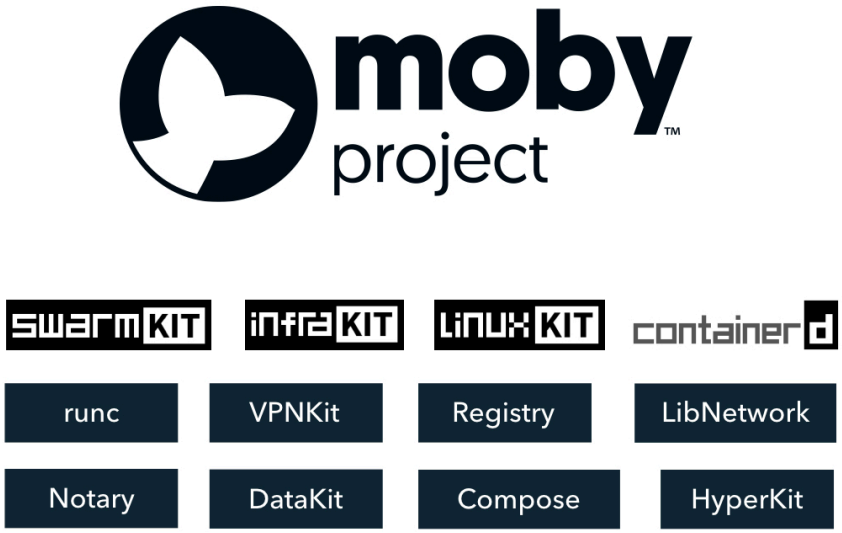
- been accepted onto every major public cloud platform
- seen more the 8 billion! container images downloaded
- released an end to end container platform

At the heart of this is still the docker engine

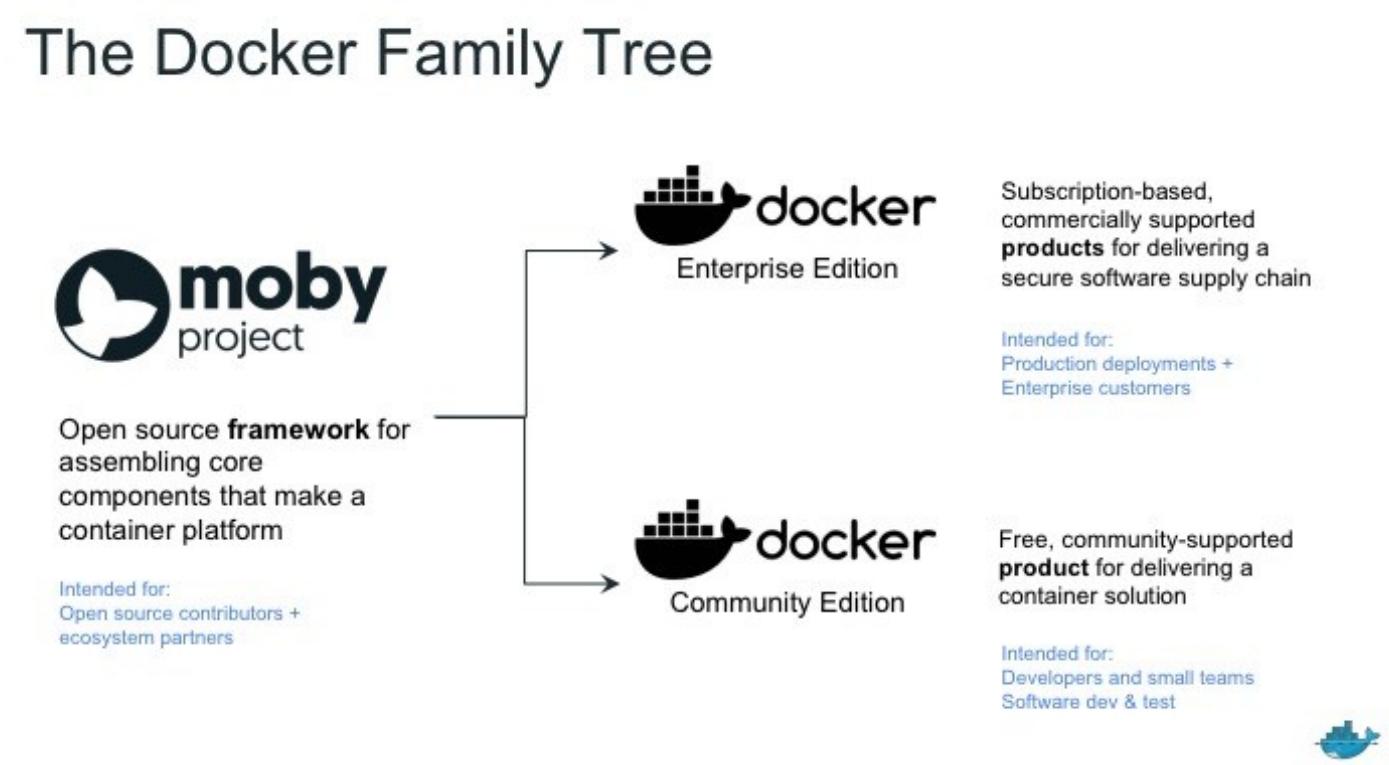


dotCloud and the docker Project


moby
project

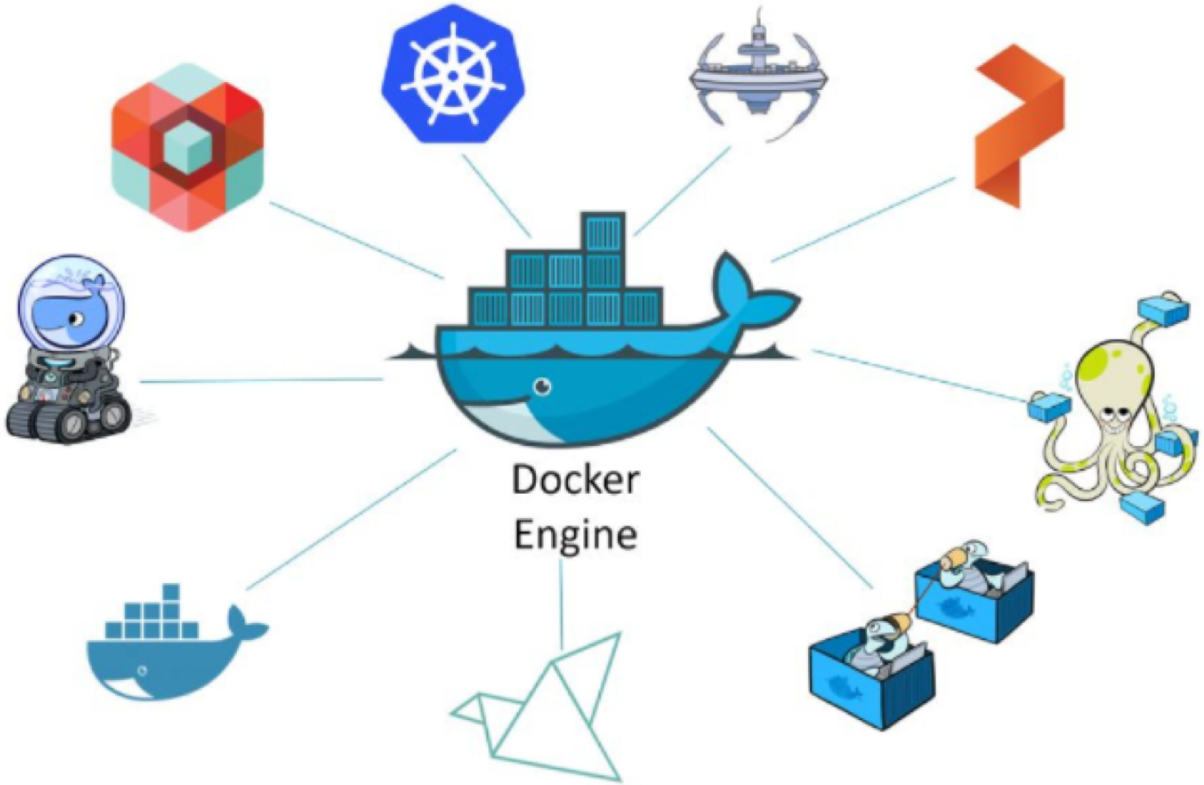


Source : Moby Project Docs



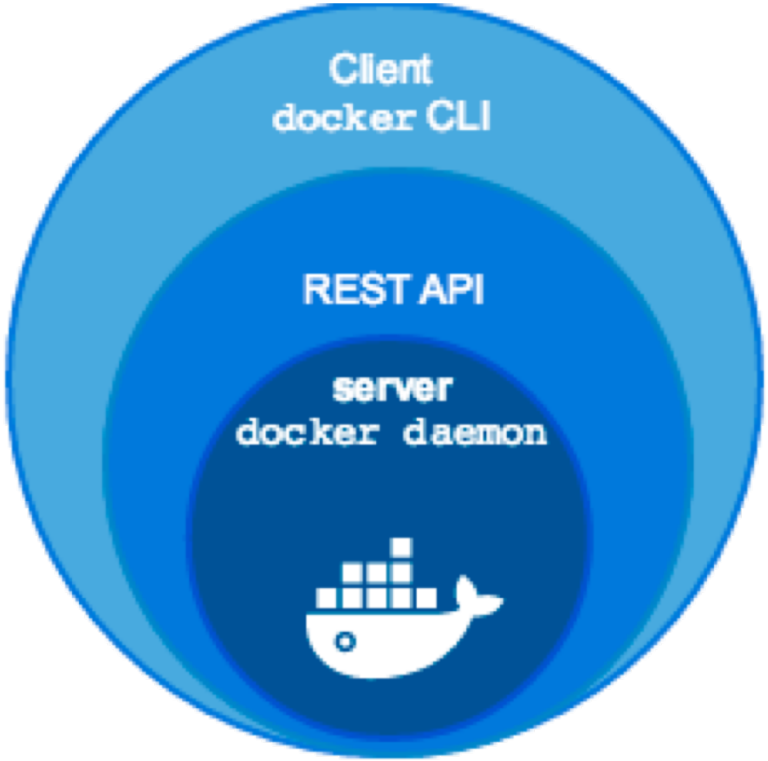
Source : Docker Documentation

≈ docker engine



Source*: Docker Deep D*ive Nigel Poulton

Client-Server Architecture



Element	Responsibility
Builder	Parsing Dockerfiles and building images based on the contents of those Dockerfiles.
Client	Providing a user interface (CLI) to interact with Docker daemon.
Daemon	Image management, REST API, security, core networking.
Container Management System	Manages container lifecycle, Image push and pull, creation/modification and deletion of interfaces, manages networking namespace
Container Runtime	Manages an execution engine that “creates” containers
Event Bus	Providing a publish-subscribe interface for events
Registry	Storing images and providing an interface for pulling those images

docker daemon

Functionality

- Implements the Docker API
- Manages Images
- Builds images
- Core Networking
- Orchestration
- Security

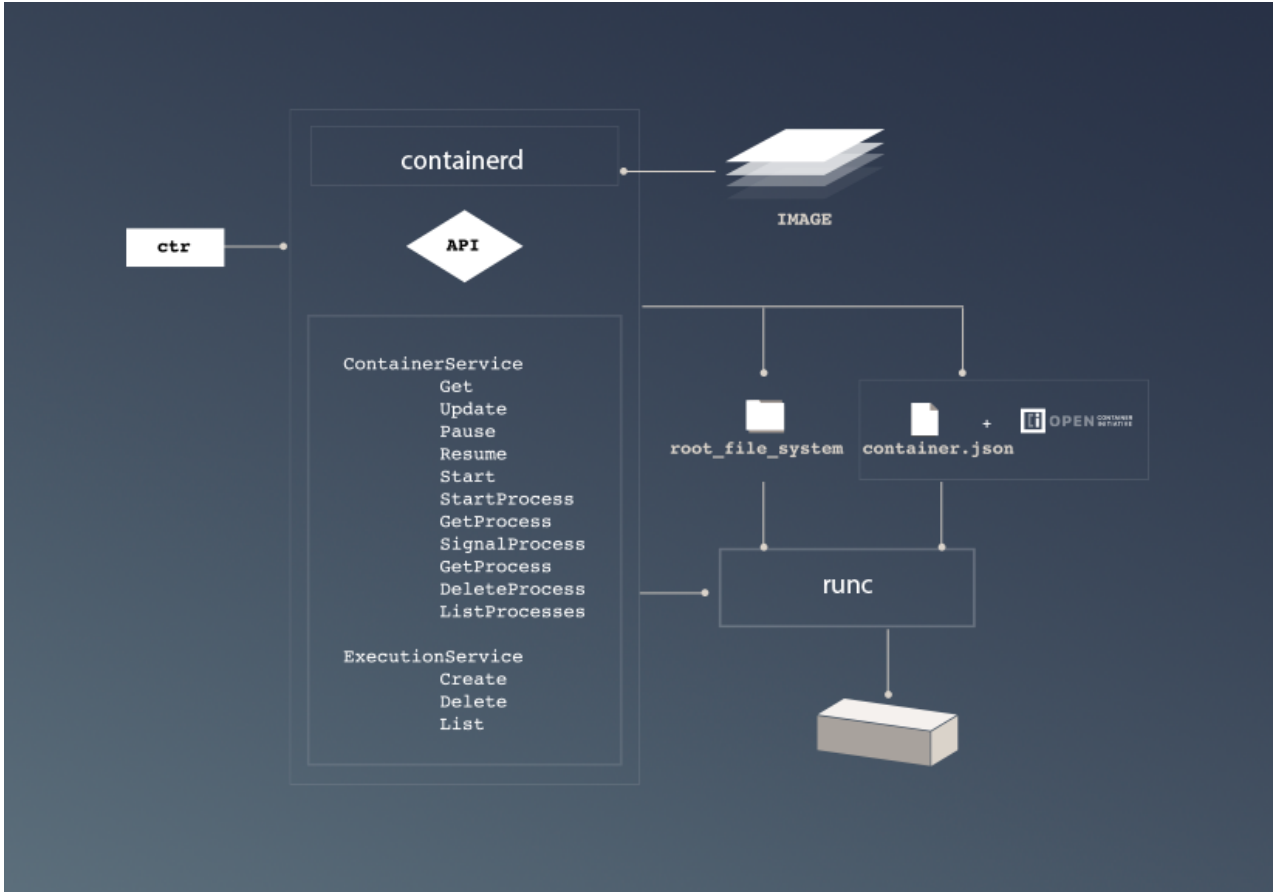
Function

- Default container management system
- OCI compliant
- Handles container lifecycle operations.
- Talks to and manages runc
- Push and Pull

Requirements

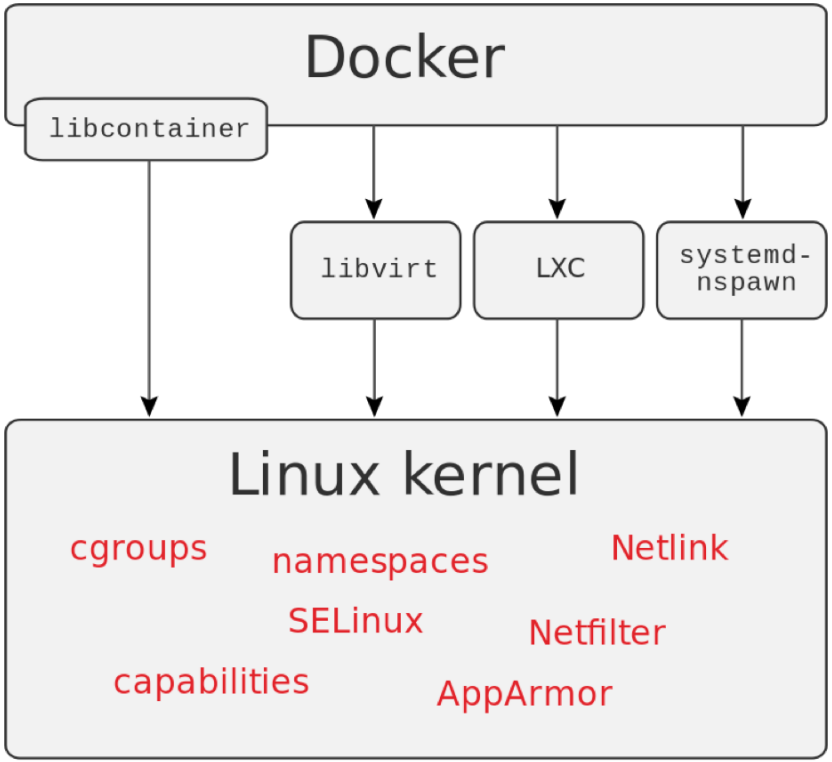
- OCI-compliant image
- OCI-compliant container runtime (runc)

containerd


containerd
IMAGE
ctr
API
ContainerService
Get
Update
Pause
Resume
Start
StartProcess
GetProcess
SignalProcess
GetProcess
DeleteProcess
ListProcesses
ExecutionService
Create
Delete
List
root_file_system
container.json
OPEN
runc

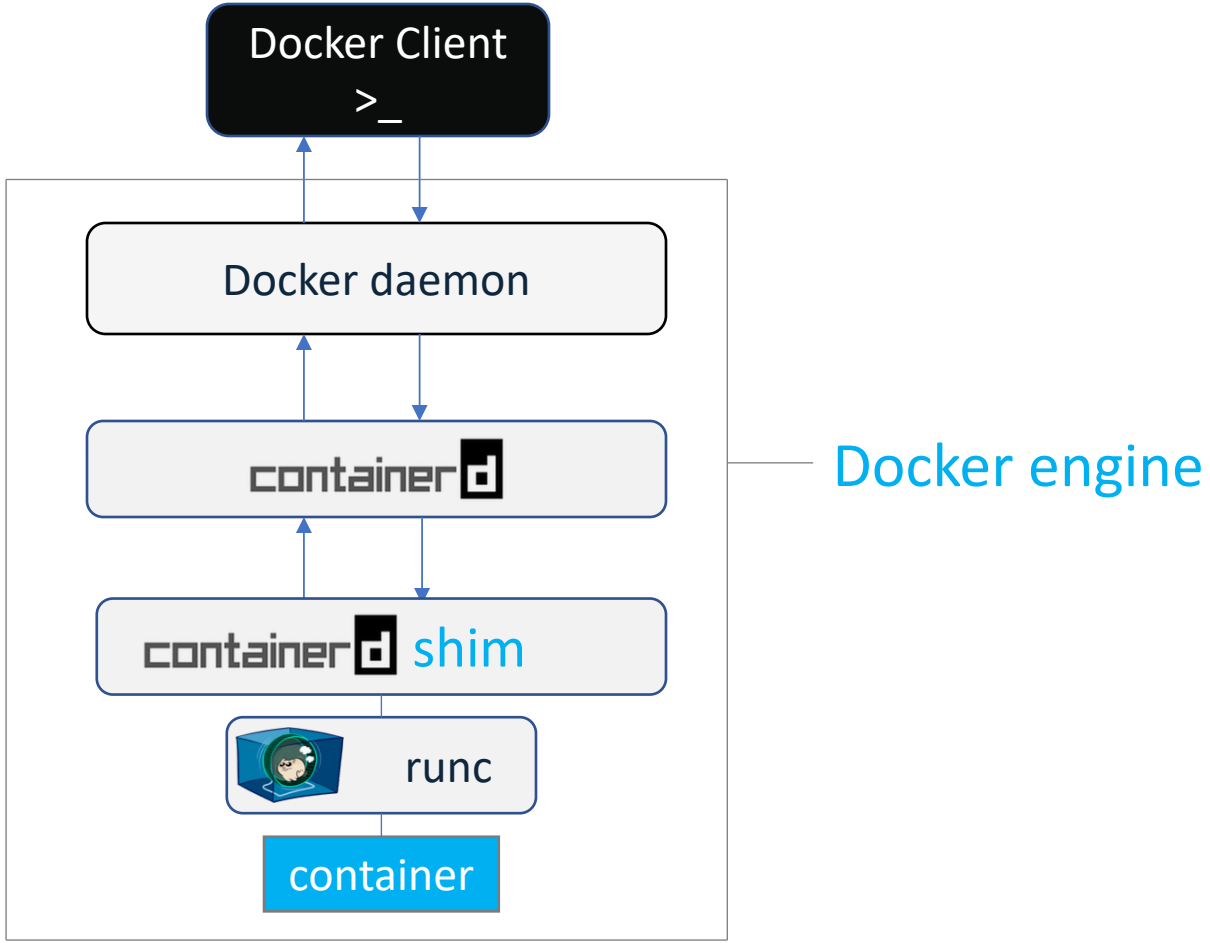
containerd - shim

- It's a process that launches runC.
  - runC will execute the container and then exit.
  - The container's process becomes a child of the shim.
  - 1 to 1 mapping
- Responsibilities.
  - Keeps containers alive!
    - Keeps any STDIN and STDOUT streams open (in case of daemon restart).
  - Tells containerd if a container has exited.

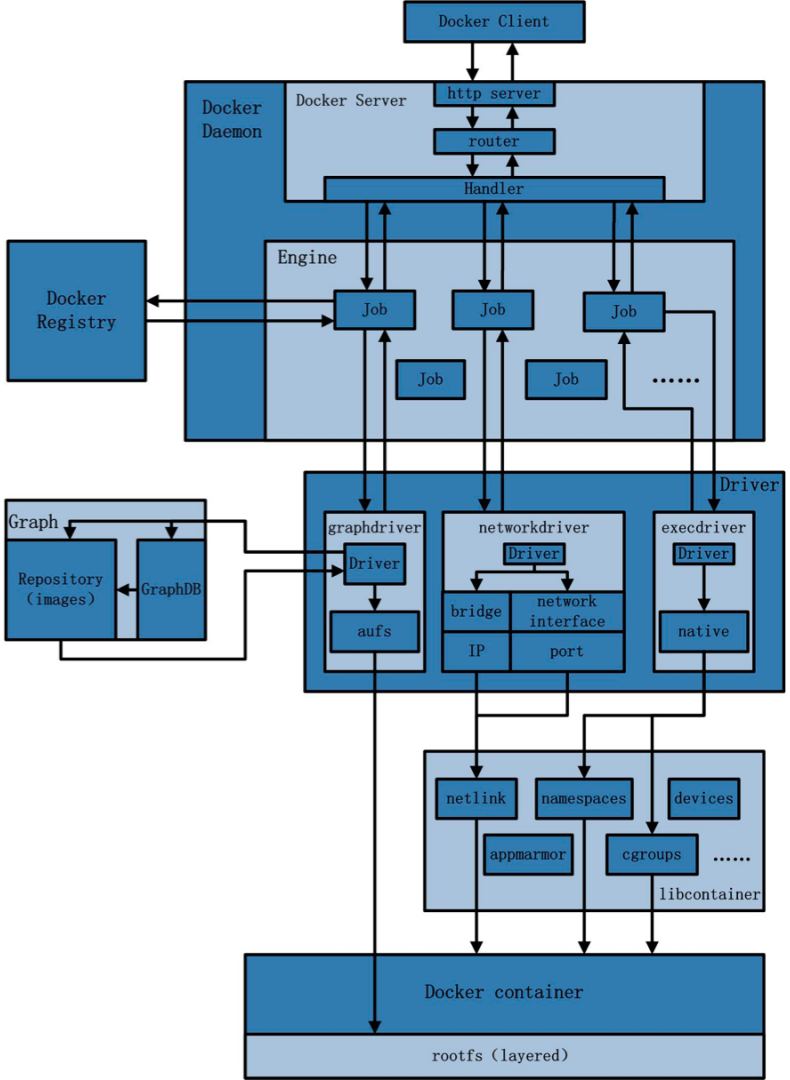

Docker
libcontainer
libvirt
LXC
systemd-
nspawn
Linux kernel
cgroups
namespaces
Netlink
SELinux
Netfilter
capabilities
AppArmor

runC

- Default container runtime for the moby projct.
- Light-weight portable container runtime.
- Provides all the code necessary to interact with system features.
- No dependency on the docker platform
- Requires an OCI-bundle

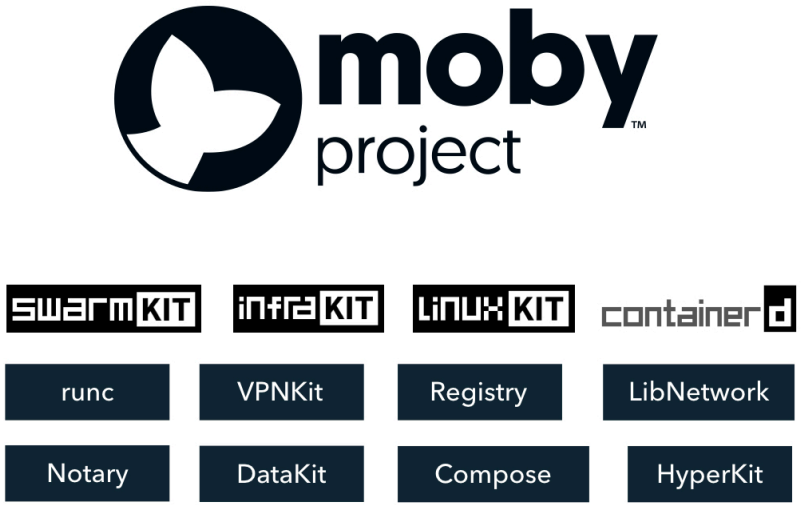

Docker Client
>_
Docker daemon
containerd
containerd shim
runc
container
Docker engine

detailed view of docker engine process and data flows.

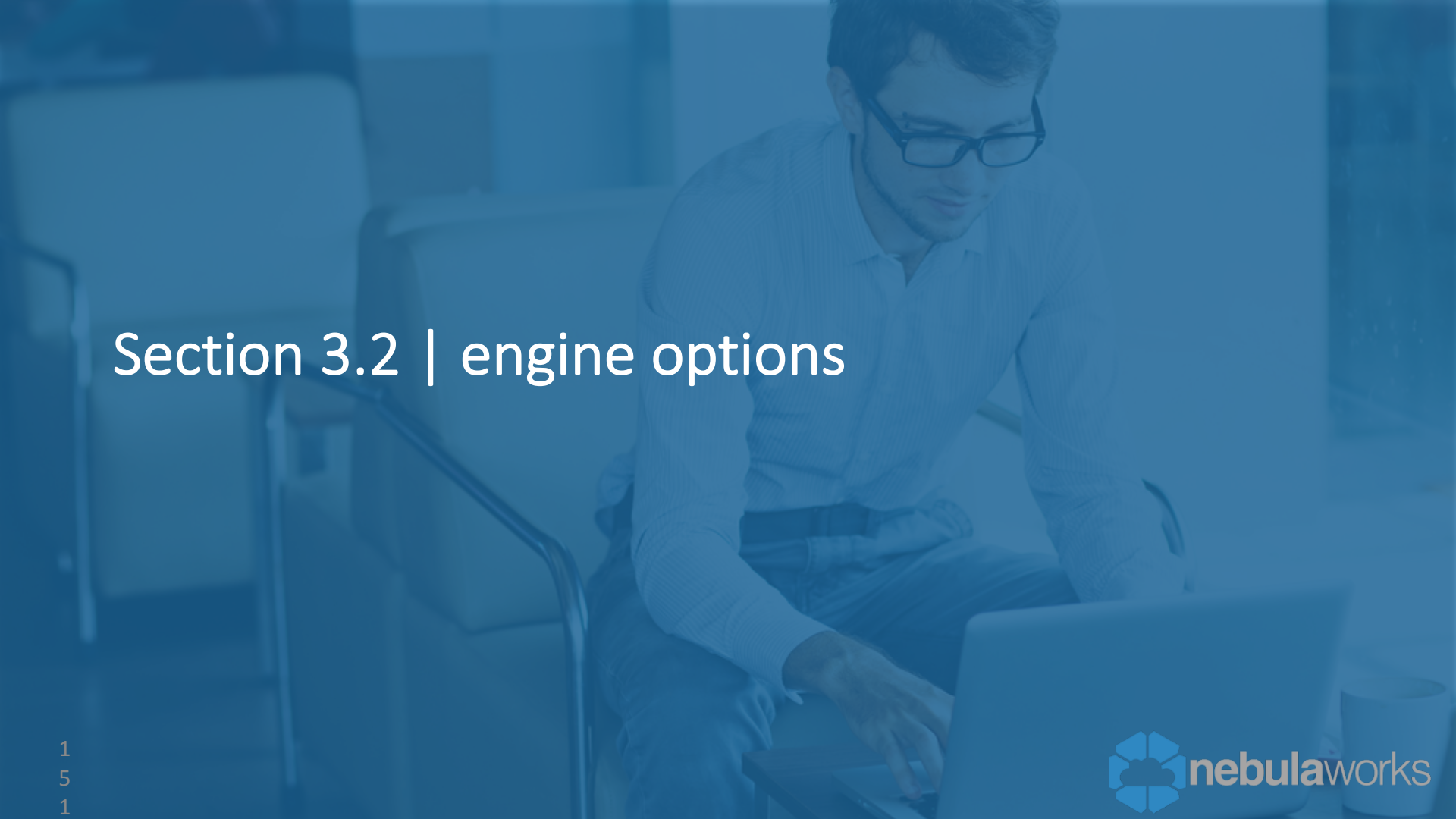


The modular architecture of the Moby Project has allowed the decoupling of various functions and has allowed the overall architecture to become more robust.

the components are still under rapid evolution as docker designs a truly cross platform system to run and manage containers



Source : Moby Project Docs



Section 3.2 | engine options

dockerd (daemon)

Two ways:

- Edit the `daemon.json file`
- `$ dockerd COMMAND`
- flags in this category define options that effect how interactions work with the docker engine in daemon mode

daemon.json

```
{
    “authorization-plugins”: [],
    “bridge”: “”,
    “cluster-advertise”: “”,
    “debug”: true,
    “dns”: []
}
```

/etc/docker/daemon.json

global engine communication options:

- affect ways in which the docker engine interacts with the api requests from the client and how it reports to the host system itself.

- `-D, --debug` : runs the docker with debug output mode providing ultra verbose output. This option may be reconfigured without service restart.
- `-G, --group string`: sets the group for the unix socket. By default this is set to docker. Members of the group may run docker commands without being root.
- `-H, --host list` : define what daemon socket(s) to connect to; three types of sockets are currently available to linux: tcp, unix, fd

operational

global engine system options, affecting the ways in which the docker engine interacts with the host system and remote registries :

- `--ip ip`: default IP (0.0.0.0) to use when binding container ports
- `--dns list` : configures docker to use specific DNS server (default [])
- `--label list` : sets key=value pairs associated to the daemon as meta data information
- `--insecure-registry list` : enables insecure registry communication with the provided list of registries (default [])

operational

global engine system options, that secures the communication channels used by docker engine to process remote requests :

- `--tls` : enforces the use of TLS for all remote communications
- `--tlscacert="~/.docker/ca.pem"` : the daemon will trust certs singled only by the above CA
- `--tlscert="~/.docker/cert.pem”` : provides path to the TLS certificate file.
- `--tlskey="~/.docker/key.pem”` : provides a path to TLS key file
- `--tlsverify` : enforces the use of TLS and verifies the remote host

network

Flags in this category define options that affect how the docker engine interacts with host networking layer and how internal components leverage the network.

network

global engine network options, that define the general networking parameters used by containers created on the host :

- `-b,--bridge string` : sets the default bridge to attach containers to.
- `--bip string` : specifies the network bridge IP.
- `--ipv6` : enables IPv6 networking for containers.
- `--mtu int` : sets the general MTU the container network uses to transmit packets.
- `--default-gateway ip` : defines the container default gateway IPv4 address.
- `--default-gateway-v6 ip` : defines the container default gateway IPv6 address.

network

global engine network options, that define the general networking parameters used by containers created on the host :

- `--dns-opt list` : sets the DNS options to use as part of a container's resolv.conf file or within the embedded dns server
- `--dns-search list` : used by the docker engine to specify the DNS search domains to use
- `--fixed-cidr string` : defines the IPv4 subnet for fixed IPs
- `--fixed-cidr-v6 string` : defines the IPv6 subnet for fixed IPs

network

global engine network options, that define the way in which the docker engine uses the system iptables for managing container communications :

- `--icc=true` : enables inter-container communication (default true)
- `--ip-forward=true` : allows the use of net.ipv4.ip_forward (default true)
- `--ip-masq=true` : allows the use of IP masquerading (default true)
- `--iptables=true` : allows the engine to make addition of iptables rules (default true)
- `--userland-proxy=true` : allows the engine to make use of the userland proxy for loopback traffic

network

global engine network options, that define the way in which the docker engine works with overlay style networks :

- `--cluster-store string` : defines the URL of the distributed storage backend.
- `--cluster-advertise string` : defines the address of the daemon instance on the cluster
- `--cluster-store-opt map` : defines the set of cluster options to use (default map [])

storage

flags in this category define options that effect how the docker engine interacts with host storage system while in daemon mode

storage

Global engine storage options, that define how the docker engine interacts with the host storage system while in daemon mode :

- `-g, --graph="/var/lib/docker”` : sets the directory where the Docker runtime places files associated with containers
- `-s, --storage-driver string` : defines the storage driver the engine uses for running containers it manages
- `--storage-opt list` : sets any storage driver options that may help optimize the use of the storage platform for desired use case (default [])

logging

flags in this category define options that effect how the docker engine interacts with host logging system while in daemon mode :

- `-l, --log-level string` : sets the logging level used by the engine
- `--log-driver string` : sets the default driver to use for container logs
  - json/syslog are primary supported versions (default : ”json-file”)
- `--log-opt map` : sets default log driver options to use with the log driver (default map [])
- `--raw-logs` : sets logs to use full timestamps without ANSI coloring

management

flags in this category define options that effect how the docker engine manages containers while in daemon mode

- `--cgroup-parent string` : sets the parent cgroup for all containers
- `--default-ulimit ulimit` : sets the default ulimit settings for containers
- `--exec-opt list` : sets the runtime execution options (default [])
- `--exec-root="/var/run/docker”` : sets the default path for execution state files/socket files
- `--config-file=“/etc/docker/daemon.json”` : indicates a daemon configuration file to use in setting up engine options
- `--containerd string` : sets the path to place the containerd socket
- `-p, --pidfile string` : sets the path to use for daemon PID file (default “/var/run/docker.pid”)
- `--registry-mirror list` : defines the preferred Docker registry mirror to use (default [])

security

flags in this category define options that effect how the docker engine interacts with various system security options :

- `--api-cors-header string` : sets the CORS headers in the Engine API
- `--selinux-enabled` : enables or disables selinux support
  - Example: On a well-configured RedHat system with SELinux enabled, you will not be able to mount a volume that is outside /var/lib/docker into a container, as that violates SELinux policy.
- `--userns-remap string` : allows the engine to use user namespace remapping
- `--authorization-plugin list` : defines which plugin to use for granular engine access (default [])

Section 3.3 | best practices

(from Nebulaworks - Docker doesn't have official best practices)

nebulaworks

common options

although there are many different environments that docker is deployed into, there are some common options that are enabled to use docker hosts in shared environments:

- `-H, --host=[ ]`
- `--log-driver`
- `--log-opt=[]`
- `--dns=[]`
- `--label=[]`
  - These require a daemon restart, so best to choose labels in advance.

environmental considerations

with the rise of the hybrid cloud it becomes very important to ensure the communications are as secure as possible. At a minimum this means all channels that the engine uses should be encrypted.

```
--insecure-registry=[]
```

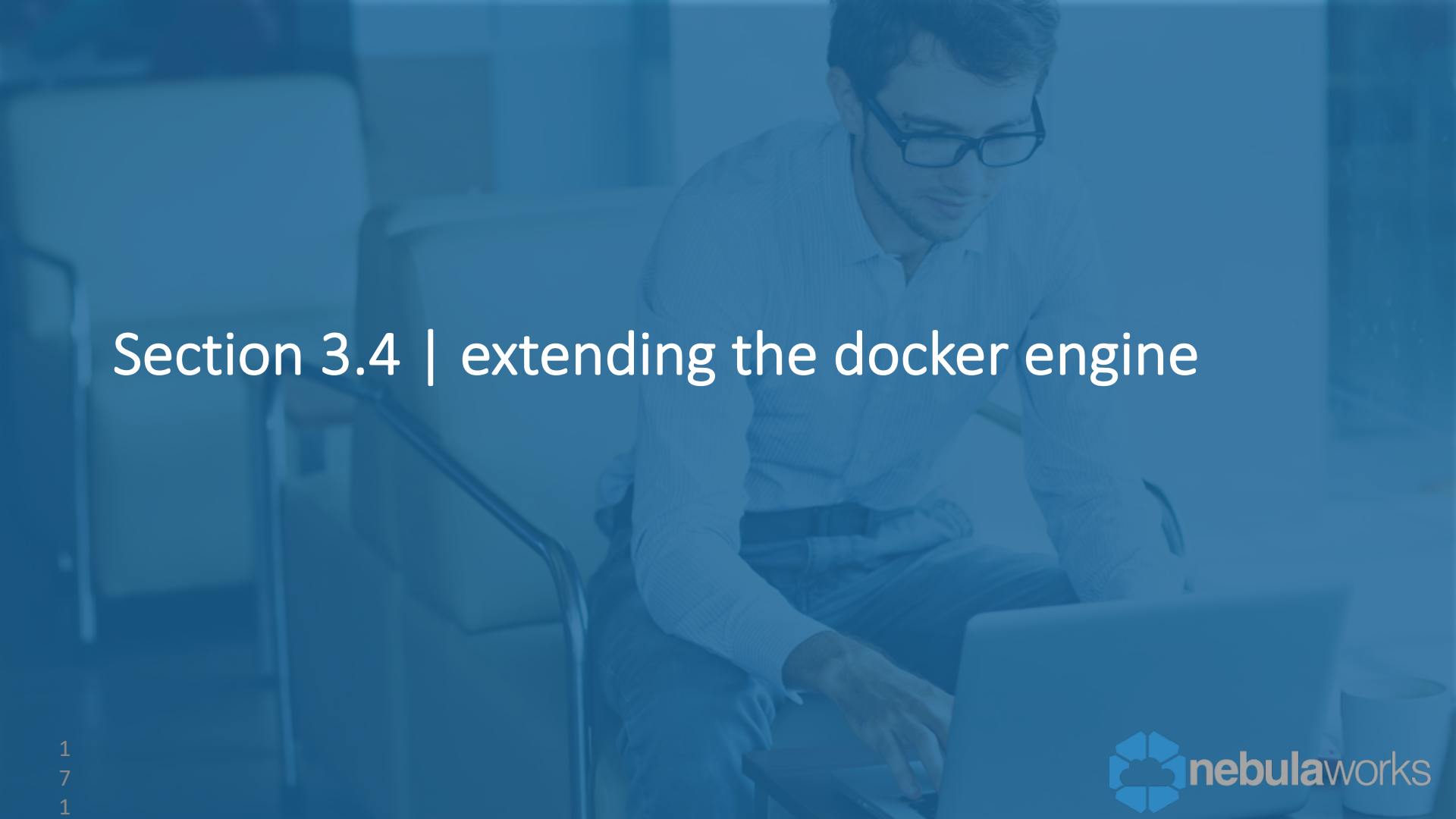
- for using self-signed certs, such as when you are using a bunch of internal resources, and don't want to pay for signed certs

```
--tlsverify
--tlscacert="~/.docker/ca.pem
--tlscert="~/.docker/cert.pem
--tlskey="~/.docker/key.pem”
```

performance considerations

the idea of a container being a better kind of virtualization often times is derived from a performance perspective. without the need to emulate all aspects of hardware the performance impact is minimal within the cpu complex but this can be less true for other parts of the stack.

```
-s, --storage-driver=""
--storage-opt=[]
-g, --graph="/var/lib/docker"
```



Section 3.4 | extending the docker engine

plugin architecture : the docker plugins are out-of-process extensions which add capabilities to the docker Engine

- “batteries included but swappable” philosophy
- three types available
  - networking
  - volume
  - access authorization
- networking and storage decoupled at the end of 2015 to allow more granular independent management of those resources

networking

designed to allow the docker engine to be extended to support a wide range of networking technologies, such as VXLAN, IPVLAN, MACVLAN or something completely different

key facts >>>

- supported via the libNetwork* project
- plugins are implemented as a “remote drivers” for libnetwork
- are used via the standard methods on the docker run command and top level network object

volume

designed to allow the docker engine to be integrated to support a wide range of external storage systems, such as Amazon EBS, and enable data volumes to persist beyond the lifetime of a single docker host.

- plugins register with the engine as volume drivers
- default is local which use the default volume driver to manage data on local system volumes
- volumes used in this fashion must be part of the host filesystem and must be writable to the docker engine

access authorization

- All or Nothing
  - How do we control access to the docker daemon (CLI and API)?
- Authorization plugins are used when you want more granular access policies for communication with the daemon.
- broken into two components: AuthZ and AuthN
  - AuthZ – deals with authorization functions
  - AuthN – deals with authentication functions
- allows for granular control over the use of docker api functions

Works Cited

Figure 3.1 : Poulton, Nigel (2008). *Docker Engine 3rd Party Pluggables* [Digital image]. Nigel Poulton.

End of day 1