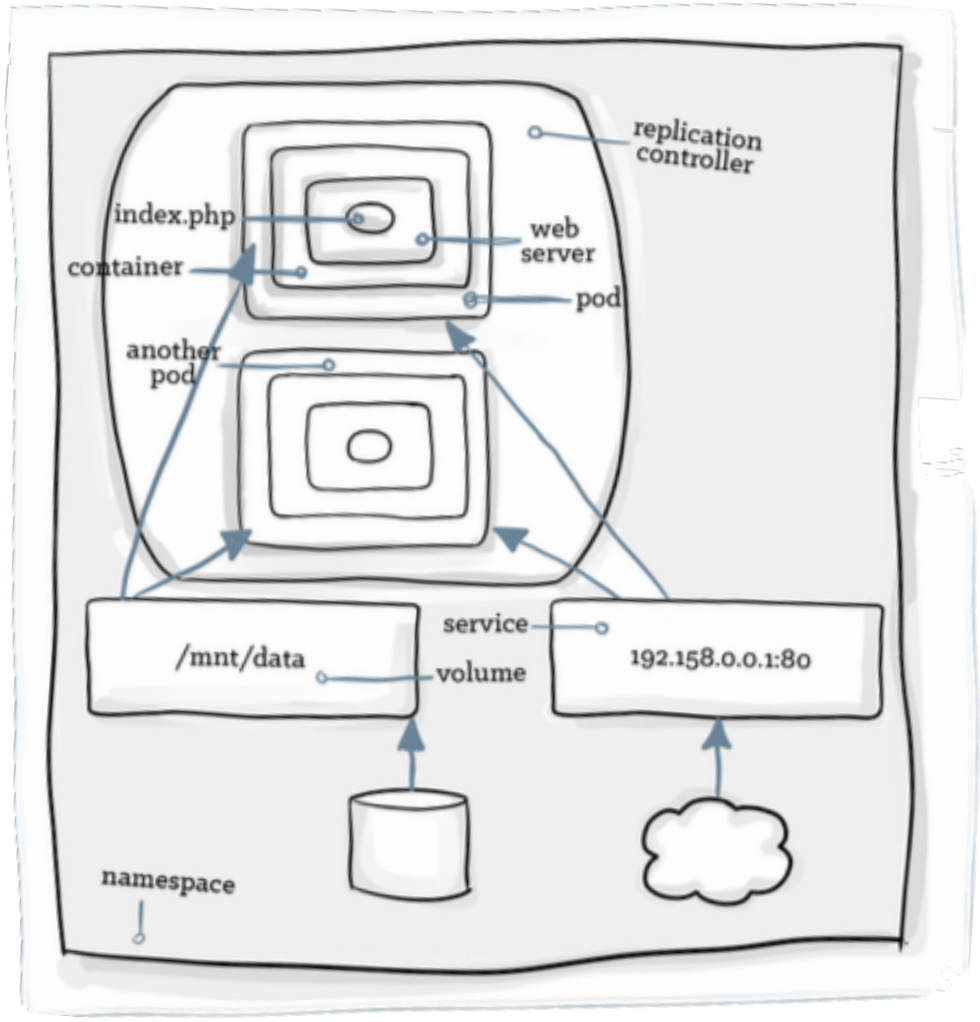
Docker

Lab Guide v8.0

Binny Raphael
Cloud and Infrastructure Services



Class Introductions

- Name
- Base Location
- Project
- Your total experience
- Your experience with Linux/Unix
- Your experience with Docker/Kubernetes/OpenShift if any.
- Why are you attending this session?
- What is your expectation out of this session?

Notes:

- As a courtesy to others, please be on time before the start of the session each-day and after every break. You are free to step outside if you need a break; no need to ask permission.
- Please do not do your project work, email, SMS, WhatsApp, etc during the sessions or talk on phone.
- Please keep you phone on silent mode; in case if you need to take the call, you may step out.
- As informed earlier, you need to be comfortable with the UNIX command line.
- This entire session if aimed towards Administration and Operations team; this is not a Developer Oriented course. (However, there is no harm for a developer in attending this course)
- Feel free to interrupt and ask questions at any time. Sometimes, we may have to take the question offline in the interest of time or wait for a particular topic to be covered first before it can be answered.
- This is a 5-day intense training and you will feel exhausted; we still need you to keep the tempo up all the time.
- The whole session is hands-on; you need to keep up with the average flow of the class. Please DO NOT do the labs during the theory sessions or skip the flow of the lab sessions. We have seen issues earlier.
- We have included as much information as possible in this session. However, *you will need to a lot of reading for many of the topics discussed in class either from internet or vendor sites*. You must read & practice a lot.
- All hands-on sections are identified by a gray rectangle box. Once a while, you may need to follow specific instruction when running those commands in some cases. Please follow the instruction.
- Last not the least, *it will take more than merely attending these sessions to make you an expert.*

Credits: Some of the slides and diagrams are from Docker, Kubernetes and Red Hat OpenShift tutorials on the internet and is reused in this presentation. Such slide have a Docker or Kubernetes or OpenShift logo at the bottom left. Thanks to all of them.

Accessing your Lab

Accessing Docker Lab

Step 1: Use MS Remote Desktop and connect to the Remote Desktop:

MYSIMSTTS01(10.122.198.27) or MYSIMSTTS02(10.122.198.28)

[user-ID: **INFOSYS_USERID** Password: **your AD Password**] -

Step 3: Set the Chrome Proxy Server to :: **172.20.0.xxx** and Port :: **8080**

Step 2: From the remote desktop, Launch Putty and SSH to you Server Instance

Server: **172.20.0.xxx**

User Name: **session7**

Password: **Infy123+**

Step 4: From the Pytty session, again ssh to you Virtual Machine instance

VM/Pod Server: **docker.podNN.example.com**

User Name: **root**

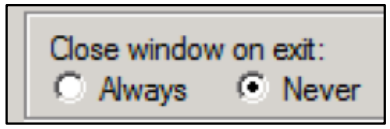
Password: **root123**

Make sure that you use the same POD-NUMBER assigned to you throughout this session.

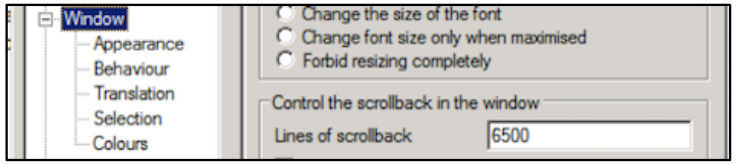
Recommended Putty Configuration

It is recommended to make the following Putty configuration so as to make things easier for you during the labs:

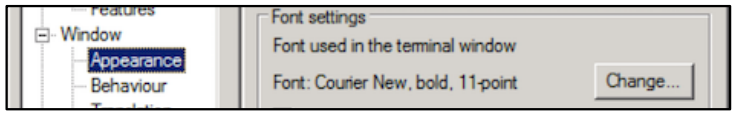
Disable Close window on exit:



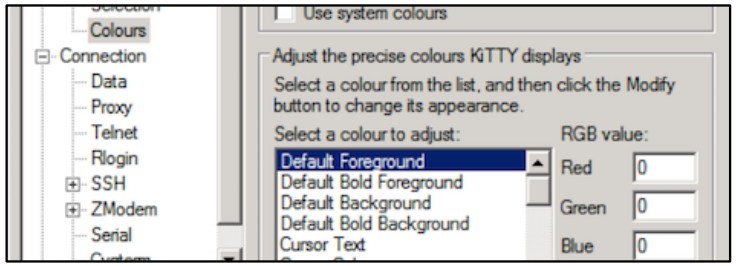
Increase the Scroll back:

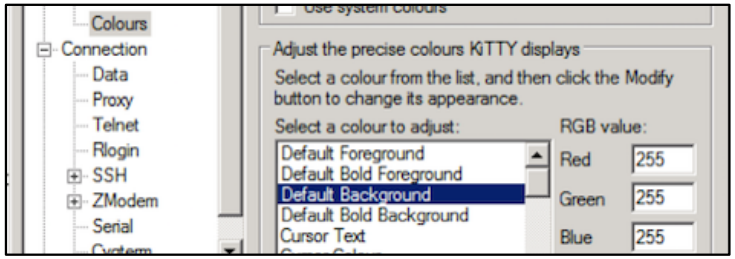


Change the font:



Adjust the Foreground and Background colors:





Note: Make sure that you save these setting; else you will end up doing this changes every time.

Basic Concepts

Linux Containers

- LXC is an operating system-level virtualization method for running multiple isolated Linux systems (containers) on a single control host
- LXC - LinuX Containers
- Lightweight virtualization
- Containerization of OS less the Kernel
- Realized using features provided by a modern Linux kernel
- VMs without the hypervisor (kind of)
- Single or Multiple application
- From the inside it looks like a VM
- Container is filesystem neutral - persistent data can be saved inside or outside the container
- From the outside it looks like a normal process
- Provides lightweight virtualization

Why Linux Containers

- Provision in seconds / milliseconds
- Near bare metal runtime performance
- VM-like agility - it’s still “virtualization” Flexibility
- Containerize a “system”
- Containerize “application(s)”
- Extremely Lightweight
- Some of Kernel Features used by LXC include namespaces and cgroups

Linux has Containers built in - No need of Docker

Linux containers are just isolated groups of processes running on a single host. The isolation leverages several underlying technologies built into the Linux kernel: **`chroots, namespaces`**, **`cgroups`**, and lots of terms you've probably heard before.

Let us us these features and see is we can build a container without Docker.

First we need a OS Image (Container image). We will use one of the rootfs for the centos Image:

```
[root@docker ~]$ whoami
root

[root@docker ~]$ cat pod*TXT
podX

[root@docker ~]$ ls -l /var/tmp/Images/centos-7-docker.tar.xz
-rw-r--r--. 1 root root 42517348 Jun 15  2018 /var/tmp/Images/centos-7-docker.tar.xz

***** FOR KATAKODA PLAYGROUND USE THE BELOW COMMANDS BEFORE NEXT SLIDE *****

$ mkdir -p /var/tmp/Images
$ wget https://buildlogs.centos.org/centos/7/docker/CentOS-7-20140625-x86_64-docker_01.img.tar.xz -O /var/tmp/Images/centos-7-docker.tar.xz
```

Linux has Containers built in - No need of Docker

Extract the file into a new folder named **`rootfs`**:

```
[root@docker ~]$ mkdir rootfs

[root@docker ~]$ pwd
/home/podxuser

[root@docker ~]$ tar xf /var/tmp/Images/centos-7-docker.tar.xz -C rootfs/

[root@docker ~]$ ls rootfs/
anaconda-post.log  bin  dev  etc  home  lib  lib64  lost+found  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp
usr  var

[root@docker ~]$ echo "export PS1='\[\e[31m\][\u@CONTAINER|\[\e[7m\]$PWD]\[\e[0m\]# \[\e[0m\]'" >> rootfs/root/.bashrc
```

The **`rootfs`** directory looks lot like a regular Linux system. There's a bin directory with executables, an etc with system configuration, a lib with shared libraries, and so on.

Linux Native Containers - First tool **chroot**

The first native tool is the **chroot.** This is a thin wrapper around system call named **syscall**, it allows us to restrict a process' view of the file system. In this case, we'll restrict our process to the "**rootfs**" directory then exec a **bash** shell.

```
[root@docker ~]$ sudo chroot rootfs /bin/bash
[root@CONTAINER /]#

[root@CONTAINER /]# pwd
/

[root@CONTAINER /]# ls
anaconda-post.log  bin  dev  etc  home  lib  lib64  lost+found  media  mnt  opt  proc  root  run  sbin  srv
sys  tmp  usr  var

[root@CONTAINER /]# cat /etc/passwd
[root@CONTAINER /]# cat /etc/hosts

[root@CONTAINER /]# echo 'Hello from Container World!!!'
Hello from Container World!!!
```

Linux Native Containers - Playing around with **chroot** environment

How safe or isolated is this **chroot** environment? Let us test it.

Start **another** Putty session and run '**top**' in it. Keep it running. Come back to the **chroot prompt:**

```
[[this is from the new Putty terminal on the host]]
[root@docker ~]$ top

[[this is from the chroot prompt]]
[root@CONTAINER /]# mount -t proc proc /proc

[root@CONTAINER /]# ps aux | grep top
root      1423  0.1  0.2 161944  2196 ?        S+   08:33   0:00 top
root      1429  0.0  0.0  12460   968 ?        R+   08:33   0:00 grep --color=auto top

[root@CONTAINER /]# pkill top
```

What happened to the top command? The chrooted shell is running as root, so it has no problem in seeing and killing the top process. This is not what we expected of the containerized process. What do we do?

The answer to this is **namespaces**.

Linux Native Containers - **Namespaces**

Namespaces allow us to create restricted views of systems like the process tree, network interfaces, and mounts. We can use the '**`unshare`**' command. The **`unshare`** command is again a wrapper around **`syscall`** and lets us setup namespace manually.

Exit out of the previous **`chroot`** shell using **`exit`** command and **then start again**

We'll now create a **`PID`** namespace for the shell, then execute the **`chroot`** like the last lab.

```
[root@docker ~]$ sudo unshare -p -f --mount-proc=$PWD/rootfs/proc chroot rootfs /bin/bash

this is from the new chroot prompt
[root@CONTAINER /]# ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.1  15200  2004 ?        S    08:47   0:00 /bin/bash
root         10  0.0  0.1  50868  1812 ?        R+   08:47   0:00 ps aux
```

What happened? What do you see? Check the PIDs. We see a PID '1' and we can't see the host's process tree anymore. Start a 'top' on the host and see if we can see it like the last lab. We have now isolated the processes using namespace.

Linux Native Containers - Entering the namespace with **`nsenter`**

One of the powerful property of namespaces is their **composability**.

Composability is a system design principle that deals with the inter-relationships of components. A highly composable system provides components that can be selected and assembled in various combinations to satisfy specific user requirements.

In this case, processes may choose to separate some namespaces but share others. For instance it may be useful for two programs to have isolated PID namespaces, but share a network namespace (like in a POD with Kubernetes - we will see this later).

We can enter a namespace using the **`nsenter`** command line tool. This tool inturn uses **`setns`** syscall.

Linux Native Containers - Entering the namespace with **nsenter**

Keep the current chroot running. Now from the host putty session as **podxuser**, find the shell running the **bash** from the last example. Note the PID.

```
[root@docker ~]$ ps aux | grep /bin/bash | grep root | grep -v unshare
root      1440  0.0  0.1  15200  2004 pts/0    S+   14:17   0:00 /bin/bash
```

The kernel exposes namespaces under **/proc/(PID)/ns** as files. In this case, **/proc/1440/ns/pid** is the process namespace we will be joining using **nsenter.**

```
[root@docker ~]$ export MYPID=1440

[root@docker ~]$ sudo ls -l /proc/$MYPID/ns
total 0
lrwxrwxrwx. 1 root root 0 Jun 16 14:38 ipc -> ipc:[4026531839]
lrwxrwxrwx. 1 root root 0 Jun 16 14:38 mnt -> mnt:[4026532119]
lrwxrwxrwx. 1 root root 0 Jun 16 14:38 net -> net:[4026531956]
lrwxrwxrwx. 1 root root 0 Jun 16 14:38 pid -> pid:[4026532129]
lrwxrwxrwx. 1 root root 0 Jun 16 14:38 user -> user:[4026531837]
lrwxrwxrwx. 1 root root 0 Jun 16 14:38 uts -> uts:[4026531838]
```

Linux Native Containers - Entering the namespace with **nsenter**

As discussed earlier, **nsenter** command provides a wrapper around **setns** to enter a namespace. The command is almost the same; we'll provide the namespace file, then run the **unshare** to remount **/proc** and **chroot** to setup a chroot. This time, instead of creating a new namespace, our shell will join the existing one.

```
[root@docker ~]$ sudo nsenter --pid=/proc/$MYPID/ns/pid unshare -f --mount-proc=$PWD/rootfs/proc chroot rootfs /bin/bash

[root@CONTAINER /]# ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.1  15200  2004 ?        S+   08:47   0:00 /bin/bash
root         11  0.0  0.0 107936   592 ?        S    09:14   0:00 unshare -f --mount-proc=/home/podxuser/rootfs/proc
chroot rootfs /bin/bash
root         12  0.0  0.1  15200  2004 ?        S    09:14   0:00 /bin/bash
root         21  0.0  0.1  50868  1816 ?        R+   09:14   0:00 ps aux

[root@CONTAINER|/root]# touch i.was.here

[root@CONTAINER /]# echo $$
12
```

when we run **ps** in the second shell **(PID 12)** we see the first shell **(PID 1**). Check if you can see the file you created using "**touch i.was.here**" above from the first shell

Linux Native Containers - Namespace

Check from the first shell

```
[root@CONTAINER|/root]# ls -l /i.was.here

-rw-r--r--. 1 root root 0 Jul  4 15:09 /i.was.here
```

We have just explored a very small portion of Namespace Features and Namespace capabilities.

You are requested to check this important feature of Linux Kernel and have a very good understanding of this. There are lot of hands on demos and tutorials available on the internet.

Note: Make sure that you exit out of all chroot shells.

Linux Native Containers - Controlling resources with **cgroup**

Another important feature provided by the Linux kernel is **cgroups**, short for control groups. Control groups allow the kernel impose isolation on resources like process, memory, CPU, disk, network, etc. This is required to prevent system problems caused by other rouge processes like hogging RAM or compute resources.

Similar to namespace, kernel exposes cgroups through the **/sys/fs/cgroup** directory.

```
[root@rhel75 ~]$ ls /sys/fs/cgroup/

blkio  cpu  cpuacct  cpu,cpuacct  cpuset  devices  freezer  hugetlb  memory  net_cls  net_cls,net_prio
net_prio  perf_event  pids  system
```

We will quickly explore how we can restrict CPU and Memory of a process.

Linux Native Containers - Controlling CPU with **cgroup**

Creating a cgroup manually is extremely easy, just create a directory. In this case we'll create a CPU group called "podxuser". Once created, the kernel fills the directory with files that can be used to configure the cgroup.

```
[root@docker ~]$ export PODNUM=poduser
[root@docker ~]$ mkdir /sys/fs/cgroup/cpu/$PODNUM

[root@docker ~]$ ls /sys/fs/cgroup/cpu/$PODNUM/
cgroup.clone_children  cgroup.procs  cpuacct.usage         cpu.cfs_period_us  cpu.rt_period_us   cpu.shares
notify_on_release
cgroup.event_control   cpuacct.stat  cpuacct.usage_percpu  cpu.cfs_quota_us   cpu.rt_runtime_us  cpu.stat
tasks
```

Linux Native Containers - Controlling CPU with **cgroup**

To adjust a value we just have to write to the corresponding file. Let's limit the cgroup to use 2% of the CPU on the system using Completely Fair Scheduler (CFS).

```
cat /sys/fs/cgroup/cpu/$PODNUM/cpu.cfs_period_us

echo 2000 > /sys/fs/cgroup/cpu/$PODNUM/cpu.cfs_quota_us
```

The tasks file is special, it contains the list of processes which are assigned to the cgroup. To join the cgroup we can write our own PID as below:

```
[root@rhel75]# echo $$ > /sys/fs/cgroup/cpu/$PODNUM/tasks
```

Now let us test it quickly.

Note: Please read the Linux Manual for cgroup and you can can see how other parameters can be used and how to calculate all these values. **cpu.cfs_quota_us** specifies the total amount of time in microseconds (µs, represented here as "us") for which all tasks in a cgroup can run during one period (as defined by **cpu.cfs_period_us**).

Linux Native Containers - Controlling CPU with **cgroup**

Run the one-liner python program which will try to throttle the CPU and see what happens.

```
[root@rhel75]# python -c "while True: 42*42"
```

From another Putty session, check the CPU usage using 'top' command or use "`top -b | grep python`" :

```
  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 9243 root      20   0  123344   4428   1920 R  2.0  0.4   0:01.51 python
```

Without this limit, it would have overloaded the CPU.

DO NOT run the above one-liner python without the limit OR change the 2%; since this is a shared system it will cause issues to all the other lab users.

Linux Native Containers - Controlling CPU with **cgroup**

If the setup is correctly done, this program won't crash your computer. The kernel will take care to allocate 2% of the CPU to that cgroup.

Finally, once you have exited all the shell, you can remove the cgroup restriction using rmdir (do not use rm -rf)

```
[root@rhel75]# rmdir /sys/fs/cgroup/cpu/$PODNUM
```

Please exit from this putty/shell session and login again else you will face issues since the PID is under CPU quota.

```
[root@rhel75]# exit
```

Linux Native Containers - Controlling memory with **cgroup**

Once again, creating a memory cgroup manually is again easy, just create a directory. In this case we'll create a memory group called "podxuser". Once created, the kernel fills the directory with files that can be used to configure the cgroup. [Use a new Putty Session]

```
[root@rhel75]# export PODNUM=poduser
[root@rhel75]# mkdir /sys/fs/cgroup/memory/$PODNUM

[root@rhel75]# ls /sys/fs/cgroup/memory/$PODNUM/
cgroup.clone_children           memory.kmem.slabinfo               memory.memsw.failcnt
memory.soft_limit_in_bytes
cgroup.event_control            memory.kmem.tcp.failcnt            memory.memsw.limit_in_bytes      memory.stat
cgroup.procs                    memory.kmem.tcp.limit_in_bytes     memory.memsw.max_usage_in_bytes  memory.swappiness
memory.failcnt                  memory.kmem.tcp.max_usage_in_bytes memory.memsw.usage_in_bytes
memory.usage_in_bytes
memory.force_empty              memory.kmem.tcp.usage_in_bytes     memory.move_charge_at_immigrate
memory.use_hierarchy
memory.kmem.failcnt             memory.kmem.usage_in_bytes         memory.numa_stat                 notify_on_release
memory.kmem.limit_in_bytes      memory.limit_in_bytes              memory.oom_control               tasks
memory.kmem.max_usage_in_bytes  memory.max_usage_in_bytes          memory.pressure_level
```

Linux Native Containers - Controlling memory with **cgroup**

To adjust a value we just have to write to the corresponding file. Let's limit the cgroup to 50MB of memory and turn swap off.

```
[root@rhel75]# echo "50000000" > /sys/fs/cgroup/memory/$PODNUM/memory.limit_in_bytes

[root@rhel75]# echo "0" > /sys/fs/cgroup/memory/$PODNUM/memory.swappiness
```

The tasks file is special, it contains the list of processes which are assigned to the cgroup. To join the cgroup we can write our own PID as below:

```
[root@rhel75]# echo $$ > /sys/fs/cgroup/memory/$PODNUM/tasks
```

Now let us test it quickly.

Linux Native Containers - Controlling memory with **cgroup**

Create a small python program which will keep eating memory; we will then run this program and see what happens.

```
[root@rhel75]# cat eatmem.py
f = open("/dev/urandom", "r")
data = ""

i=0
while True:
    data += f.read(10000000) # 10mb
    i += 1
    print "%dmb" % (i*10,)


[root@rhel75]# python eatmem.py
10mb
20mb
30mb
Killed
```

Linux Native Containers - Controlling memory with **cgroup**

If the setup is correctly done, this program won't crash your computer. The kernel will take care of it and kill this rogue program. You can see the details in /var/log/messages

```
[root@docker|~]# dmesg  | tail
Jun 16 15:36:38 rhel75 kernel: Task in /poduser killed as a result of limit of /poduser
Jun 16 15:36:38 rhel75 kernel: memory: usage 48828kB, limit 48828kB, failcnt 34
Jun 16 15:36:38 rhel75 kernel: memory+swap: usage 48828kB, limit 9007199254740988kB, failcnt 0
Jun 16 15:36:38 rhel75 kernel: kmem: usage 0kB, limit 9007199254740988kB, failcnt 0
Jun 16 15:36:38 rhel75 kernel: Memory cgroup stats for /podxuser: cache:4KB rss:48824KB rss_huge:28672KB mapped_file:0KB swap:0KB
inactive_anon:31712KB active_anon:17112KB inactive_file:4KB active_file:0KB unevictable:0KB
Jun 16 15:36:38 rhel75 kernel: [ pid ]   uid  tgid total_vm      rss nr_ptes swapents oom_score_adj name
Jun 16 15:36:38 rhel75 kernel: [ 9107]     0  9107    28859      532      14        0             0 bash
Jun 16 15:36:38 rhel75 kernel: [ 9127]     0  9127    43048    12645      41        0             0 python
Jun 16 15:36:38 rhel75 kernel: Memory cgroup out of memory: Kill process 9127 (python) score 1008 or sacrifice child
Jun 16 15:36:38 rhel75 kernel: Killed process 9127 (python) total-vm:172192kB, anon-rss:48580kB, file-rss:2000kB, shmem-rss:0kB
```

Finally, once you have exited all the shell, you can remove the cgroup restriction using rmdir (do not use rm -rf)

```
[root@docker|~]# rmdir /sys/fs/cgroup/memory/poduser

[root@docker|~]# exit
```

Linux has Containers built in - Summary

Linux containers always existed. It can make use of Namespaces, cgroups, SELinux etc to build containers. Do you now need to build wrappers around these tools to roll out your own Container Management framework. Don’t bother; there are far too many. Projects like Docker and rkt are similar frameworks. We will look at Docker in sometime.

What you need to realize is that having a better understanding of these lower level technologies will help you work with these higher level tools; especially when debugging.

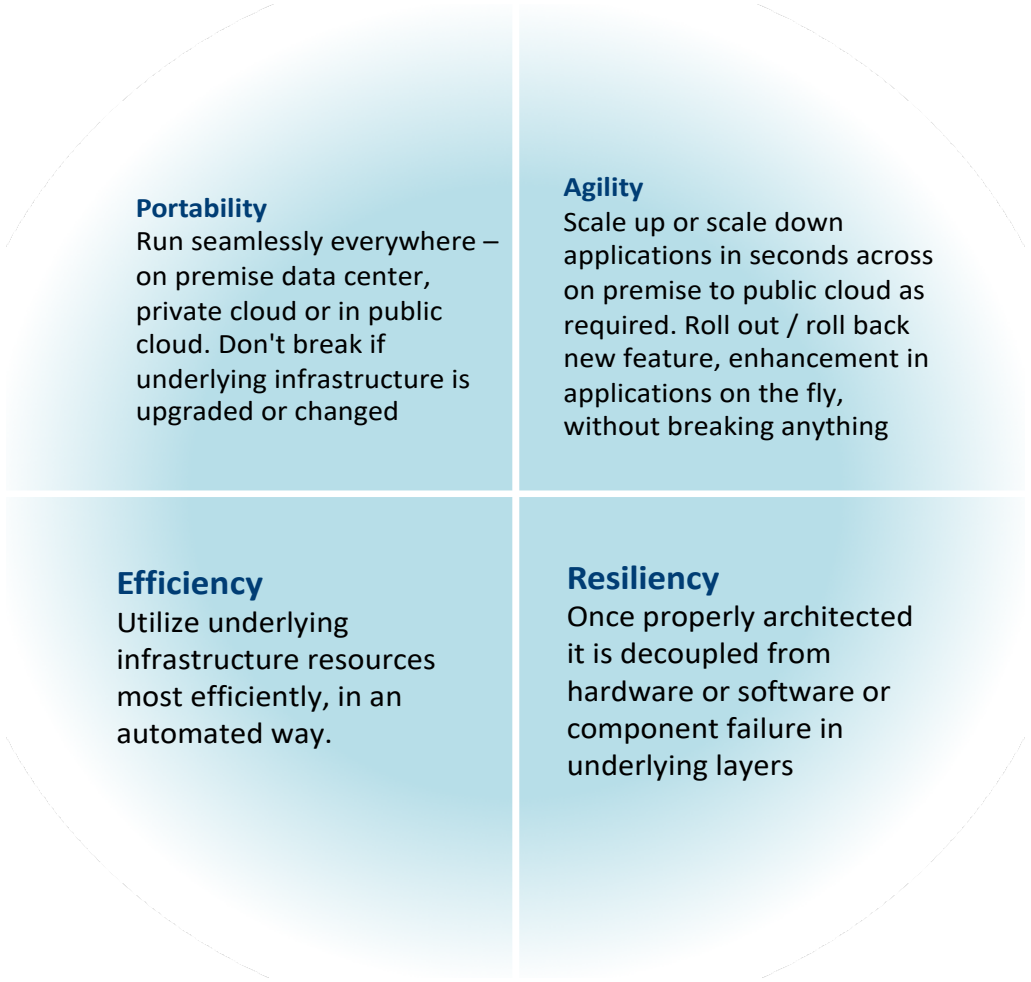
There are lot of other Kernel level features that are not covered and you should read and get your hands dirty.

Now, lets continue….

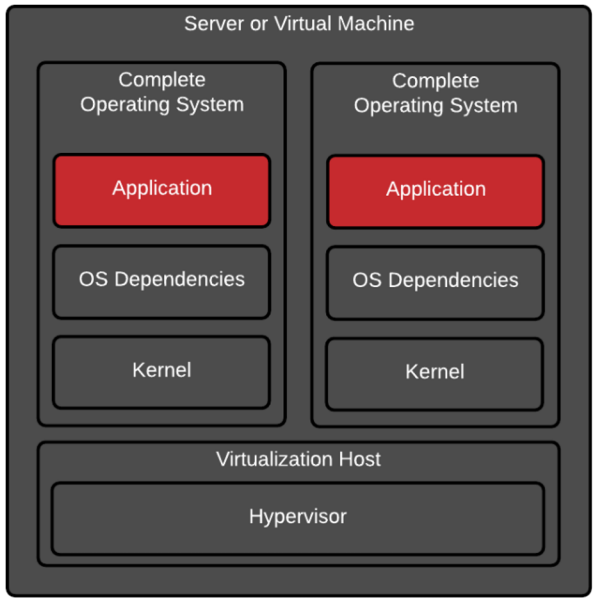
Containers and Docker Timeline

- 1979: Unix V7 in 1979, the chroot system call was introduced
- 2000: FreeBSD Jails
- 2001: Linux VServer
- 2004: Solaris Containers
- 2005: Open VZ (Open Virtuzzo)
- 2006: Process Containers by Google
- 2008: LXC (LinuX Containers) was the first, most complete implementation of Linux container manager. It was implemented in this year using cgroups and Linux namespaces, and it works on a single Linux kernel without requiring any patches.
- 2011: Warden from CloudFoundry
- 2013: LMCTFY Let Me Contain That For You kicked off as an open-source version of Google's container stack
- 2013: Docker - When Docker emerged in 2013, containers exploded in popularity; the growth of docker and people assuming containers mean docker and docker is what made container goes hand-in-hand.
- 2016: Lot of focus on Container Security
- 2017: Container Tools Become Mature
- 2017: Adoption of rkt and Containerd by CNCF (Docker’s donated Containerd project to the CNCF)
- 2017: Kubernetes Grows Up

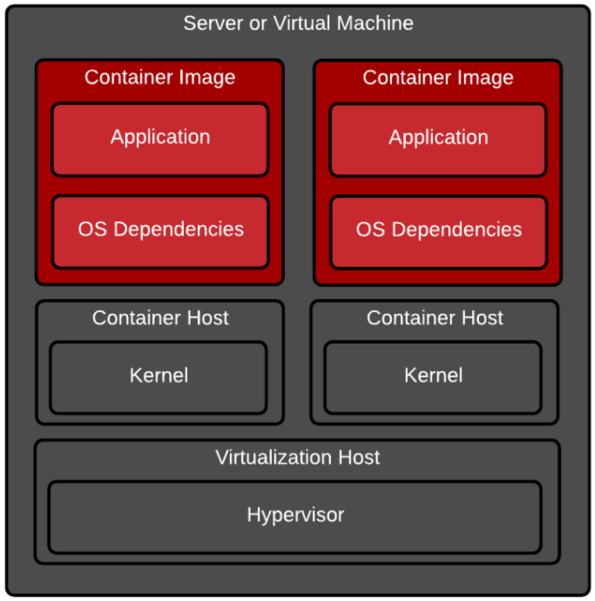
Key business drivers



Virtualized vs. Containerized Architecture



Application & Infrastructure
Updates Tightly Coupled



Application & Infrastructure
Updates Loosely Coupled

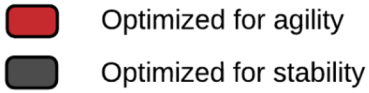


***Image Credits**: www.redhat.com*

Application Deployment Models

- Different deployment models exist:
  - Model 1: Physical Server
  - Application(s) runs natively on a physical host

- Model 2: Virtualized Server
  - Application(s) runs within a virtual machine

- Model 3: Containerized Server
  - Application(s) run within operating system containers

Traditional Application Deployment problems..

- Have you tried deploying multiple instances of an application or service as simple as Apache on same server?
- Now, what about different versions?
- What are the current solutions?
- What have you faced in you projects?
- What about updates?
- What about upgrade?
- What happens when one application dependency beaks another application/server?
- What about rollback?

Docker

What is Docker?

“Docker is an open platform for developers and sysadmins to build, ship, and run distributed applications. Consisting of Docker Engine, a portable, lightweight runtime and packaging tool, and Docker Hub, a cloud service for sharing applications and automating workflows, Docker enables apps to be quickly assembled from components and eliminates the friction between development, QA, and production environments. As a result, IT can ship faster and run the same app, unchanged, on laptops, data center VMs, and any cloud.”

source: https://www.docker.com/whatisdocker/

What is Docker?

- A container manager
- lightweight virtualization
- based on Linux namespaces and cgroups
- immutable architecture
- massively copy-on-write
- immutable images - instance is ephemeral, persistent data is stored outside the container, on the host or volume-containers
- instant deployment
- suitable for micro-services (one process, one container) - is designed to support a single application

What is Docker?

- a build system
  - images may be build from sources
  - using a simple DSL (Dockerfile)

- a set of REST APIs
  - Engine API (control the docker engine)
  - Plugin API (extend the engine → network, storage, authorisation)
  - Registry API (publish/download images)

And more features…

- Combination of technologies like cgroups + namespaces + image format + lifecycle management + application virtualization
- Git like semantics around image building like pull, commit, push ...
- Application virtualization
- Native Speeds
- Very high Density of Applications than virtual machines

The 'Holy Grail' of Clustering

- Provide maximum application density per- machine (currently in the hundreds) than virtualization
- Improved 'application namespace' isolation across your cluster
- No longer need to ensure stacks / sub-stacks are rolled onto your cluster
- Enables multiple versions across a cluster easily
- Combined with clustering, it can obviate traditional 'deployment problems'

Why Docker?

The software industry has changed

- Before:
  - monolithic applications
  - long development cycles
  - single environment
  - slowly scaling up

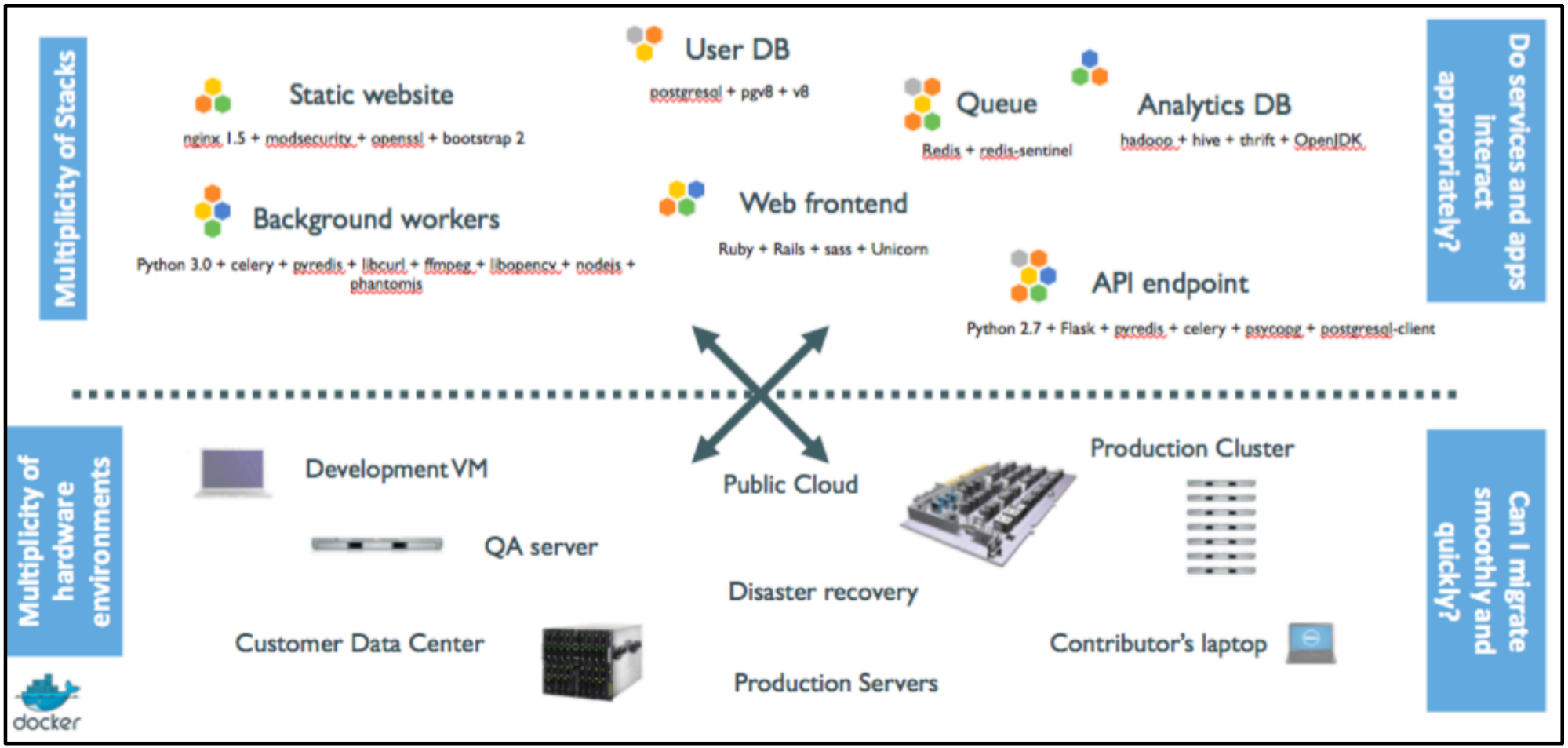
- Now:
  - decoupled services
  - fast, iterative improvements
  - multiple environments
  - quickly scaling out

Deployment becomes very complex

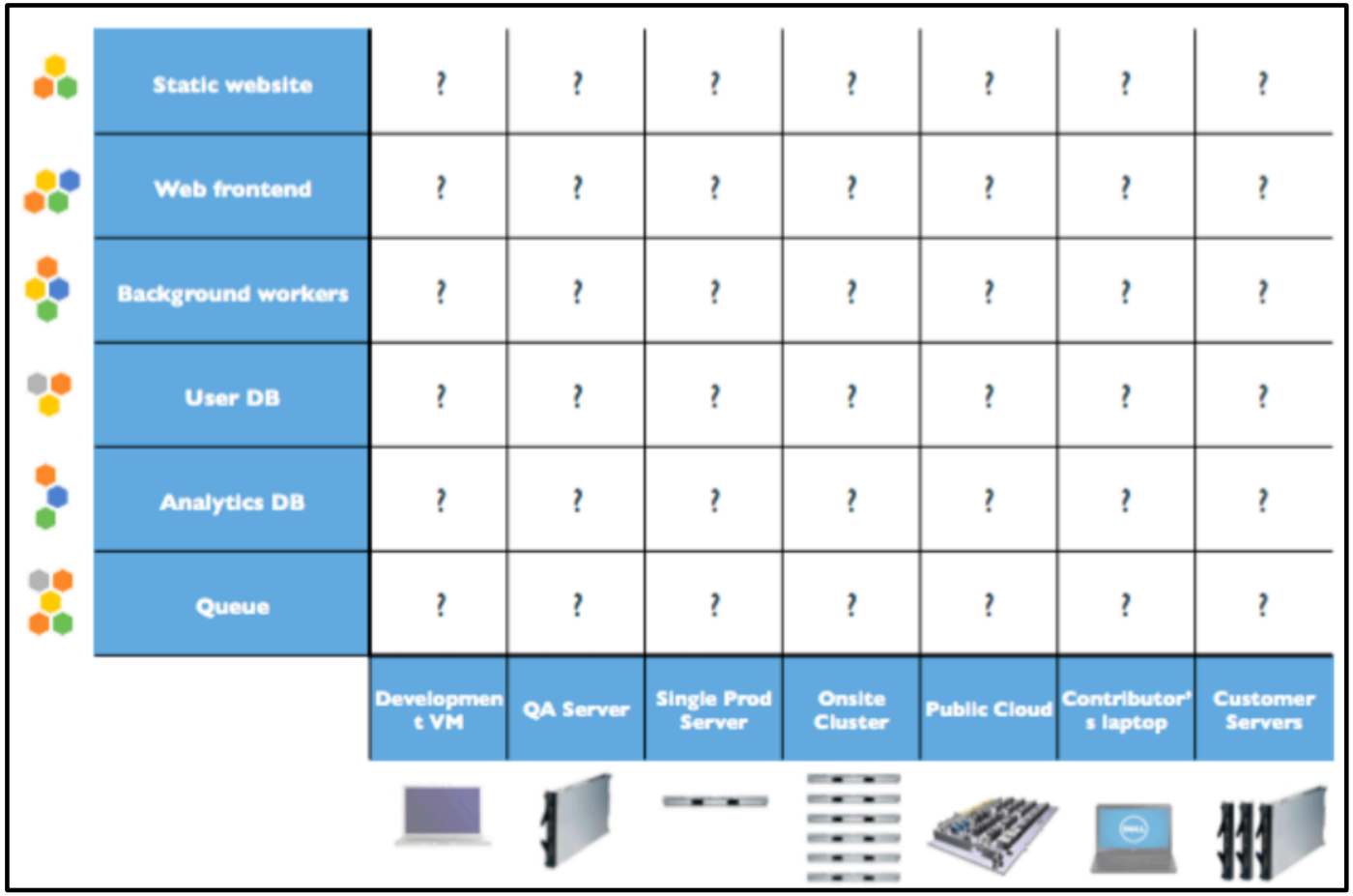
- Many different stacks:
  - languages
  - frameworks
  - databases

- Many different targets:
  - individual development environments
  - pre-production, QA, staging...
  - production: on premises, cloud, hybrid

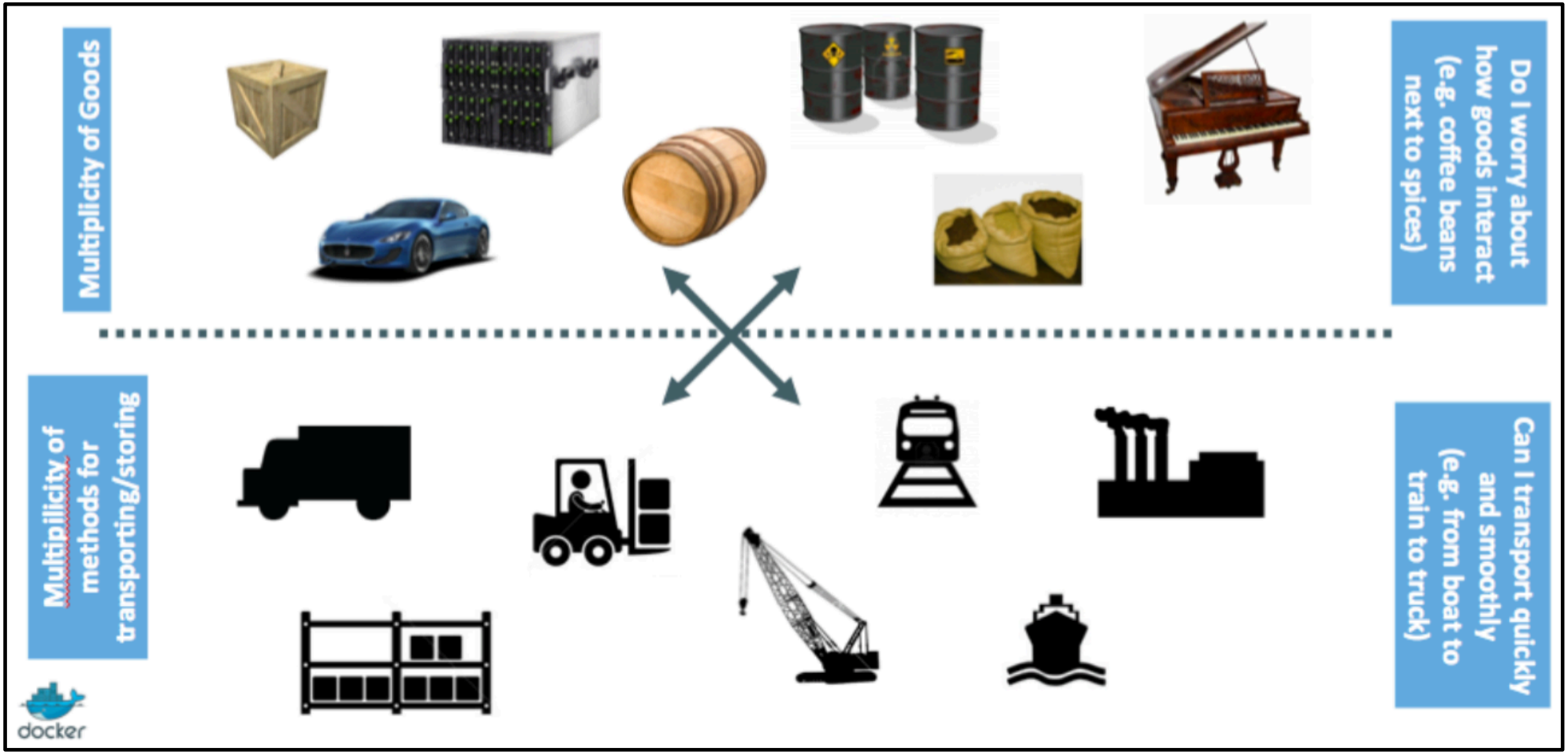
The Deployment Problem



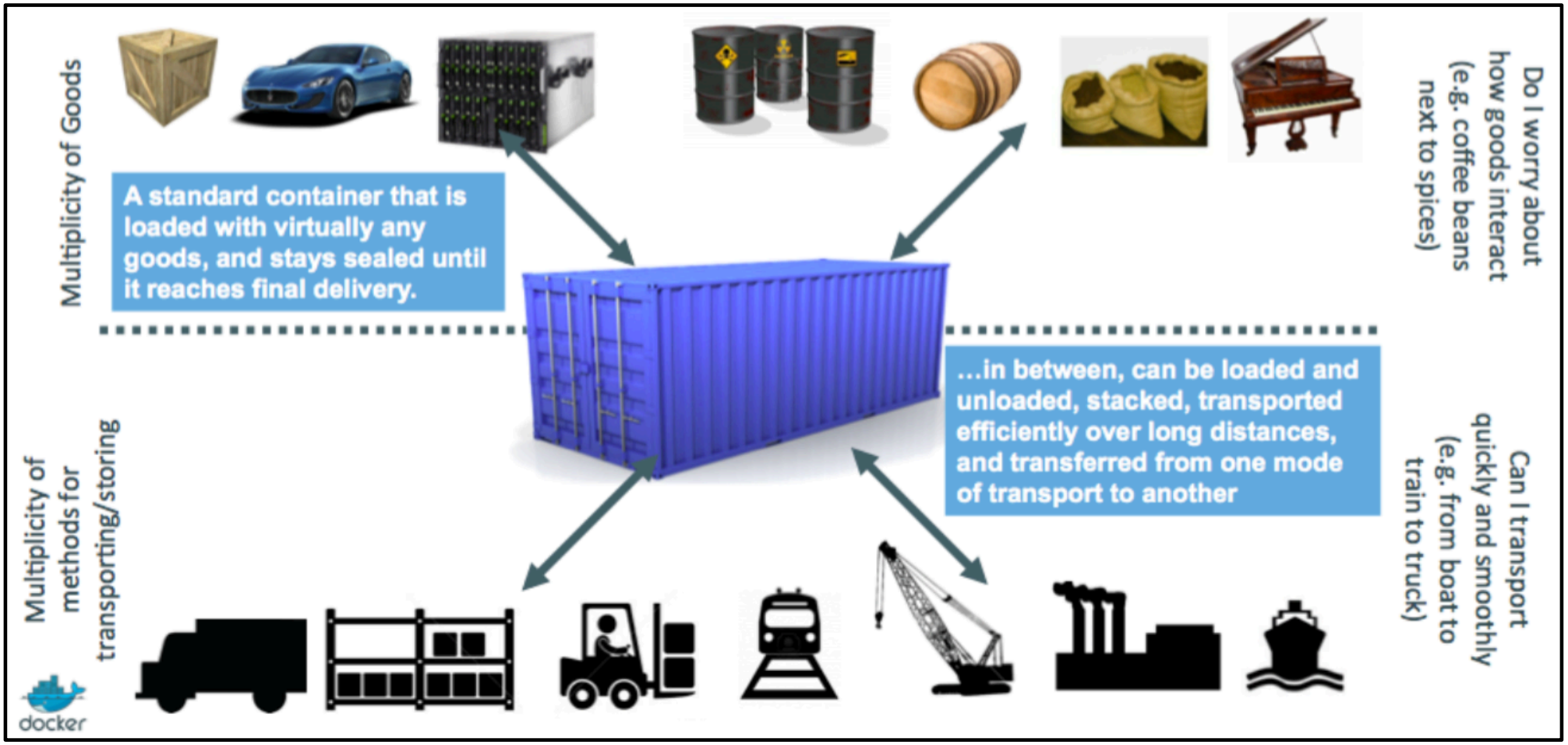
The matrix


Static website
Web frontend
Background workers
User DB
Analytics DB
Queue
?
Developmen t VM
QA Server
Single Prod Server
Onsite Cluster
Public Cloud
Contributor' s laptop
Customer Servers

The parallel with the shipping industry



Intermodal shipping containers



A new shipping ecosystem

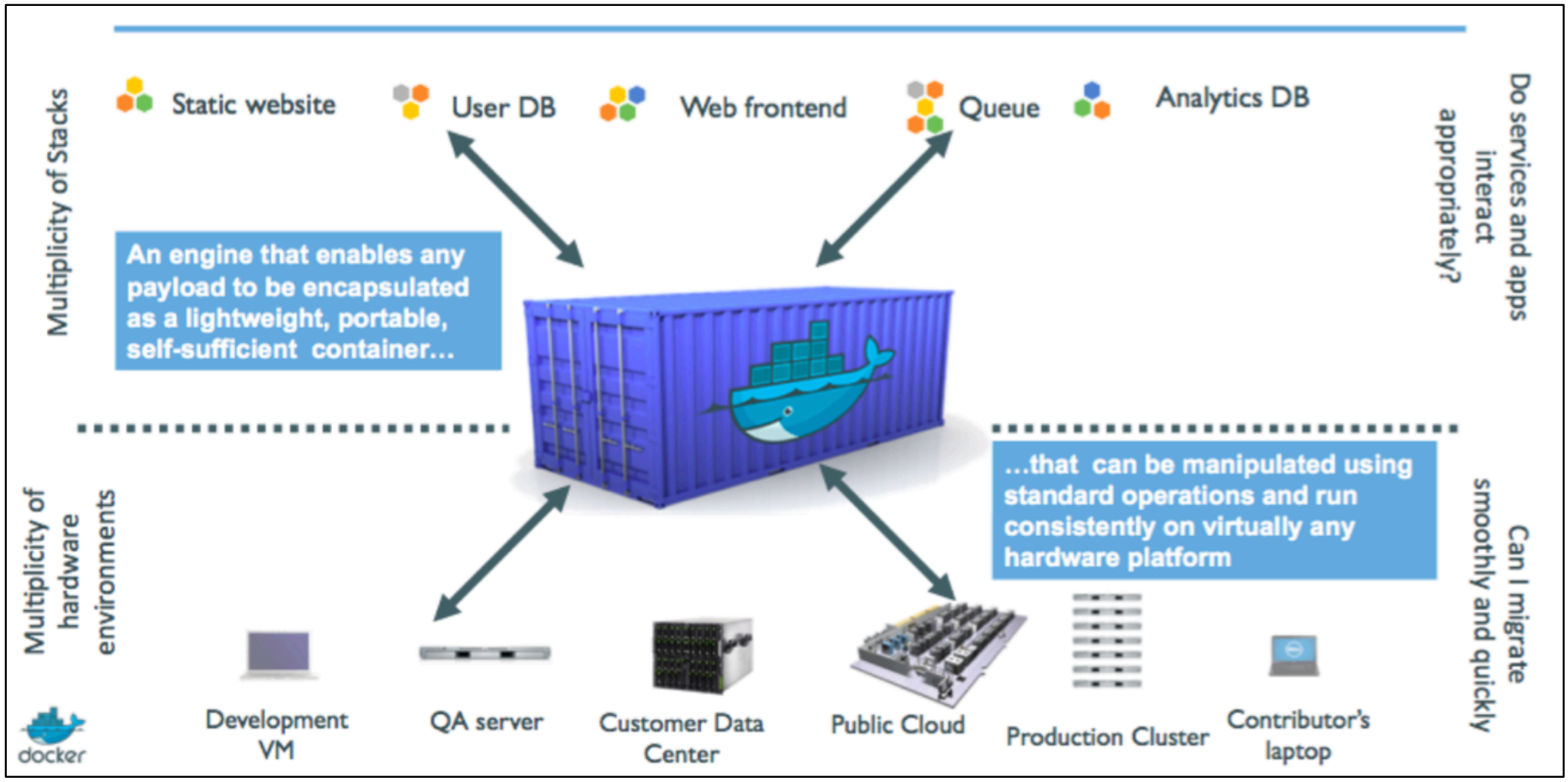
- 90% of all cargo now shipped in a standard container
- Order of magnitude reduction in cost and time to load and unload ships
- Massive reduction in losses due to theft or damage
- Huge reduction in freight cost as percent of final goods (from >25% to <3%)

→ massive globalization

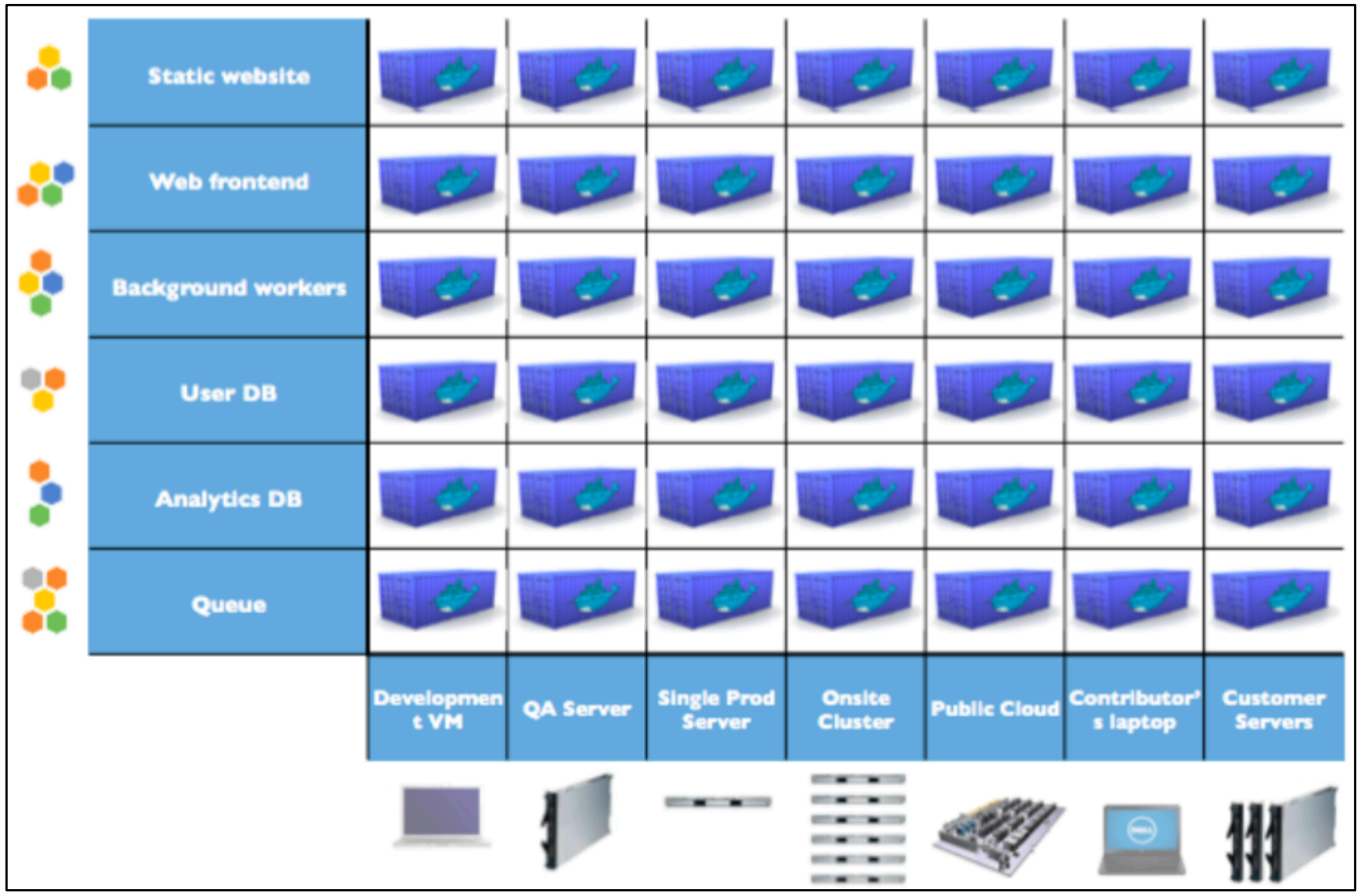
- 5000 ships deliver 200M containers per year

docker

A shipping container system for applications



Eliminate the matrix from hell



Results

- Escape dependency hell:
  - If the Docker file builds on your machine, it will build anywhere
  - Never again "worked in dev - ops problem now!”

- Create development, integration, QA environments in minutes
- Implement reliable CI easily
- Use container images as build artefacts
- Images contain all the libraries, dependencies, etc. needed to run the app.
- Images are bigger, but they are broken down into layers.
- Only ship layers that have changed
- Save disk, network, memory usage.

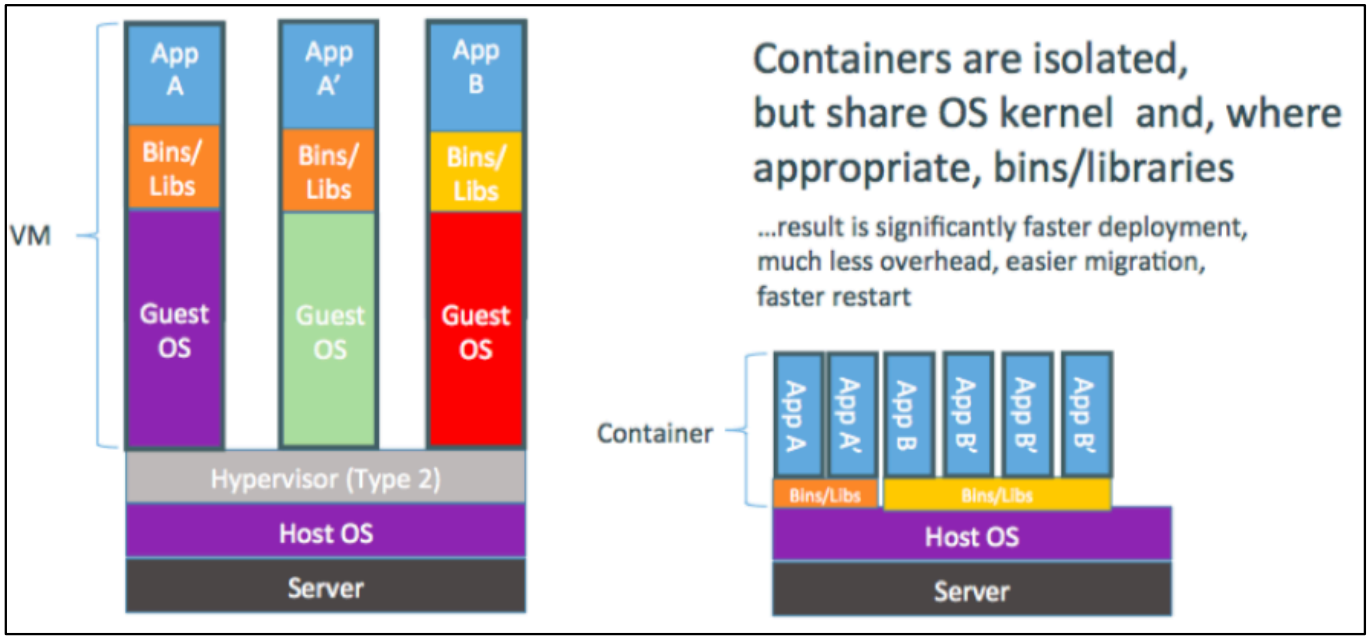
Results

- Normalization: same environment (container image) for:
  - development
  - jobs on the computing grid
  - continuous integration
  - peer review
  - demonstrations, tutorials
  - technology transfer

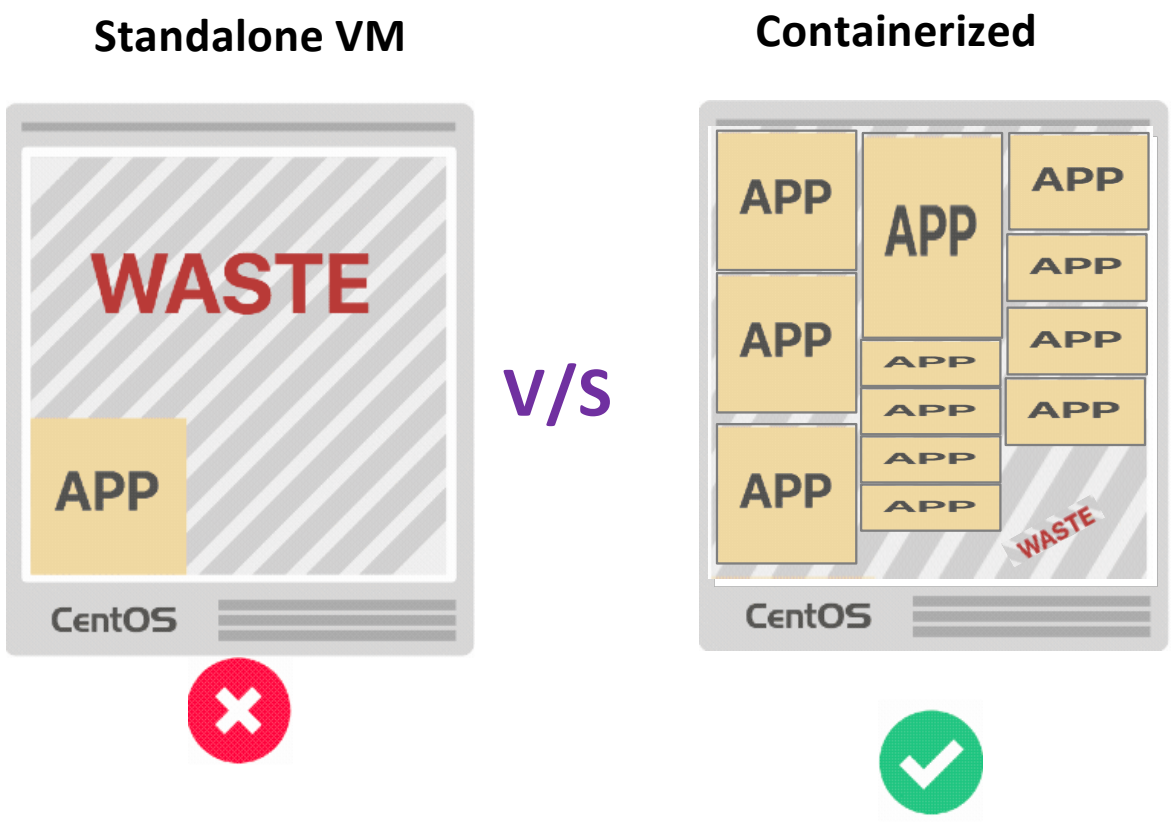
- Archival (ever tried to reuse old codes)
  - source → Dockerfile = recipe to rebuild the environment from scratch
  - binary → docker image = immutable snapshot of the software
  - with its runtime environment can be rerun it at any time later

Results

- Less Overhead:



Higher density of applications than virtual machines



Installing Docker

Installing Docker, enable and start the service

[[Make sure that you have closed all the old Putty-session and started a *new* Putty session]]

```
[root@docker ~]# yum install -y docker

[root@docker ~]# ls -l docker.setting.txt
-rw-r--r--. 1 root root 160 Jun  7 01:49 docker.setting.txt

[root@docker ~]# cat docker.setting.txt >>  /etc/sysconfig/docker

[root@docker ~]# cat /etc/sysconfig/docker

[root@docker ~]# systemctl enable docker

[root@docker ~]# systemctl start docker

[root@docker ~]# systemctl status docker
```

Installing Docker, enable and start the service (OPTIONAL)

[[if the docker.setting.txt file is missing, follow these steps; else ignore]]

You can use wget to fetch the file

```
[root@docker ~]# wget http://172.20.11.97/repos/Docker-Lab-Files/docker-lab-support-files/docker.setting.txt

[root@docker ~]# cat docker.setting.txt >>  /etc/sysconfig/docker
```

Or Manually execute the steps below:

```
[root@docker ~]# echo "INSECURE_REGISTRY='--insecure-registry registry.example.com:5000'" >> /etc/sysconfig/docker
[root@docker ~]# echo "ADD_REGISTRY='--add-registry registry.example.com:5000'" >> /etc/sysconfig/docker
[root@docker ~]# echo "BLOCK_REGISTRY='--block-registry all'"  >> /etc/sysconfig/docker

[root@docker ~]# systemctl enable docker

[root@docker ~]# systemctl start docker

[root@docker ~]# systemctl status docker
```

Check the version (you can skip this slide if Docker images are available in registry)

```
[root@docker ~]# docker version
Client:
 Version:         1.12.6
 API version:     1.24
 Package version: docker-1.12.6-61.git85d7426.el7.centos.x86_64
 Go version:      go1.8.3
 Git commit:      85d7426/1.12.6
 Built:           Tue Oct 24 15:40:21 2017
 OS/Arch:         linux/amd64

Server:
 Version:         1.12.6
 API version:     1.24
 Package version: docker-1.12.6-61.git85d7426.el7.centos.x86_64
 Go version:      go1.8.3
 Git commit:      85d7426/1.12.6
 Built:           Tue Oct 24 15:40:21 2017
 OS/Arch:         linux/amd64
```

Preload all Docker images that we need for our lab

We will need to load all the Docker images required for the Docker lab since we are working in a restricted lab environment. All the images are available in your 'root' user home directory. This needs to be done an all nodes.

```
[root@docker ~]# cd /root/docker-images

[root@docker|/root/docker-images]# sh -x import-for-docker-lab.sh

[this will take few minutes]
```

(you can skip this slide if Docker images are available in registry)

Verify that container images are available

To check imported images use the command below (will not work if using Registry)

```
[root@docker ~]# docker images
REPOSITORY                          TAG                 IMAGE ID            CREATED             SIZE
docker.io/centos                    latest              3fa822599e10        2 weeks ago         203.5 MB
docker.io/busybox                   latest              6ad733544a63        5 weeks ago         1.129 MB
docker.io/mhausenblas/simpleservice 0.5.0               601917f29430        7 months ago        682.6 MB
docker.io/jpetazzo/clock            latest              12068b93616f        2 years ago         2.43 MB
```

To check the images in the registry run the command below (will not work if you import Images)

```
[root@docker ~]# curl -X GET http://registry.example.com:5000/v2/_catalog
{"repositories":["centos/python-35-centos7","centos/ruby-22-
centos7","kubernetes/guestbook","mhausenblas/simpleservice","openshift/hello-openshift",. . . }
```

Our First Few Containers

Hello World

In your Docker environment, just run the following command

```
# docker run busybox echo hello world

Unable to find image 'busybox:latest' locally
Trying to pull repository registry.example.com:5000/busybox ...
latest: Pulling from registry.example.com:5000/busybox
d070b8ef96fc: Pull complete
Digest: sha256:c7b0a24019b0e6eda714ec0fa137ad42bc44a754d9cea17d14fba3a80ccc1ee4
Status: Downloaded newer image for registry.example.com:5000/busybox:latest
hello world
```

We used one of the smallest, simplest images available: **busybox**.
busybox is typically used in embedded systems (phones, routers...)
We ran a single process and echo'ed **hello world**.

Hello World, again….

Run the same command once more and see the difference…

```
# docker run busybox echo hello world
```

What is the output? What is the difference?

How about a centos container…

Let us launch a centos container

```
# docker run -it centos:7
[root@151eb148d8a7 /]#
[root@151eb148d8a7 /]# cat /etc/centos-release
CentOS Linux release 7.4.1708 (Core)
[root@151eb148d8a7 /]#
[root@151eb148d8a7 /]# uptime
14:22:54 up  3:13,  0 users,  load average: 0.67, 2.30, 2.20
[root@151eb148d8a7 /]#
```

New bare-bones container with centos OS

-i tells Docker to connect us to the container's STDIN.

-t tells Docker that we want a pseudo-terminal.

What is the uptime output? Why is it showing that the docker container is up for a very long time?

Lets run the ‘banner’ command in Centos

Try to run Banner

```
[root@151eb148d8a7 /]# banner Hello
bash: banner: command not found

[root@151eb148d8a7 /]#
```

Follow Instruction as below to configure the repository [for Katakoda Playground labs, ignore below and see next slide]:

```
rm -f /etc/yum.repos.d/CentOS*

curl http://172.20.11.97/repos/dockerimages/docker-lab-files/CentOS.repo -o /etc/yum.repos.d/CentOS.repo

ls -l /etc/yum.repos.d/CentOS.repo

yum clean all

yum repolist
```

ONLY For Katakoda Playground

Use this command below for Katakoda Playground Labs

```
[root@151eb148d8a7 /]# yum install epel-release -y
```

How about a centos container…

Now install ‘banner’ and run banner

```
[root@151eb148d8a7 /]# yum install   -y banner


[root@151eb148d8a7 /]# banner Hello

#     #  #######  #        #        #######
#     #  #        #        #        #     #
#     #  #        #        #        #     #
#######  #####    #        #        #     #
#     #  #        #        #        #     #
#     #  #        #        #        #     #
#     #  #######  #######  #######  #######

[root@151eb148d8a7 /]#
```

Check the packages in container and in the host server

Check the number of packages in container and exit with ‘^D or exit’

```
[root@151eb148d8a7 /]# rpm -qa | wc
    147     147    4462
[root@151eb148d8a7 /]#
[root@151eb148d8a7 /]# exit
exit
```

Once in the host server, run the ’banner command’ and the package count command

```
# banner
sh: banner: command not found
#
# rpm -qa | wc
    331     331   10900
#
```

Did the banner command work? How many packages are there? What is your conclusion?

Host and containers are independent things

- We ran an centos container on an centos host.
- But they have different, independent packages and are isolated
- Installing something on the host doesn't expose it to the container. And vice-versa.
- We can run any container on any host.

How about launching another centos container…

Let us launch a second container and try to run banner

```
# docker run -it centos
[root@7ad722fc444c /]#
[root@7ad722fc444c /]# banner Hello from Container two...
bash: banner: command not found
[root@7ad722fc444c /]#
[root@7ad722fc444c /]#

<<keep this session running>>
```

We launched and new container
'banner' command is not there
This is a new installation
The basic Centos image was used to launch

What happened to our other container?

Let's check where the first container is

Connect to the server in using another session using putty:

```
# docker ps | grep centos
7ad722fc444c        centos
"/bin/bash"              12 hours ago        Up 13 minutes                           elegant_lumiere

# docker ps -a | grep centos
7ad722fc444c        centos
"/bin/bash"              12 hours ago        Up 4 seconds                              elegant_lumiere
151eb148d8a7        centos
"/bin/bash"              13 hours ago        Exited (0) 13 hours ago                   awesome_pike
#
```

By default 'docker ps' will only show **running** container. If we execute 'docker ps -a', we can see both the container; one is running and other has 'Exited' and is 'stopped'.

Why did it stop when we 'Exited'?
Can we start the container with id **151eb148d8a7** and connect to it again?
Is 'banner' packages still there in the container?

Try starting the first container

From the second putty session:

```
# docker start 151eb148d8a7
151eb148d8a7

# docker ps | grep centos
7ad722fc444c        centos
"/bin/bash"              12 hours ago        Up 8 minutes                                    elegant_lumiere
151eb148d8a7        centos
"/bin/bash"              13 hours ago        Up About a minute                               awesome_pike
#
```

Now both containers are running

Attaching to the first container

From the second putty session:

```
# docker attach 151eb148d8a7
[root@151eb148d8a7 /]#
[root@151eb148d8a7 /]#
[root@151eb148d8a7 /]# banner Hello new
```

What is the output of the banner command?
Exit from any one container and use ‘docker ps’ commands; what do you see?
How do we exit (detach) from a container and still keep it running?

Starting a non-interactive container

Launch a new container which continuously displays time every second:

```
# docker run jpetazzo/clock
Mon Dec 11 04:47:03 UTC 2017
Mon Dec 11 04:47:04 UTC 2017
Mon Dec 11 04:47:05 UTC 2017
^C
```

This container will run forever.
To stop it, press **^C**.

What is the state of the container? So, how do we run the container in background?

Run a container in the background

Containers can be started in the background, with the -d flag (daemon mode):

```
# docker run -d jpetazzo/clock
3391c63fb240c478f6d6f835f1466b32d8b16c1d2f904425f7e4cd0c9f0c763a
#
```

We don't see the output of the container (like we saw earlier)

The output you see is the ID of the container (yes its that long); what you saw in the outputs earlier is truncated version of this (just first 12 characters)

List the container

Use ‘docker ps’ to list the

```
# docker ps | grep clock
3391c63fb240        jpetazzo/clock
"/bin/sh -c 'while da"   18 minutes ago      Up 18 minutes                           prickly_turing
```

Start some more containers in the background

```
# docker run -d jpetazzo/clock
954a68eaf4314c4ad13ecef27a6e63620708b503d16265581d7e53bcf25df935
#
# docker run -d jpetazzo/clock
023e561a534be1126d441219e080324ac1a0b6022afaaa2219473cebfee76f0e
#
#
```

List the containers

```
# docker ps | grep clock
023e561a534b        jpetazzo/clock
"/bin/sh -c 'while da"   2 minutes ago       Up 2 minutes                            hopeful_yalow
954a68eaf431        jpetazzo/clock
"/bin/sh -c 'while da"   2 minutes ago       Up 2 minutes                            silly_brahmagupta
3391c63fb240        jpetazzo/clock
"/bin/sh -c 'while da"   23 minutes ago      Up 23 minutes                           prickly_turing
#
```

Some more useful options of ’docker ps’

List the last container started

```
# docker ps -l
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
NAMES
023e561a534b        jpetazzo/clock      "/bin/sh -c 'while da"   4 minutes ago       Up 4 minutes
hopeful_yalow
```

List container with just the container IDs

```
# docker ps -q
023e561a534b
954a68eaf431
3391c63fb240

#  docker ps -lq
023e561a534b
```

Now let us see how to see the output of the containers…

All output produced by container is logged and can be seen by the ‘docker logs’ command:

```
# docker logs 023e561a534b
Mon Dec 11 05:13:33 UTC 2017
Mon Dec 11 05:13:34 UTC 2017
Mon Dec 11 05:13:35 UTC 2017
Mon Dec 11 05:13:36 UTC 2017
<<this is truncated here>>
```

The ‘logs’ sub-command will output the entire logs of the container; we can use the --tail option to limit just like the Unix ‘tail’ command. You can also use the ‘--follow’ option to follow the logs.

```
# docker logs --tail 5 --follow 023e561a534b
Mon Dec 11 05:34:14 UTC 2017
Mon Dec 11 05:34:15 UTC 2017
Mon Dec 11 05:34:16 UTC 2017
Mon Dec 11 05:34:17 UTC 2017
Mon Dec 11 05:34:18 UTC 2017
```

Stopping our containers

Earlier we saw that we when we exited the container (using ^D or exit or ^C), it stopped for container that was not running in the background.

Now, let us see how to stop containers running in background (detached mode).

Method #1: Killing it using the **`docker kill`** command. This will send the Unix **KILL** signal to the container and it will terminate the container immediately. Please note that the Unix KILL signal cannot be intercepted.

Method #2: Stopping it using the **`docker stop`** command. This will send the Unix **TERM** signal to the container. This is a graceful way to shutdown a container. However, if the container is still running after 10-seconds, it will send the KILL signal and terminate the container immediately.

Stopping our containers

```
# docker ps | grep clock
<<see output from previous slides>>

# docker stop 023e561a534b
# docker ps | grep clock
```

How long did it take to stop the 023e561a534b container?

```
# docker kill 954a68eaf431
# docker kill 3391c63fb240
# docker ps | grep clock
```

How long did it take to stop (kill) the last two containers? What are your observations?

List stopped containers

We already saw the use of ‘docker ps -a ‘ option:

```
# docker ps -a | grep clock


<<see output from previous slides>>
```

Background and foreground containers

It is important to note that there is no difference between containers running in the background or foreground. It is irrelevant from Docker's POV.

All containers run the same way, whether there is a client attached to them or not.

And, it is always possible to detach from a container, and to reattach to a container as long as it is in the running state.

Detaching from a container started with '-it'

For any container started with '-it' option you can detach from it using the **Ctrl-P + Ctrl-Q** key sequence. If you still have the very first two containers that was launched with '-it' option you can detach it with this sequence as below (else start a fresh new container with '-it' and do the below):

```
# docker run -it centos
[root@e62bdae9b7e5 /]#
[root@e62bdae9b7e5 /]#

<<enter the Ctrl-P + Ctrl-Q key sequence>>
#
# docker ps -a | grep e62bdae9b7e5
```

What is the output of the above command? Can we still see the container running?
Is the behavior different from '^D' or exit?

Detaching from a container by killing the docker client

You can also detach by killing the Docker client

```
# docker run  jpetazzo/clock
.
.
.
<<enter the Ctrl-P + Ctrl-Q key sequence>>
```

Did the **Ctrl-P + Ctrl-Q** sequence work? What about **Ctrl-C**?

Detaching from a container by killing the docker client

If the container was started without -it You won't be able to detach with **Ctrl-P+ Ctrl-Q**. If you hit **Ctrl-C**, the signal will be proxied to the container. The '**Ctrl-C**', then this would send SIGINT to the container and terminate it!! (start a new container if you used **Ctrl-C** in the earlier session):

```
# docker run  jpetazzo/clock
.
.
<<keep this running>>
```

From another SSH session, find the container ID and also find the Unix process ID of the docker client using '**ps -ef | grep clock**' and docker PID container using "**docker inspect**"

```
# docker ps | grep clock

# docker inspect -f '{{.State.Pid}}' container_name
4382

# ps -ef | grep 4382

# ps -ef | grep clock
root      1528 16440  0 12:40 pts/0    00:00:00 /usr/bin/docker-current run jpetazzo/clock

# kill -9 1528
```

Detaching from a container by killing the docker client

Check the other running session…

```
# docker run  jpetazzo/clock
Mon Dec 11 07:10:29 UTC 2017
.
.
Mon Dec 11 07:10:44 UTC 2017
Killed
```

Verify that the container is still running

```
# docker logs --tail 1 --follow  8f197ee83b90
Mon Dec 12 02:00:24 UTC 2017
Mon Dec 12 02:00:25 UTC 2017
Mon Dec 12 02:00:26 UTC 2017
Mon Dec 12 02:00:27 UTC 2017

# docker ps -l | grep clock
```

Attaching to a container

We already saw this earlier.

```
# docker attach <containerID>
Mon Dec 11 07:10:44 UTC 2017
.
.
.
```

Quick Notes:

- The container must be running for ‘attach’ to connect to
- There can be multiple clients attached to the same container
- Use ‘attach’ sub-command to interact with the container and send input to
- If you want to see the output produced by the container use the ‘logs’ sub-command

Restarting the container

We already saw this earlier. We can use the ‘start’ option.

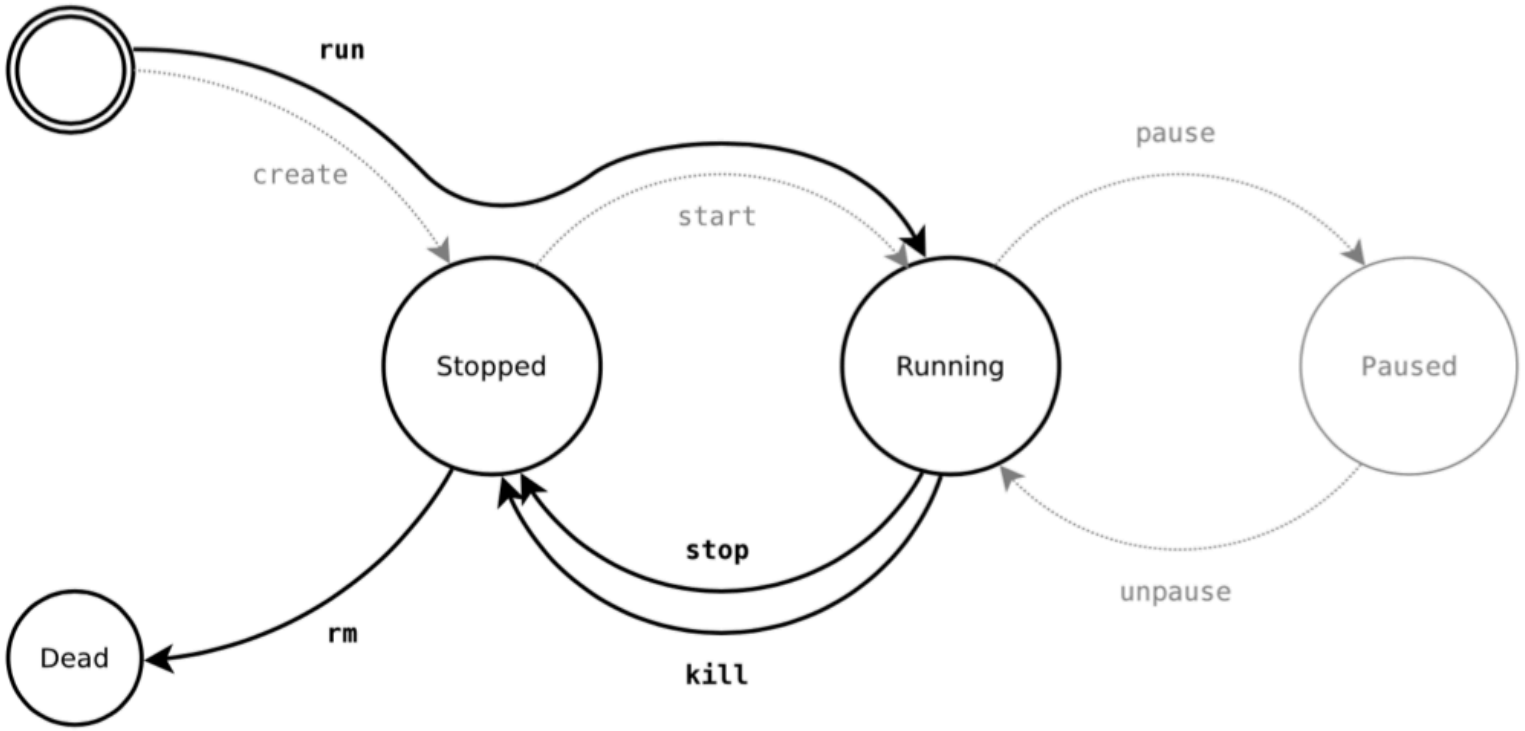
```
# docker start <containerID>
Mon Dec 11 07:10:44 UTC 2017
.
.
.
```

Quick Notes:

- The container will be restarted using the same options you launched it with.
- You can re-attach to it if you want to interact with it

Try the different sub-commands we have learned so far so that you are familiar with them.

Lifecycle of a Docker Container



Giving our container a name…

You can assign a name to the container using the '--name' option.(by default a random name is generated and is of the form adjective_name)

```
# docker run  -d --name new_clock jpetazzo/clock
b152218664441f42979c90dd1a088662bd7efd99c683a25f6f5fd4db74d6bf4d
.
.
# docker ps | grep new
b15221866444        jpetazzo/clock
"/bin/sh -c 'while da"   10 seconds ago      Up 9 seconds                              new_clock
```

Docker Images

What are Containers?

Docker explains Containers as:

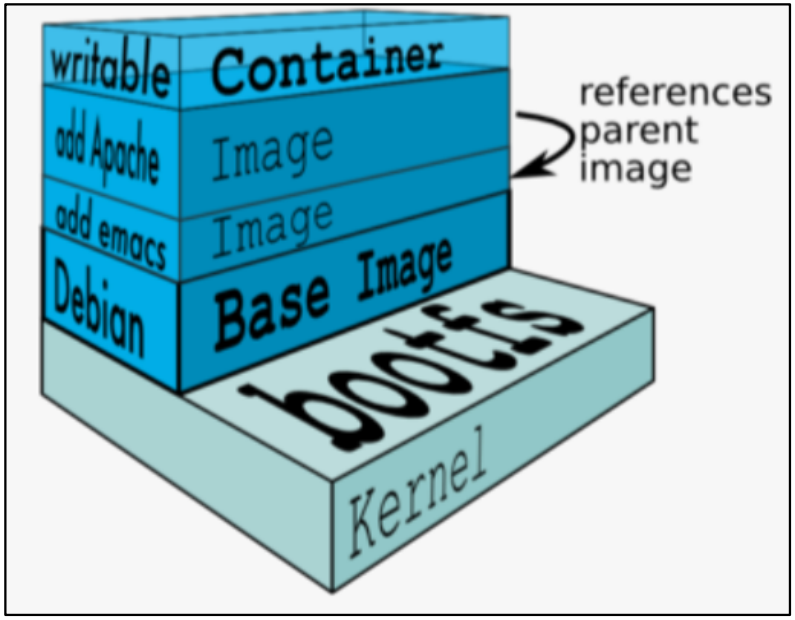
"A container image is a lightweight, standalone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings."

What it means is that once an application is packaged as a container, the underlying environment where it is run doesn't really matter. It can run anywhere, even on a multi-cloud environment.

***Source**: Docker.io and What are containers?*

What is a Docker Image?

- Image = files + metadata
- These files form the root filesystem of our container.
- The metadata can indicate a number of things:
  - the author of the image
  - the command to execute in the container when starting it
  - environment variables to be set
  - etc.
- Images are made of layers, conceptually stacked on top of each other.
- Each layer can add, change, and remove files and/or metadata.
- Images can share layers to optimize disk usage, transfer times, and memory use.
- Finally images are immutable (more on this later)



Differences between containers and images

- An image is a read-only filesystem.
- A container is an encapsulated set of processes running in a read-write copy of that filesystem.
- To optimize container boot time, copy-on-write is used instead of regular copy.
- docker run starts a container from a given image.

- If an image is read-only, how do we change it?
  - We don't.
  - We create a new container from that image.
  - Then we make changes to that container.
  - When we are satisfied with those changes, we transform them into a new layer.
  - A new image is created by stacking the new layer on top of the old image.

Creating images

- **`docker commit`**
  - Saves all the changes made to a container into a new layer.
  - Creates a new image (effectively a copy of the container).

- **`docker build`**
  - Performs a repeatable build sequence.
  - This is the preferred method!

Images namespaces

There are three namespaces for the images

- Official images
  - e.g. centos, ubuntu, busybox ....
- User (and organizations) images
  - e.g. jpetazzo/clock or weaveworks/scope
- Self-hosted images
  - e.g. registry.mylab.com:5000/my-private/image

How do you store and manage images?

Images can be stored:

- On your Docker host
  - We are doing this in this lab (will see how to see the images in next slide)
- In a Docker registry (Official Site)
  - e.g. jpetazzo/clock or weaveworks/scope
- Self-hosted registry
  - Build your own registry

And you can use the Docker client to download (pull) or upload (push) images from one place to another.

Listing current images

what images do we have on our host now? (Your output may vary)

```
# docker images
REPOSITORY                                                TAG                 IMAGE ID            CREATED             SIZE
gcr.io/google_containers/kube-apiserver-amd64             v1.8.5              ff90510bd7a8        3 days ago          194.4 MB
gcr.io/google_containers/kube-controller-manager-amd64    v1.8.5              b3710be972a6        3 days ago          129.3 MB
gcr.io/google_containers/kube-scheduler-amd64             v1.8.5              b7977f445d3b        3 days ago          54.98 MB
gcr.io/google_containers/kube-proxy-amd64                 v1.8.5              ca1d1d5680df        3 days ago          93.21 MB
docker.io/centos                                          latest              3fa822599e10        11 days ago         203.5 MB
.
.
.
And lot more....
```

Search Docker images

We can search the repository using image names or special keywords. (this will not work in lab)

```
# docker search cacti
INDEX       NAME                                    DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
docker.io   docker.io/quantumobject/docker-cacti    docker container with cacti                     45                   [OK]
docker.io   docker.io/polinux/cacti                 Cacti Server with HTTP2/support and SSL te...   7                    [OK]
docker.io   docker.io/smcline06/cacti               Cacti v1+ in docker, please note this is s...   4                    [OK]
docker.io   docker.io/leniy/cacti                                                                   3                    [OK]
docker.io   docker.io/thedollarsign/cacti           Cacti® - The Complete RRDTool-based Graph...    2                    [OK]
docker.io   docker.io/eoskin/rpi-cacti              Raspberry Pi Image for cacti, based on giu...   1
latest              3fa822599e10        11 days ago         203.5 MB
.
.
.
And lot more....
```

How do I download an image

There are two ways to download images:

- Explicitly, with docker pull
- Implicitly, when executing docker run and the image is not found locally

```
# docker pull ubuntu:latest
Trying to pull repository docker.io/library/ubuntu ...
latest: Pulling from docker.io/library/ubuntu
660c48dd555d: Pull complete
4c7380416e78: Pull complete
421e436b5f80: Pull complete
e4ce6c3651b3: Pull complete
be588e74bd34: Pull complete
Digest: sha256:7c67a2206d3c04703e5c23518707bdd4916c057562dd51c74b99b2ba26af0f79

# docker images | grep ubuntu
docker.io/ubuntu                                    latest              20c44cd7596f        3 weeks ago         122.8 MB

[[ Note: your output may vary]]
```

- We now know that images are made up of multiple layers
- Docker has downloaded all the necessary layers

Image and tags

- Images can have tags.
- Tags define image versions or variants.
- docker pull ubuntu will refer to ubuntu:latest.
- The :latest tag is generally updated often.

- Don't specify tags:
  - When doing rapid testing and prototyping.
  - When experimenting.
  - When you want the latest version.

- Do specify tags:
  - When recording a procedure into a script.
  - When going to production.
  - To ensure that the same version will be used everywhere.
  - To ensure repeatability later.

Let’s tag the image we downloaded earlier for lab purpose

```
# docker images | grep centos
 registry.example.com:5000 /centos                        latest              20c44cd7596f        3 weeks ago         122.8 MB

# docker tag registry.example.com:5000/centos registry.example.com:5000/centos:labimage

# docker images | grep centos
 registry.example.com:5000/centos              labimage            20c44cd7596f        3 weeks ago         122.8 MB
 registry.example.com:5000/centos              latest              20c44cd7596f        3 weeks ago         122.8 MB
```

Building images interactively

When we created and launched the second centos image, we did not see the ‘banner’ command in it. Also, all the effort in installing and creating an image was lost (sort of). Ideally, we should be able to make changes to an container and then reuse that to launch new containers.

We can do this using the docker commands, the steps are as follows:

- Create a container from a base image.
- Install software manually in the container, and turn it into a new image.
- Use docker sub-commands: **`docker diff`**, **`docker commit`**, and **`docker tag`** and see how it will help us build the custom image we need

Building images interactively - Step 1 - Setup the container

Start the base centos container:

```
# docker run -it centos
root@151eb148d8a7:#/
```

Building images interactively - Step 1 - Setup the container

Try to run Banner

```
[root@151eb148d8a7 /]# banner Hello
bash: banner: command not found

[root@151eb148d8a7 /]#
```

Follow Instruction as below to configure the repository from the **[root@151eb148d8a7 /]#** prompt:
[for Katakoda Playground labs, ignore below and see next slide]:

```
rm -f /etc/yum.repos.d/CentOS*

curl http://172.20.11.97/repos/dockerimages/docker-lab-files/CentOS.repo -o /etc/yum.repos.d/CentOS.repo

ls -l /etc/yum.repos.d/CentOS.repo

yum clean all

yum repolist
```

ONLY For Katakoda Playground

Use this command below for Katakoda Playground Labs

```
[root@151eb148d8a7 /]# yum install epel-release -y
```

Building images interactively - Step 1 - Setup the container

Now install ‘banner’ and run banner

```
[root@151eb148d8a7 /]# yum install   -y banner
```

Building images interactively - Step 1 - Setup the container

Verify that the ‘banner’ command is running. Finally type ‘exit’ to leave the interactive session.

```
root@151eb148d8a7:# banner MyImage
.
.
root@151eb148d8a7:# exit
```

Building images interactively - Step 2 - Inspect the changes

Use the '**docker diff**' command to see the difference between the base and new image:

```
# docker diff 151eb148d8a7
C /run
A /run/secrets
C /usr
C /usr/bin
A /usr/bin/banner
C /usr/lib
.
.
```

docker diff gives us an easy way to audit changes
Makes use of CoW security benefits
docker tracks filesystem changes
As explained before:

- An image is read-only
- When we make changes, they happen in a copy of the image.
- Docker can show the difference between the image, and its copy.
- For performance, Docker uses copy-on-write systems. (i.e. starting a container based on a big image doesn't incur a huge copy.)

Building images interactively - Step 3 - Commit our changes into a new image

Use the '**docker commit**' command will create a new layer with the changes and create a new image with this added new layer:

```
# docker commit 151eb148d8a7
sha256:73d6da5a46b635be71e32dc381adc5d50139142eb449fb2011432a5056961fc7


# docker images | grep 73d6da5a46b6
<none>                                  <none>              73d6da5a46b6        26 seconds ago      324.7 MB
```

The output of the '**docker commit**' command will be the ID of the new Image. The output of 'docker image' will show truncated image
We can use this ID to launch/run new instances

Step 4 - Using the new Image, launch a new instance

Run an new container with the new image

```
# docker run -it 73d6da5a46b6
[root@9ff96bddfe0e /]#
[root@9ff96bddfe0e /]# banner FromNewImage
```

Eureka!!! It worked. Finally all our effort and work has paid off

Step 5 - Tag the new image so that we don’t have to remember cryptic ID

Let's tag it instead

```
# docker tag 73d6da5a46b6 withbanner

# docker images | grep withbanner
withbanner                    latest              73d6da5a46b6        17 minutes ago      324.7 MB
```

And we are at the beginning of the cycle.. We can now use the tag to launch a new image:

```
# docker run -it withbanner
[root@ab8742bb81be /]#
[root@ab8742bb81be /]# banner after-tag
[root@ab8742bb81be /]#
[root@ab8742bb81be /]# exit
```

Building images using automated process with **`Dockerfile`**

When we created the image in the earlier section there was lot of manual command that we have to give. It was very interactive. For a more complex changes this may lead to erros.

A **`Dockerfile`** is a build recipe for a Docker image. It contains a series of instructions telling Docker how an image is to be constructed.

The **`docker build`** command builds an image from a **`Dockerfile`**.

Creating our image using **Dockerfile**

The new Dockerfile must be in a new, empty directory. Create a directory to hold our Dockerfile

```
[root@docker|/root]# mkdir bannerImage
[root@docker|/root]# cd bannerImage/
```

Create a file by name **"Dockerfile"** and make sure the contents are as below:

```
[root@docker bannerImage]# cat <<EOF  > Dockerfile
FROM centos:7
RUN yum -y install epel-release
RUN yum -y install banner
EOF



[root@docker bannerImage]# cat Dockerfile
```

Changes needed for our lab environment…

Ideally, if we had direct internet access we could do the below:

```
[root@docker|/root]# docker build -t bannerimg .
```

In our lab, we will use the local Centos repository

Copy the CentOS.repo to the current directory [ignore this for Katakoda labs]

```
[root@docker|~/bannerImage]# curl http://172.20.11.97/repos/dockerimages/docker-lab-files/CentOS.repo -o
CentOS.repo
```

Modify the `Dockerfile` to include the local repository…

Execute the command block below and verify the contents are similar as below:
[for Katakoda Playground labs, ignore below and see next slide]:

```
[root@docker bannerImage]# cat <<EOF > Dockerfile
FROM centos:7
RUN rm /etc/yum.repos.d/CentOS*
COPY CentOS.repo  /etc/yum.repos.d/CentOS.repo
RUN yum -y install banner
EOF
```

FROM indicates the base image for our build.

Each **RUN** line will be executed by Docker during the build.

RUN commands must be non-interactive. (No input can be provided to Docker during the build.)

COPY will literally copy the file from the current location to inside the image.

Modify the `Dockerfile` to include the local repository…

[for Katakoda Playground labs ONLY]:

```
[root@docker bannerImage]# cat <<EOF > Dockerfile
FROM centos:7
RUN yum -y install epel-release
RUN yum -y install banner
EOF
```

FROM indicates the base image for our build.

Each **RUN** line will be executed by Docker during the build.

RUN commands must be non-interactive. (No input can be provided to Docker during the build.)

COPY will literally copy the file from the current location to inside the image.

Build it….

```
[root@docker bannerImage]# docker build -t bannerimg .
```

The output of docker build looks like this:

```
Sending build context to Docker daemon 3.072 kB
Step 1 : FROM centos
 ---> 3fa822599e10
Step 2 : RUN rm /etc/yum.repos.d/CentOS*
 ---> Running in 0a722975830b
 ---> ddd721e58108
Removing intermediate container 0a722975830b
Step 3 : COPY EpelRepo.repo /etc/yum.repos.d/EpelRepo.repo
 ---> f45e5a360639
Removing intermediate container a2669eee7c99
Step 4 : RUN yum -y install banner
 ---> Running in 12382bcb6d33
<snip snip snip snip>
Complete!
 ---> ff805af79b06
Removing intermediate container 12382bcb6d33
Successfully built ff805af79b06
```

Let’s check if there is a new image…

```
[root@docker bannerImage]# docker images
REPOSITORY                          TAG                 IMAGE ID            CREATED             SIZE
bannerimg                           latest              ff805af79b06        2 minutes ago       258.3 MB
docker.io/centos                    latest              3fa822599e10        2 weeks ago         203.5 MB
```

But wait.. what all happened when we build the image?

Sending the build context to Docker…

```
Sending build context to Docker daemon 3.072 kB
```

- The build context is the “.” directory given to docker build.
- It is sent as an archive by the **Docker client** to the **Docker daemon**.
- This allows to use a remote machine to build using local files.
- If that directory is big and your link is slow this part of the build will take sometime.

Executing each step - Step 1

```
Step 1 : FROM centos
 ---> 3fa822599e10
```

- Step 1, it just uses the existing base ‘centos’ image and launches a new image for us. The image ID (`#0a722975830b`) of that can be seen in the next step

Executing each step - Step 2

```
Step 2 : RUN rm /etc/yum.repos.d/CentOS*
 ---> Running in 0a722975830b
 ---> ddd721e58108
Removing intermediate container 0a722975830b
```

- In this step, we see that it is executing the **RUN** 'rm' command.
- This is executed in the image `0a722975830b`.
- The container is committed into an image (`ddd721e58108`)
- The build container (`0a722975830b`) is removed
- The output of this step will be the base image for the next one

Executing each step - Step 3 and Step 4

```
Step 3 : COPY EpelRepo.repo /etc/yum.repos.d/EpelRepo.repo
 ---> f45e5a360639
Removing intermediate container a2669eee7c99
Step 4 : RUN yum -y install banner
 ---> Running in 12382bcb6d33
```

- These two steps also follow the same logic
- Any intermediate containers are removed

Executing each step - Step 3 and Step 4

```
.
.
Complete!
 ---> ff805af79b06
Removing intermediate container 12382bcb6d33
Successfully built ff805af79b06
```

- At the end of Step #4, we see that it has installed the ‘banner’ binaries
- Committed the changes to committed into an image (`ff805af79b06`).
- Any intermediate containers are removed (`12382bcb6d33`)

- The final image ID with the banner installed is: `ff805af79b06`

Let us see what all changes was done on our final container: `ff805af79b06`

For that we will make use of the “docker history” command. The history command lists all the layers composing an image. For each layer, it shows its creation time, size, and creation command. When an image was built with a Dockerfile, each layer corresponds to a line of the Dockerfile.

```
[root@docker bannerImage]# docker history ff805af79b06
IMAGE               CREATED             CREATED BY                                      SIZE                COMMENT
ff805af79b06        2 minutes ago       /bin/sh -c yum -y install banner                54.8 MB
f45e5a360639        2 minutes ago       /bin/sh -c #(nop) COPY file:9986668a53643d377   72 B
ddd721e58108        2 minutes ago       /bin/sh -c rm /etc/yum.repos.d/CentOS*          0 B
3fa822599e10        2 weeks ago         /bin/sh -c #(nop)  CMD ["/bin/bash"]            0 B
<missing>           2 weeks ago         /bin/sh -c #(nop)  LABEL name=CentOS Base Ima   0 B
<missing>           2 weeks ago         /bin/sh -c #(nop) ADD file:7441d818786942af84   203.5 MB
```

Let’s check this out:

- ✓ `3fa822599e10` - Original centos image
- ✓ `ddd721e58108` - We delete the CentOS.* repo files
- ✓ `f45e5a360639` - copied repo file as per our Lab requirement
- ✓ `ff805af79b06` - The final image with ‘banner‘ command

Using the new Image, launch a new instance

Run an new container with the new image

```
[root@docker bannerImage]# docker run -it bannerimg
[root@fc718905540c /]# banner Dockerfile

######   #######   #####   #    #  #######  ######   #######  ###  #       #######
#     #  #     #  #     #  #   #   #        #     #  #         #   #       #
#     #  #     #  #        #  #    #        #     #  #         #   #       #
#     #  #     #  #        ###     #####    ######   #####     #   #       #####
#     #  #     #  #        #  #    #        #   #    #         #   #       #
#     #  #     #  #     #  #   #   #        #    #   #         #   #       #
######   #######   #####   #    #  #######  #     #  #        ###  #######  #######

[root@fc718905540c /]# exit
```

Another Eureka for us!!! It worked. Once again, our effort and work has paid off

Exporting and Importing Docker Images

Create a new directory and CHDIR to that directory:

```
[root@docker|~]# cd /root

[root@docker ~]# mkdir Images
[root@docker ~]# cd Images/

[root@docker Images]# docker images | grep bannerimg
bannerimg                          latest              865bb65ebfeb        About an hour ago   258.3 MB

[root@docker Images]# docker save -o bannerimg.img bannerimg

[root@docker Images]# ls -l bannerimg.img

[root@docker Images]# docker rmi bannerimg    [[if there is an error, remove the referenced container]]

[root@docker Images]# docker images | grep bannerimg

[root@docker Images]# yum install file -y

[root@docker Images]# file bannerimg.img

[root@docker Images]# tar -tvf bannerimg.img | more

[root@docker Images]# docker load -i bannerimg.img

[root@docker Images]# docker images | grep bannerimg
```

Working with Registry

Building a simple webapp in Go …

Install Go binaries and extract the application source…

```
[root@docker|/root]# cd /root

## Install Go ##
[root@docker|/root]# tar -C /usr/local -xzf /var/tmp/go1.10.3.linux-amd64.tar.gz

[root@docker|/root]# export PATH=$PATH:/usr/local/go/bin

[root@docker|/root]# tar xf /var/tmp/simple-webapp.tar

[root@docker|/root]# cd simple-webapp/

[root@docker|/root/simple-webapp]# ls -l
total 28
-rw-r--r-- 1 501 wheel 1347 Aug  5  2018 app-server.go
-rw-r--r-- 1 501 wheel  127 Aug  5  2018 Dockerfile
-rw-r--r-- 1 501 wheel  884 Aug  5  2018 homepage.html
-rw-r--r-- 1 501 wheel  884 Aug  5  2018 homepage-with-details-v1.html
-rw-r--r-- 1 501 wheel  884 Aug  5  2018 homepage-with-details-v2.html
-rw-r--r-- 1 501 wheel   47 Aug  5  2018 HowTo.txt
-rw-r--r-- 1 501 wheel 1347 Aug  5  2018 template.go
```

Building a simple webapp in Go …

Compile the Go program and build the application image…

```
[root@docker|/root/simple-webapp]# GOOS=linux GOARCH=386 go build ./app-server.go

[root@docker|/root/simple-webapp]# ls -l app-server

[root@docker|/root/simple-webapp]# cat Dockerfile

[root@docker|/root/simple-webapp]# docker build -t simple-webapp:your-name .
Sending build context to Docker daemon  7.47 MB
Step 1/5 : FROM registry.example.com:5000/busybox:lab
Trying to pull repository registry.example.com:5000/busybox ...
lab: Pulling from registry.example.com:5000/busybox
07a152489297: Pull complete
Digest: sha256:74f634b1bc1bd74535d5209589734efbd44a25f4e2dc96d78784576a3eb5b335
Status: Downloaded newer image for registry.example.com:5000/busybox:lab
 ---> 8c811b4aec35
Step 2/5 : COPY ./app-server /home/
 ---> c1a8708048ad
Removing intermediate container 43f292ebaf4b
Step 3/5 : COPY ./homepage.html /home/
 ---> f5d0de394f2b
Removing intermediate container fd56d31e27a7
Step 4/5 : WORKDIR /home
 ---> dcf44650cf2a
Removing intermediate container 3c63efe5b614
Step 5/5 : CMD ./app-server
 ---> Running in 28d3ffe9bf84
 ---> b2875bf3a1e9
Removing intermediate container 28d3ffe9bf84
Successfully built b2875bf3a1e9
```

Building a simple webapp in Go …

Check the image

```
[root@docker|/root/simple-webapp]# docker images | grep simple-webapp
simple-webapp                                  your-name           b2875bf3a1e9        2 minutes ago       8.6 MB
```

Building a simple webapp in Go …

Install and setup a docker registry…

```
[root@docker|/root]# cd /root

[root@docker|/root]# yum -y install docker-distribution

[root@docker|/root]# cat /etc/docker-distribution/registry/config.yml

[root@docker|/root]# systemctl enable docker-distribution

[root@docker|/root]# systemctl start docker-distribution

[root@docker|/root]# systemctl status docker-distribution
```

Building a simple webapp in Go …

Add Insecure Registry to Docker Engine and restart Docker…

```
[root@docker|~]# grep docker.${PODNUM}.example.com /etc/sysconfig/docker

[root@docker|~]# wget http://172.20.11.97/repos/Docker-Lab-Files/docker-lab-support-files/add-pod-registry.sh

[root@docker|~]# cat add-pod-registry.sh
#!/bin/bash
sed -i "s/--insecure-registry registry.example.com:5000/--insecure-registry registry.example.com:5000 --insecure-registry
docker.${PODNUM}.example.com:5000/g" /etc/sysconfig/docker

sed -i "s/--add-registry registry.example.com:5000/--add-registry registry.example.com:5000 --add-registry
docker.${PODNUM}.example.com:5000/g" /etc/sysconfig/docker

[root@docker|~]#  sh add-pod-registry.sh

[root@docker|~]# grep docker.${PODNUM}.example.com /etc/sysconfig/docker

INSECURE_REGISTRY='--insecure-registry registry.example.com:5000 --insecure-registry docker.podXX.example.com:5000'
ADD_REGISTRY='--add-registry registry.example.com:5000 --add-registry  docker.podXX.example.com:5000'

[root@docker|/root]# systemctl restart docker

[root@docker|/root]# systemctl status docker
```

Building a simple webapp in Go …

Tag the image and push it to the registry…

```
[root@docker|/root]# docker tag simple-webapp:your-name docker.${PODNUM}.example.com:5000/simple-webapp:your-name

[root@docker|/root]# docker images | grep simple-webapp
docker.podXX.example.com:5000/simple-webapp         your-name           b2875bf3a1e9        38 minutes ago      8.6 MB
simple-webapp                                       your-name           b2875bf3a1e9        38 minutes ago      8.6 MB

[root@docker|/root]# docker push docker.${PODNUM}.example.com:5000/simple-webapp:your-name
The push refers to a repository [docker.podXX.example.com:5000/simple-webapp]
723e308907b9: Pushed
152acab337b2: Pushed
432b65032b94: Pushed
your-name: digest: sha256:cf23cd4cd517ab3b685e1bba3d2fd80f83c291a2ac0fc84e0f7dcb665c1b069c size: 945


[root@docker|/root]# docker push docker.${PODNUM}.example.com:5000/simple-webapp:your-name
The push refers to a repository [docker.podXX.example.com:5000/simple-webapp]
723e308907b9: Layer already exists
152acab337b2: Layer already exists
432b65032b94: Layer already exists
your-name: digest: sha256:cf23cd4cd517ab3b685e1bba3d2fd80f83c291a2ac0fc84e0f7dcb665c1b069c size: 945
```

Building a simple webapp in Go …

Where did it store the image?

```
# cat /etc/docker-distribution/registry/config.yml

# ls -l /var/lib/registry

# ls -l /var/lib/registry/docker/registry/v2/repositories/simple-webapp

# ls -l /var/lib/registry/docker/registry/v2/repositories/simple-webapp/_layers/sha256/

# ls -l /var/lib/registry/docker/registry/v2/repositories/simple-webapp/_layers/sha256/07a152489297fc2b.....8a30fd543a160cd689ee548

# cat /var/lib/registry/docker/registry/v2/repositories/simple-webapp/_layers/sha256/07a152489297fc2b.....8a30fd543a160cd689ee548/link
sha256:07a152489297fc2bca20be96fab3527ceac5668328a30fd543a160cd689ee548

# ls -l /var/lib/registry/docker/registry/v2/blobs/sha256

# ls -l /var/lib/registry/docker/registry/v2/blobs/sha256/07/07a152489297fc2b.....8a30fd543a160cd689ee548

# yum install file -y

# file /var/lib/registry/docker/registry/v2/blobs/sha256/07/07a152489297fc2b.....8a30fd543a160cd689ee548/data
gzip compressed data

# tar ztvf /var/lib/registry/docker/registry/v2/blobs/sha256/07/07a152489297fc2b.....8a30fd543a160cd689ee548/data
```

Lets check the registry…

```
[root@docker|/root]# curl -s  -X GET http://docker.${PODNUM}.example.com:5000/v2/_catalog  | grep simple

[root@docker|/root]# curl -s -X GET http://docker.${PODNUM}.example.com:5000/v2/_catalog?n=200 | grep simple

....,"simple-webapp”,...

[root@docker|/root]# curl -s -X GET http://docker.${PODNUM}.example.com:5000/v2/simple-webapp/tags/list

{"name":"simple-webapp","tags":["your-name"]}


<<Now delete the local images>>

[root@docker|/root]# docker rmi docker.${PODNUM}.example.com:5000/simple-webapp:your-name

[root@docker|/root]# docker rmi simple-webapp:your-name
```

Try starting the simple-webapp container with the image

```
[root@docker|/root]# docker run -d simple-webapp:your-name
Unable to find image 'simple-webapp:your-name' locally
Trying to pull repository registry.example.com:5000/simple-webapp ...
Pulling repository registry.example.com:5000/simple-webapp
Trying to pull repository docker.podXX.example.com:5000/simple-webapp ...
your-name: Pulling from docker.podXX.example.com:5000/simple-webapp
07a152489297: Already exists
abb66bc3d286: Pull complete
ae24add1b78e: Pull complete
Digest: sha256:cf23cd4cd517ab3b685e1bba3d2fd80f83c291a2ac0fc84e0f7dcb665c1b069c
Status: Downloaded newer image for docker.pod23.example.com:5000/simple-webapp:your-name
9fec51246afe80752c324c9d8a5f1706833b5bb49c0bb347659e7d293fdb0c19

[root@docker|/root]# docker ps -l
CONTAINER ID        IMAGE                     COMMAND                  CREATED             STATUS              PORTS
NAMES
9fec51246afe        simple-webapp:your-name   "/bin/sh -c ./app-..."   12 seconds ago      Up 12 seconds
pedantic_fermi
```

Limiting a container's resources

How do we limit a container's resources…

By default, a container has no resource constraints and can use as much of a given resource as the host's kernel scheduler will allow. Docker provides ways to control how much memory, CPU, or block IO a container can use, setting runtime configuration flags of the `docker run` command.

Check this for more details:: https://docs.docker.com/config/containers/resource_constraints/

Run a new container with the 'centos' image and set and test CPU constrains.

```
[root@docker|/root]# docker run --cpu-quota=25000 --memory=50000000 --memory-swappiness=0 -i -t centos

[root@e134d1e9346a /]# python -c "while True: 42*42"
```

From **another** putty session check the usage using the below commands (what do you see)

- `top`
- `docker stats e134d1e9346a`

How do we limit a container's resources…

Now use **Ctrl-C** to abort the CPU intensive process and test the memory usage:

```
[root@e134d1e9346a /]# curl  http://172.20.13.125/Docker/docker-lab-files/eatmem.py -o eatmem.py



[root@e134d1e9346a /]# python eatmem.py
10mb
20mb
30mb
Killed
```

<<issue Ctrl-P + Ctrl-Q to exit this session -- DO NOT EXIT or Do Ctrl-C >>

Now let us check the details at the Host OS Level…

```
[root@docker|/root]# docker ps -lq
e134d1e9346a

[root@docker|/root]# cd /sys/fs/cgroup/cpu/system.slice/

[root@docker|/sys/fs/cgroup/cpu/system.slice]# ls -ld docker-e134d1e9346a*
drwxr-xr-x. 2 root root 0 Feb 22 21:10 docker-e134d1e9346a0f26909bca5e56f5a32bd65c65d5266078507d3b7b6b115e2011.scope

[root@docker|/sys/fs/cgroup/cpu/system.slice]# cd docker-e134d1e9346a*

[root@docker|/sys/fs/cgroup/cpu/system.slice/docker-e134.......scope]# cat cpu.cfs_quota_us
25000

[root@docker|/root]# cd /sys/fs/cgroup/memory/system.slice/

[root@docker|/sys/fs/cgroup/cpu/system.slice]# ls -ld docker-e134d1e9346a*
drwxr-xr-x. 2 root root 0 Feb 22 21:10 docker-e134d1e9346a0f26909bca5e56f5a32bd65c65d5266078507d3b7b6b115e2011.scope

[root@docker|/sys/fs/cgroup/cpu/system.slice]# cd docker-e134d1e9346a*

[root@docker|/sys/fs/cgroup/cpu/system.slice/docker-e134.......scope]# cat memory.limit_in_bytes
49999872
```

Now, kill the container (e134d1e9346a) and see what happens to these above directories?

Container Networking Basis

A simple web server using nginx

Run an new container with the ‘nginx’ image (make sure that you have the ‘nginx’ image)

```
[root@docker|/root]# docker run -d -P nginx
38b0d56407214b1d24c53c237279261fe91eb41c5d7eb3388a4626937b709158
```

Docker will download the image from the Docker Hub (in our case it is already downloaded)
-d tells Docker to run the image in the background.
-P tells Docker to make this service reachable from other computers

Now what? How do we connect to this web server now?

Finding our web server port

Use the docker ps command and see if we can get some help:

```
[root@docker|/root]# docker ps

38b0d5640721        nginx
 "nginx -g 'daemon off"   4 minutes ago       Up 4 minutes        0.0.0.0:32768->80/tcp   romantic_roentgen
```

The web server is running on ports **80** inside the container. Those ports are mapped to ports **32768** on our Docker host. You can also use iptables command to verify the NAT rules from **32768->80** You can also use the **docker port <containerID> 80** command to get the port.

Let's try connecting to the server and see if it is working (make sure the port is as per your container)

```
[root@docker|/root]# docker port 38b0d5640721 80
0.0.0.0:32768

[root@docker|/root]# netstat -an | grep 32768
tcp6       0      0 :::32768                :::*                    LISTEN

[root@docker|/root]# iptables -L -n -t nat | grep 32768
```

Connecting to our web server (GUI)

Point your browser to the PODURL:PORT below; make sure that the port below is as per you container. In this case it is **“32768”**.

```
[root]# export ID=`docker ps -l -q`

[root]# export PORT=`docker inspect --format '{{ (index (index .NetworkSettings.Ports "80/tcp") 0).HostPort }}' $ID`

[root]# echo $PODURL:$PORT/

http://docker.podX.example.com:32768/
```

Finding the container's private IP address

Use the docker inspect command and see if we can get some help:

```
[root@docker|/root]# docker inspect --format '{{ .NetworkSettings.IPAddress }}' 38b0d5640721

172.17.0.2

[root@docker|/root]# ping -c 2 172.17.0.2
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.075 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.064 ms
```

Just in case you were not curious…

Use the docker inspect command without the additional arguments and see the details it gives:

```
[root@docker|/root]# docker inspect 38b0d5640721

[
    {
        "Id": "38b0d56407214b1d24c53c237279261fe91eb41c5d7eb3388a4626937b709158",
        "Created": "2017-12-15T07:27:24.897579921Z",
        "Path": "nginx",
        "Args": [
            "-g",
            "daemon off;"
        ],
        "State": {

<snip snip snip snip snip snip snip snip>
```

There is plenty of information in the output.. You can always use the "**--format**" to parse the output. Check the manuals for more details.

If we have an IP, why are we mapping ports?

- We are out of IPv4 addresses.
- Containers cannot have public IPv4 addresses.
- They have private addresses.
- Services have to be exposed port by port.
- Ports have to be mapped to avoid conflicts.
- And many other reasons.

Launch couple of more web server using nginx

We will use some very simple prebuild sites [if you are using internal lab, you will see 9 websites with 'ls' command below

```
[root@docker|/root]# ls -l /var/tmp/websites
```

If you are running on Katakoda Playground, run the below commands:

```
***** ONLY NEEDED IF YOU ARE USING KATAKODA PLAYGROUND *****


mkdir -p /var/tmp/websites/web{1..9}

echo "Hello from Website 1" > /var/tmp/websites/web1/index.html
echo "Hello from Website 2" > /var/tmp/websites/web2/index.html
echo "Hello from Website 3" > /var/tmp/websites/web3/index.html
echo "Hello from Website 4" > /var/tmp/websites/web4/index.html
echo "Hello from Website 5" > /var/tmp/websites/web5/index.html
echo "Hello from Website 6" > /var/tmp/websites/web6/index.html
echo "Hello from Website 7" > /var/tmp/websites/web7/index.html
echo "Hello from Website 8" > /var/tmp/websites/web8/index.html
echo "Hello from Website 8" > /var/tmp/websites/web9/index.html
```

Launch couple of more web server using nginx

```
[root@docker|/root]# docker run -v /var/tmp/websites/web1:/usr/share/nginx/html:ro  -d -P nginx
68b046c6b0a4c7755b5eab8156a0b310d3dfa7207978c86307f9eb7c6dc7c738

[root@docker|/root]# docker run -v /var/tmp/websites/web2:/usr/share/nginx/html:ro -d -P nginx
a7f103904dbf6a56cb81e0ce9902bb2cb46eb0864009a92f79f0120c22a13523

[root@docker|/root]# docker run -v /var/tmp/websites/web3:/usr/share/nginx/html:ro -d -P nginx
2f3e662fb6e88b3dc36d208c8ce0ab62064b7be1c4f8c15c6407dbda2b6de14d

[root@docker|/root]# docker ps | grep nginx
2f3e662fb6e8        nginx
"nginx -g 'daemon off"   6 seconds ago       Up 6 seconds        0.0.0.0:32771->80/tcp   zen_heisenberg
a7f103904dbf        nginx
"nginx -g 'daemon off"   9 seconds ago       Up 8 seconds        0.0.0.0:32770->80/tcp   condescending_mcnulty
68b046c6b0a4        nginx
"nginx -g 'daemon off"   11 seconds ago      Up 11 seconds       0.0.0.0:32769->80/tcp   peaceful_davinci
38b0d5640721        nginx
"nginx -g 'daemon off"   15 minutes ago      Up 15 minutes       0.0.0.0:32768->80/tcp   romantic_roentgen
```

Manual allocation of port numbers

Is there a way I can set these port myself rather than Docker giving me random ports? Wish granted.. (just make sure there are no port clash; else you will get “port is already allocated” error)

```
<< exposed on port 80 >>
[root@docker|/root]# docker run -v /var/tmp/websites/web4:/usr/share/nginx/html:ro -d -p 80:80 nginx
5505c1f893bf3e1b214295fe23ef9a0b276cbe503d95143d45dd43b410fb6a39

[root@docker|/root]# docker port 5505c1f893bf 80
0.0.0.0:80

<< exposed on port 8000 >>
 [root@docker|/root]# docker run -v /var/tmp/websites/web5:/usr/share/nginx/html:ro  -d -p 8000:80 nginx
05b86589d46e196cea7b186515d7771a055ee5206aacf1617c16c1288ec3751c

[root@docker|/root]# docker port 05b86589d46e  80
0.0.0.0:8000

<< Just for fun, exposed on port 8080 and 8888 >>
[root@docker|/root]# docker run -v /var/tmp/websites/web6:/usr/share/nginx/html:ro  -d -p 8080:80 -p 8888:80 nginx
df1d270f9219c307f965d0d1cdde056df2b6790435db9d35405cdfdefe913f1a

[root@docker|/root]# docker port df1d270f9219  80
0.0.0.0:8888
0.0.0.0:8080
```

Wish I could run multiple version of ‘nginx’… Ok.. wish granted again!!!! Enjoy...

```
[root@docker|/root]# docker images | grep nginx
docker.io/nginx                                 1.12                dfe062ee1dc8        3 days ago          108.3 MB
docker.io/nginx                                 latest              f895b3fb9e30        3 days ago          108.5 MB
docker.io/nginx                                 1.8                 0d493297b409        20 months ago       133.2 MB
docker.io/nginx                                 1.7                 d5acedd7e96a        2 years ago         93.4 MB

[[Note: Your output may be different; if the images are missing, it will be pulled from the registry]]

[root@docker|/root]# docker run -v /var/tmp/websites/web7:/usr/share/nginx/html:ro -d -P nginx:1.8
832ef83d2bac5d4049e5d8a7403bd6239286a89aec39905e4c29fc5e7443694c

[root@docker|/root]# docker run -v /var/tmp/websites/web8:/usr/share/nginx/html:ro -d -P nginx:1.7
c6103ec66fc5296c7cba9076599ebc0d9bedc42ae8fad4485322bb0114a4a42f

[root@docker|/root]# docker run -d -v /var/tmp/websites/web9:/usr/share/nginx/html:ro -P nginx:1.12
4f0b712b8635bb765e3dfb11b5deb0732005b0d3b8e78d6523f187b2aded467f

[root@docker|/root]# docker port 832ef83d2bac 80
0.0.0.0:32776

[root@docker|/root]# docker port c6103ec66fc5 80
0.0.0.0:32778

[root@docker|/root]# docker port 4f0b712b8635 80
0.0.0.0:32779
[[Check the home page.. You will see different websites all coexisting on the same server]]
```

Do you recollect the “Traditional Application Deployment problems” slide?

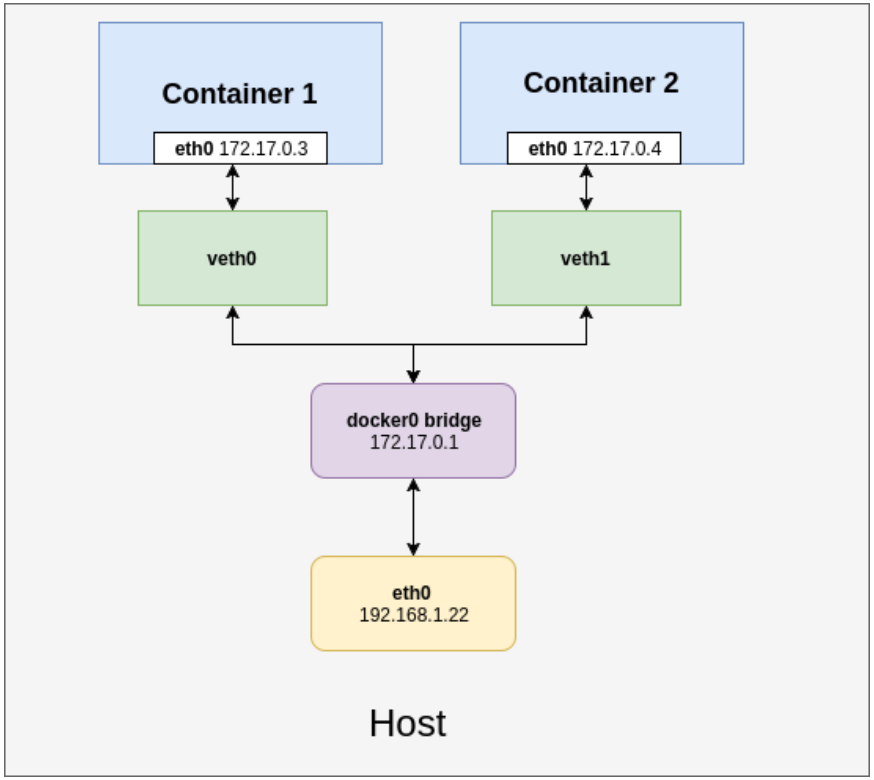
Find the “**Traditional Application Deployment problems**” slide at the start of this deck set…

Does container technology solve (or help you mitigate) some of those problems? If so how?

Understanding Networking between container and host

Relation between docker0 and eth0

Below figure show how a container connects to a host interface:



Let us trace the interfaces

Launch a new container (we can use nginx:1.12 image)

```
# docker run -d --name nwstudy  -P nginx

# docker inspect  91586a354d17 | grep -i Pid

# pid="$(docker inspect -f '{{.State.Pid}}' "nwstudy")"

# echo $pid
2620

# mkdir -p /var/run/netns

# ln -s /proc/$pid/ns/net /var/run/netns/nwstudy

# ip netns
nwstudy

# docker inspect --format '{{ .NetworkSettings.IPAddress }}' nwstudy
172.17.0.12
```

Let us trace the interfaces

```
<<  Use ‘ip netns‘ command to see the details inside the Namespace ‘nwstude‘ >>

# ip netns exec nwstudy ip a
        1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
            link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
            inet 127.0.0.1/8 scope host lo
            valid_lft forever preferred_lft forever
            inet6 ::1/128 scope host
            valid_lft forever preferred_lft forever
        29: eth0@if30: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
            link/ether 02:42:ac:11:00:0c brd ff:ff:ff:ff:ff:ff link-netnsid 0
            inet 172.17.0.12/16 scope global eth0
            valid_lft forever preferred_lft forever
            inet6 fe80::42:acff:fe11:c/64 scope link
            valid_lft forever preferred_lft forever
```

Let us trace the interfaces

```
# ip a | grep "30: veth"
      30: vethc9f09cd@if29: <BROADCAST,MULTICAST,UP,LOWER_UP> ..... master docker0 state UP group default

# yum install bridge-utils -y

# brctl show

# brctl show | grep vethc9f09cd
        vethc9f09cd

# docker exec -it nwstudy  bash -c 'cat /sys/class/net/eth0/iflink'
30


# grep -l 30 /sys/class/net/veth*/ifindex
/sys/class/net/vethc9f09cd/ifindex


# ip a s  vethc9f09cd

[[Remember this virtual interface (e.g. vethc9f09cd) name. We will need it a later lab.]]
```

Running arbitrary "network" commands inside Docker/Kubernetes pods using "nsenter" command

```
Example a. Find the IP address
    ------------------------------

    # nsenter -t ${pid} -n ip a s


Example b. Find the 'routes' using 'netstat'
    -----------------------------------------

    # nsenter -t ${pid} -n netstat -rn


Example c. Find the LISTENING ports
    ----------------------

    # nsenter -t ${pid} -n netstat -tuln | grep LISTEN
```

Running `tcpdump` for containers inside Docker/Kubernetes pods using "nsenter" command

```
# nsenter -t ${pid} -n tcpdump
```

Running `tcpdump` for containers inside Docker/Kubernetes pods (another way)

From the earlier lab, we saw that the virtual interface of the container is `vethc9f09cd`'.

Now, we can use this virtual interface to perform network packet capture's on as root

```
    # tcpdump -i vethc9f09cd
```

[[This is the virtual interface from the earlier lab]]

Restricting all arbitrary commands to the namespace ONLY

```
# nsenter --target ${pid}  --mount --uts --ipc --net --pid



Now run below commands and see the outputs

    . ps -aux
    . ps -ef
    . ifconfig -a
    . mount
    . netstat -an
    . netstat -rn
    . ss
```

Container Network Drivers

Container Network Drivers

The Docker Engine supports many different network drivers.

The built-in drivers include:

- bridge (default)
- none
- host
- container

The driver is selected with docker run --net ....

We will quickly discuss these on the next few slides.

Bridge Network Driver

Bridge Network Driver

- By default, the container gets a virtual eth0 interface. (In addition to its own private lo loopback interface.)
- That interface is provided by a veth pair.
- It is connected to the Docker bridge.
- It is named docker0 by default
- Addresses are allocated on a private, internal subnet.
- By default Docker uses 172.17.0.0/16
- Outbound traffic goes through an iptables MASQUERADE rule.
- Inbound traffic goes through an iptables DNAT rule.
- The container can have its own routes, iptables rules, etc.

Null Network Driver

Null Network Driver

- Container is started with **docker run --net none** ...
- It only gets the lo loopback interface. No eth0.
- It can't send or receive network traffic.
- Useful for isolated/untrusted workloads.

Host Network Driver

Host Network Driver

- Container is started with **docker run --net host** ...
- It sees (and can access) the network interfaces of the host.
- It can bind any address, any port (for ill and for good).
- Network traffic doesn't have to go through NAT, bridge, or veth.
- Performance = native!
- Use cases:
- Performance sensitive applications (VOIP, gaming, streaming...)
- Peer discovery

Container Network Driver

Container Network Driver

- Container is started with **docker run --net container:id** ...
- It re-uses the network stack of another container.
- It shares with this other container the same interfaces, IP address(es), routes, iptables rules, etc.
- Those containers can communicate over their lo interface. (i.e. one can bind to 127.0.0.1 and the others can connect to it.)

The Container Network Model (CNM)

The Container Network Model

The Container Network Model

- The CNM was introduced in Engine 1.9.0 (November 2015).
- The CNM adds the notion of a network, and a new top-level command to manipulate and see those networks: docker network

```
[root@docker|/root]# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
d421b809e341        bridge              bridge              local
6025b2c820f3        host                host                local
e3d1c8935565        none                null                local
```

What's in the Container Network

- Conceptually, a container network is a virtual switch.
- It can be local (to a single Engine) or global (spanning multiple hosts).
- A network has an IP subnet associated to it.
- Docker will allocate IP addresses to the containers connected to a network.
- Containers can be connected to multiple networks.
- Containers can be given per-network names and aliases.
- The names and aliases can be resolved via an embedded DNS server.
- A network is managed by a driver.
- A new multi-host driver, overlay, is available out of the box. More drivers can be provided by plugins (OVS, VLAN...)
- A network can have a custom IPAM (IP allocator).

Creating a network

Let's create a network called ‘webnet’

```
[root@docker|/root]# docker network create webnet
3840e964245d46ad64dcd12dca8def69b233d9d3d6522ab7c9bb3e59fabfaf0d
```

Check the newly created network:

```
[root@docker|/root]# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
d421b809e341        bridge              bridge              local
6025b2c820f3        host                host                local
013f0764df3d        isolated            bridge              local
e3d1c8935565        none                null                local
3840e964245d        webnet              bridge              local
```

Placing containers on a new network

Create a new container named “web1” and connect it to the network ‘webnet’

```
[root@docker|/root]# docker run -d --name web1 --net webnet nginx
4fc513e7edc6a224816ccb33ed5da0587a3741a8c3d58402ad7ba349dc0da233

[root@docker|/root]# docker ps -lq
4fc513e7edc6
```

Accessing the newly created containers

create another container on this webnet network

```
[root@docker|/root]# docker run -ti --net webnet centos
[root@665db9266403 /]# ping -c 2 web1
PING web1 (172.18.0.2) 56(84) bytes of data.
64 bytes from web1.webnet (172.18.0.2): icmp_seq=1 ttl=64 time=0.049 ms
64 bytes from web1.webnet (172.18.0.2): icmp_seq=2 ttl=64 time=0.132 ms
```

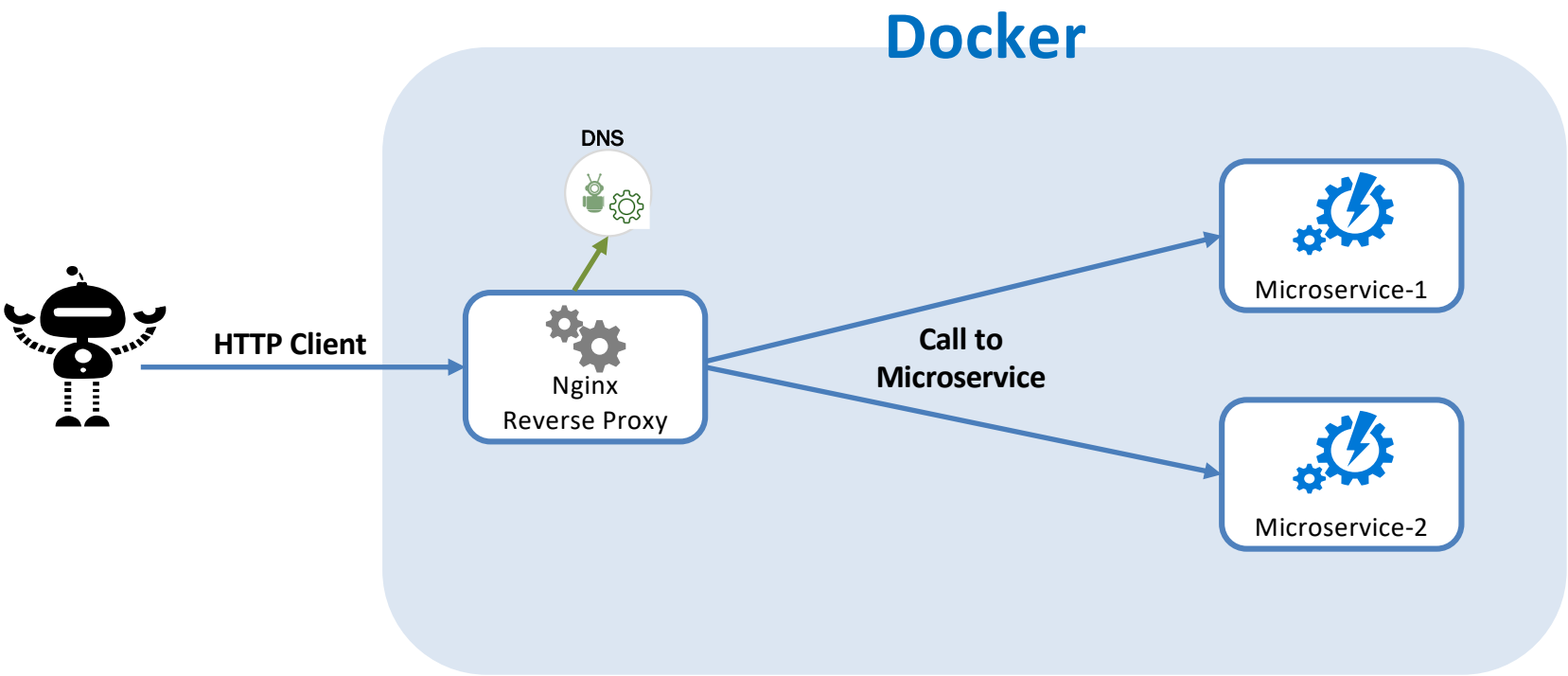
We can see that from this new container, we can resolve and ping the other one, using its assigned name. Docker Engine uses dynamic resolver to resolve these names.

Service discovery with containers

Service discovery with containers in action

- Let's try to run a demo application that has three containers.
- We will create a new network with a specific CIDR
- Place all the containers on the new network.
- Two container that returns a simple output via an URL.
- Automated DNS entry management based on Docker container name
- Call one service from another using service name as short host name
- One is a nginx reverse proxy server to access these servers from outside the network.
- Configuring Nginx to act as a simple reverse proxy (API gateway) for Dockerized microservices

Microservices discovery with Docker built-in DNS and Nginx running in a container



Create our custom network in Docker (it runs built-in DNS automatically)

Create and verify user-defined network with Docker

Check HTML Guide

```
[root@docker|/root]# docker network create --subnet=172.19.0.0/16 \
                                    --gateway=172.19.0.1 --driver bridge mysrvnet
11ecdb98150d98e8d48968165f9e5c11bee6e0c7bdfc1c537019fe40a2acbfc4
```

Check the network that was just created:

```
[root@docker|/root]# docker network inspect mysrvnet
............
"Name": "mysrvnet",
        "Id": "11ecdb98150d98e8d48968165f9e5c11bee6e0c7bdfc1c537019fe40a2acbfc4",
        "Scope": "local",
        "Driver": "bridge",
........
            "Config": [
                {
                    "Subnet": "172.19.0.0/16",
                    "Gateway": "172.19.0.1"
                }
            ]
```

Start the microservice containers

We need to start our microservices containers and connect to ‘mysrvnet’ network

```
[root@docker|/root]# docker run -d --net mysrvnet --name service1 \
                              -e SERVICE_NAME=service1  beh01der/web-service-dockerized-example

868b2c37635cf87490cb2402b3cadf51f9f81d7bfc4dd26d72f75ae644d52901

[root@docker|/root]# docker run -d --net mysrvnet --name service2 \
                              -e SERVICE_NAME=service2  beh01der/web-service-dockerized-example

32c30d1fedbfe8011a970428f856267ea46b34081a8e860a3d5a80a95346db79
```

Check HTML Guide

Test microservice host name resolution with ‘curl’

We will access them using curl through short host names. We need to make sure that both, client and server have to run in Docker containers using our custom network.

We run curl command inside docker container to make services discoverable with Docker built-in DNS.

```
[root@docker|/root]# docker run --rm --net mysrvnet beh01der/web-service-dockerized-example curl -s service1:3000
Hello World!
I am service1!

[root@docker|/root]# docker run --rm --net mysrvnet beh01der/web-service-dockerized-example curl -s service2:3000
Hello World!
I am service2!
```

Setting up nginx as a reverse-proxy with DNS-based discovery

For our microservices to be useful, they must be accessible for external clients. This can be achieved thorough API Gateway pattern. In our example we will use Nginx configured as reverse proxy with DNS resolver set to Docker internal DNS.

Here is a minimal nginx.conf file satisfying our requirements. It will route all incoming HTTP requests to existing Docker containers using simple rewrite rule. We will create and save this file to /etc/nginx/nginx.conf on host file system

```
worker_processes  1;

events {
   worker_connections  1024;
}

http {
   resolver 127.0.0.11 valid=60s ipv6=off;
   resolver_timeout 10s;

   server {
      listen 80;

      location ~ /api/([^/]*)/?(.*) {
         proxy_pass http://$1:3000/$2;
      }
   }
}
```

Setting up nginx as a reverse-proxy with DNS-based discovery

Copy the configuration file from the FTP server:

[[For Reference Only]]

```
[root@docker|/root]# wget http://172.20.11.97/repos/Docker-Lab-Files/docker-lab-support-files/nginx.conf

[root@docker|/root]# ls -l nginx.conf

[root@docker|/root]# mkdir -p /etc/nginx

[root@docker|/root]# cp /root/nginx.conf /etc/nginx/nginx.conf
```

Start nginx container with Reverse-Proxy

We can start nginx: (please delete any old container bound to port :80 before running this)

```
[root@docker|/root]# cat /etc/nginx/nginx.conf
worker_processes  1;

events {
   worker_connections  1024;


------SNIP-----SNIP----SNIP------

      location ~ /api/([^/]*)/?(.*) {
         proxy_pass http://$1:3000/$2;
      }
   }
}

[root@docker|/root]# docker run -d --net mysrvnet --name nginx-reverse-proxy -p 80:80  -v
/etc/nginx/nginx.conf:/etc/nginx/nginx.conf:ro nginx:1.11.5-alpine

2557da18fae965753e94ac113a1229153400e0257e016f3bdac1393d10ac8837
```

Check HTML Guide

Access the service from outside the network:

Test and make sure that our services accessible from outside Docker mysrvnet network

```
[root@docker|/root]# curl http://localhost/api/service1
Hello World!
I am service1!

[root@docker|/root]# curl http://localhost/api/service2
Hello World!
I am service2!

[root@docker|/root]# echo $PODURL/api/service1
http://docker.podX.example.com/api/service1

[root@docker|/root]# echo $PODURL/api/service2
http://docker.podX.example.com/api/service2
```

Using the above URL, you should be able to access it from your browser by visiting the url:

DNS load balancing

A quick DNS load balancing can be implemented using docker features. We will make use of '--net-alias' service parameter to define or load balanced service name. Make sure that we use use new names for the service (else it will conflict with existing names - or you can delete the older container instances)

Check HTML Guide

```
[root@docker|/root]# docker run -d --net mysrvnet --name newservice1 --net-alias service \
                                -e SERVICE_NAME=newservice1  beh01der/web-service-dockerized-example
b08286794724523e10b59bf373784c02c1b6693eea6f8e5c841921542ca76258



[root@docker|/root]# docker run -d --net mysrvnet --name newservice2 --net-alias service \
                                -e SERVICE_NAME=newservice2  beh01der/web-service-dockerized-example
47b4b445b68d5d7dd83678ee505ac33d40145e96e1554d1c5b1d41dacb88ea23
```

Access the new Load balanced service from outside the network:

Access our new service from DNS Load Balanced endpoints:

```
[root@docker|/root]# curl http://localhost/api/service
Hello World!
I am newservice2!

[root@docker|/root]# curl http://localhost/api/service
Hello World!
I am newservice1!

[root@docker|/root]# curl http://localhost/api/service
Hello World!
I am newservice2!

[root@docker|/root]# echo $PODURL/api/service
http://docker.podX.example.com/api/service
```

Note: You can also access the individual service directly
`http://docker.podX.example.com/api/newservice1`
OR
`http://docker.podX.example.com/api/newservice2`

Few notes on Service discovery

- Docker will not create network names and aliases on the default bridge network, if you want to use those features, you have to create a custom network first.
- Network aliases are not unique on a given network. In that scenario, the Docker DNS server will return multiple records. You will get DNS round robin out of the box.
- Don't rely exclusively on round robin DNS to achieve load balancing
- All traffic may end up going to a single instance or split (unevenly) between some instances
- You have to re-resolve every now and then to discover new endpoints

Docker Volumes

Working with volumes

- Docker volumes can be used to achieve many things, including:
- Bypassing the copy-on-write system to obtain native disk I/O performance.
- Bypassing copy-on-write to leave some files out of docker commit.
- Sharing a directory between multiple containers.
- Sharing a directory between the host and a container.
- Sharing a single file between the host and a container.

Volumes are special directories in a container

Volumes can be declared in two different ways.

Within a Dockerfile, with a VOLUME instruction.

```
VOLUME /uploads
```

On the command-line, with the -v flag for docker run. We saw a use of this earlier.

```
$ docker run -d -v /uploads myapp
```

In both cases, /uploads (inside the container) will be a volume.

How does Volumes help…

- Volumes bypass the copy-on-write system
- Volumes act as pass-through to the host filesystem.
- The I/O performance on a volume is exactly the same as I/O performance on the Docker host.
- When you docker commit, the content of volumes is not brought into the resulting image.
- Files can be shared between the host and the container if required

Managing volumes explicitly

- In some cases, you want a specific directory on the host to be mapped inside the container.
- You want to manage storage and snapshots yourself. (With LVM, or a SAN, or ZFS, or anything else!)
- You have a separate disk with better performance (SSD) or resiliency (EBS) than the system disk, and you want to put important data on that disk.
- You want to share your source directory between your host (where the source gets edited) and the container (where it is compiled or executed).

Let’s see how we can do this.

Start a nginx container and check the access logs (logs are in container)

Start a nginx container: (please delete any old container bound to port :80 before running this)

```
[root@docker|/root]# docker run --name "nginxsrv1"  -d  -p 7000:80  nginx


[root@docker|/root]# docker ps -l
2acf540ffe8a registry.example.com/nginx     "nginx -g 'daemon off"   10 seconds ago      Up 9 seconds
0.0.0.0:7000->80/tcp   nginxsrv1

[root@docker|/root]# docker logs --tail 1 --follow  2e1c1dba9dae
```

Connect to http:// docker.podX.example.com:7000/

```
[root@docker|/root]# docker logs --tail 1 --follow  2e1c1dba9dae
192.168.56.1 - - [16/Dec/2017:11:21:50 +0000] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS
X 10_11_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36" "-”
192.168.56.1 - - [16/Dec/2017:11:22:01 +0000] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS
X 10_11_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36" "-"
```

Start a nginx container and save the access.log outside the container

Start a nginx container with a local directory mounted as the log directory:

```
[root@docker|/root]# mkdir  -p /var/tmp/nginx/logs

[root@docker|/root]# ls -l /var/tmp/nginx/logs

[root@docker|/root]# docker run --name "nginxsrv2"  -d  -p 8000:80  \
                              -v /var/tmp/nginx/logs:/var/log/nginx docker.io/nginx

[root@docker|/root]# ls -l /var/tmp/nginx/logs

[root@docker|/root]# echo $PODURL:8000/
http://docker.podX.example.com:8000/

<<what do you see in the directory?>>
```

Check HTML Guide

Connect to http://<<ip-add-ress-of-host>>:8000/ or http://docker.podX.example.com:8000/

```
[root@docker|/root]# tailf /var/tmp/nginx/logs/access.log
192.168.56.1 - - [16/Dec/2017:11:29:30 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36" "-"
192.168.56.1 - - [16/Dec/2017:11:29:30 +0000] "GET /favicon.ico HTTP/1.1" 404 571 "http://192.168.56.170:8000/" "Mozilla/5.0
(Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36" "-"
```

Sharing Volumes (from host) to containers:

Perform the following steps:

```
[root@docker|/root]# mkdir -p /apps/website/nginx/html

[root@docker|/root]# docker run --name "nginxsrv3"  -d  -p 9000:80  \
                              -v /apps/website/nginx/html:/usr/share/nginx/html nginx

[root@docker|/root]# docker run --name "nginxsrv4"  -d  -p 9001:80  \
                              -v /apps/website/nginx/html:/usr/share/nginx/html nginx

[root@docker|/root]# echo $PODURL:9000/
http://docker.podX.example.com:9000/

[root@docker|/root]# echo $PODURL:9001/
http://docker.podX.example.com:9001/
```

Check HTML Guide

Connect to http://docker.podX.example.com:9000/ and http://docker.podX.example.com:9001/

What do you see? Why are we not able to see the welcome page?

Create a simple index.html file and reload the websites

Perform the following steps:

```
[root@docker|/root]# echo 'Hello from host shared directory!!!'  > /apps/website/nginx/html/index.html
```

Now connect to
http://docker.podX.example.com:9000/ and
http://docker.podX.example.com:9001/

(or refresh the webpages)
What do you see?

```
[root@docker|/root]# echo 'We made some changes just now.....'  > /apps/website/nginx/html/index.html
```

Refresh the webpages. What do you see?

Sharing Volumes across containers

- Volumes can be shared across containers
- You can start a container with exactly the same volumes as another one.
- The new container will have the same volumes, in the same directories.
- They will contain exactly the same thing, and remain in sync.
- Under the hood, they are actually the same directories on the host.
- This is done using the --volumes-from flag for docker run.

Let’s see how we can do this.

Creating an Independent Volume

Perform the following steps:

```
[root@docker|/root]# docker volume create --name DataVolume1
DataVolume1
```

Let's attach it to a container and access it:

```
[root@docker|/root]# docker run -ti --rm -v DataVolume1:/datavolume1 centos

[root@91885a382e5b /]# df -h /datavolume1

[[ what is the size of the mount point ]]

[root@91885a382e5b /]# echo "This is in DataVolume1" > /datavolume1/DataVolume1_file.txt
```

Inspecting the Independent Volume

We can verify the volume is present on the system with docker volume inspect and also check the directory:

```
[root@docker|/root]# docker volume inspect DataVolume1
[
    {
        "Name": "DataVolume1",
        "Driver": "local",
        "Mountpoint": "/var/lib/docker/volumes/DataVolume1/_data",
        "Labels": {},
        "Scope": "local"
    }
]

[root@docker|/root]# ls -l /var/lib/docker/volumes/DataVolume1/_data

-rw-r--r-- 1 root root 23 Dec 16 19:57 DataVolume1_file.txt
```

Note: As seen, we can even look at the data on the host at the path listed as the Mountpoint. **We should avoid altering it as it can cause data corruption if applications or containers are unaware of changes.**

Attaching it to a new container

Perform the following steps:

```
[root@docker|/root]# docker run --rm -ti -v DataVolume1:/datavolume1 centos

[root@91885a382e5b /]# df -h /datavolume1

[root@e9a82889dacb /]# cat /datavolume1/DataVolume1_file.txt
This is in DataVolume1

[root@e9a82889dacb /]# exit
```

Creating an Volume along with the container creation

Perform the following steps:

```
[root@docker|/root]# docker run -ti --name=Container2 -v DataVolume2:/datavolume2 centos
[root@2e6b04782101 /]#
```

Let's attach it to a container and access it:

```
[root@2e6b04782101 /]# echo "This is in DataVolume2" > /datavolume2/DataVolume2_file.txt

[root@2e6b04782101 /]# exit
```

Let us restart the container and check the file.

```
[root@2e6b04782101 /]# cat /datavolume2/DataVolume2_file.txt
This is in DataVolume2

[root@2e6b04782101 /]# exit
```

Removing a Volume when attached to a container

Let us remove **DataVolume2** volume referenced by the container.

```
[root@docker|/root]# docker volume rm DataVolume2
Error response from daemon: Unable to remove volume, volume still in use: remove DataVolume2: volume is in use
- [2e6b04782101fd2a731c113ed53354389f10e9c4c436c9505f57697914d97847]
```

Since the Volume is still in use it will not allow us to delete the volume. If you want to delete the Volume, you will need to first delete the container and then the volume.

```
[root@docker|/root]# docker rm 2e6b04782101fd2a731c113ed53354389f10e9c4c436c9505f57697914d97847
2e6b04782101fd2a731c113ed53354389f10e9c4c436c9505f57697914d97847

[root@docker|/root]# docker volume rm DataVolume2
DataVolume2
```

Creating a Volume from an Existing Directory from a container

If we try to create a volume at the same time as we create a container and we provide the path to a directory that contains data in the base image, that data will be copied into the volume.

```
[root@docker|/root]# docker run -it --rm -v DataVolume3:/var centos
root@c787c02c178f:/#
root@c787c02c178f:/# df -h /var
root@c787c02c178f:/#
root@c787c02c178f:/# echo "I was here"  > /var/tmp/hello.txt
root@c787c02c178f:/# exit
```

Checking the contents of the DataVolume3

Let's launch a new container and attach this Volume and validate our understanding

```
[root@docker|/root]# docker run --rm -it -v DataVolume3:/datavolume3 centos

[root@f9ab5108a9ff /]# ls -l /datavolume3
total 0
drwxr-xr-x  2 root root   6 Apr 12  2016 backups
drwxr-xr-x  5 root root  48 Nov 14 13:48 cache
.
<<snip snip snip snip>>
.
lrwxrwxrwx  1 root root   4 Nov 14 13:48 run -> /run
drwxr-xr-x  2 root root  18 Nov 14 13:48 spool
drwxrwxrwt  2 root root  23 Dec 16 15:46 tmp

[root@f9ab5108a9ff /]# cat /var/tmp/hello.txt            [[why did this give error]]
cat: /var/tmp/hello.txt: No such file or directory

[root@f9ab5108a9ff /]# cat /datavolume3/tmp/hello.txt
I was here

[root@f9ab5108a9ff /]# exit
exit
```

Sharing Data between Multiple Docker Containers

Create a new container named Container4 with a data volume attached.

```
[root@docker|/root]# docker run -it --name=Container4 -v DataVolume4:/datavolume4 centos

[root@f4c244ab673a /]# echo "This file is shared between containers" > /datavolume4/DataVolume4_file.txt

<<issue Ctrl-P + Ctrl-Q to exit this session>>
```

Create Container5 and Mount Volumes from Container4

```
[root@docker|/root]# docker run -it --name=Container5 --volumes-from Container4 centos

[root@0665be39dad5 /]# cat /datavolume4/DataVolume4_file.txt
This file is shared between containers

[root@0665be39dad5 /]# echo "Even I can write to this Volume"  > /datavolume4/FromContainer5_file.txt

<<issue Ctrl-P + Ctrl-Q to exit this session  and you can validate if this appeared in Container4>>
```

Let’s just validate if the file can be see in Container4

Attach to Container4

```
[root@docker|/root]# docker attach Container4

[root@f4c244ab673a /]# cat /datavolume4/FromContainer5_file.txt
Even I can write to this Volume

<<issue Ctrl-P + Ctrl-Q to exit this session>>
```

We can see that both containers were able to read and write from the data volume

Mounting a volume as Read-Only

We can mount a Volume as a Read-Only volume. Start a new container and test it out:

```
[root@docker|/root]# docker run -ti --name=Container6 --volumes-from Container4:ro centos

[root@36612a2f2b9f /]# cat /datavolume4/DataVolume4_file.txt
This file is shared between containers

[root@36612a2f2b9f /]# cat /datavolume4/FromContainer5_file.txt
Even I can write to this Volume

[root@36612a2f2b9f /]# rm /datavolume4/DataVolume4_file.txt
rm: cannot remove '/datavolume4/DataVolume4_file.txt': Read-only file system

[root@36612a2f2b9f /]# echo Hello >> /datavolume4/FromContainer5_file.txt
bash: /datavolume4/FromContainer5_file.txt: Read-only file system
```

We can easily access the contents of DataVolume4; however since it is mounted as Read-Only, there is no way Container6 can make any changes to DataVolume4.

Can we mount multiple Volumes on a container? Yes!!!

We are almost done with volumes; we just want to let everyone know that we can indeed attach multiple volumes on a container and with some volumes in RO mode

Launch Container7 and try that:

```
[root@docker|/root]# docker volume create --name DataVolume5
[root@docker|/root]# docker volume create --name DataVolume6
[root@docker|/root]# docker volume create --name DataVolume7

[root@docker|/root]# docker run -ti --name=Container7 --volumes-from Container4:ro -v DataVolume5:/datavolume5
-v DataVolume6:/datavolume6  -v DataVolume7:/datavolume7  centos

[root@docker|/root]# docker volume ls
```

Check HTML Guide

Finally a quick look at the Volumes lifecycle

- When you remove a container, its volumes are kept around.
- You can list them with **`docker volume ls`**.
- You can access them by creating a container with **`docker run -v`**.
- You can remove them with **`docker volume rm`**.
- You can always use **`docker inspect`** on a volume or container to get more details.
- As usual, you are the one responsible for logging, monitoring, and backup of your volumes.

Container Summary

- Sandboxed application processes on a shared Linux OS Kernel
- Packages application and all its dependencies together
- Simpler, lighter, and denser than virtual machines
- Portable across different environments
- Deploy to any environment in seconds and enable CI/CD
- Needs orchestration and management to be useful
- Easily access and share containerized components

Docker – Is this all or is it just the beginning…

Docker Ecosystem

- Infrastructure
  - docker machine (provisioning)
  - docker swarm (clustering)
  - swarm mode (clustering)
  - infrakit (automated self-healing infrastructure monitoring and provisioning)
- Container deployment & configuration
  - docker compose
- Image distribution
  - docker distribution (registry)
  - docker notary (content trust, image signing)