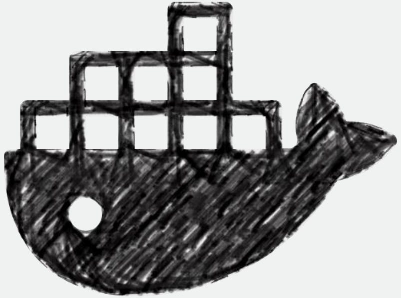
Docker for java developers

Day 1

- Dmitry Buhtiyarov, Siarhei Beliakou, 2018



Agenda

Containers Overview:

- ➢ Docker Components
- ➢ Docker Architecture
- ➢ Docker Storage Drivers

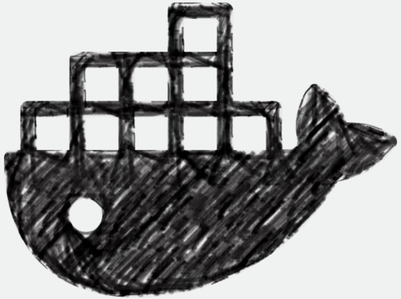
Installation and Configuration

Creating Docker Images

Running Containers

Getting Logs From Containers

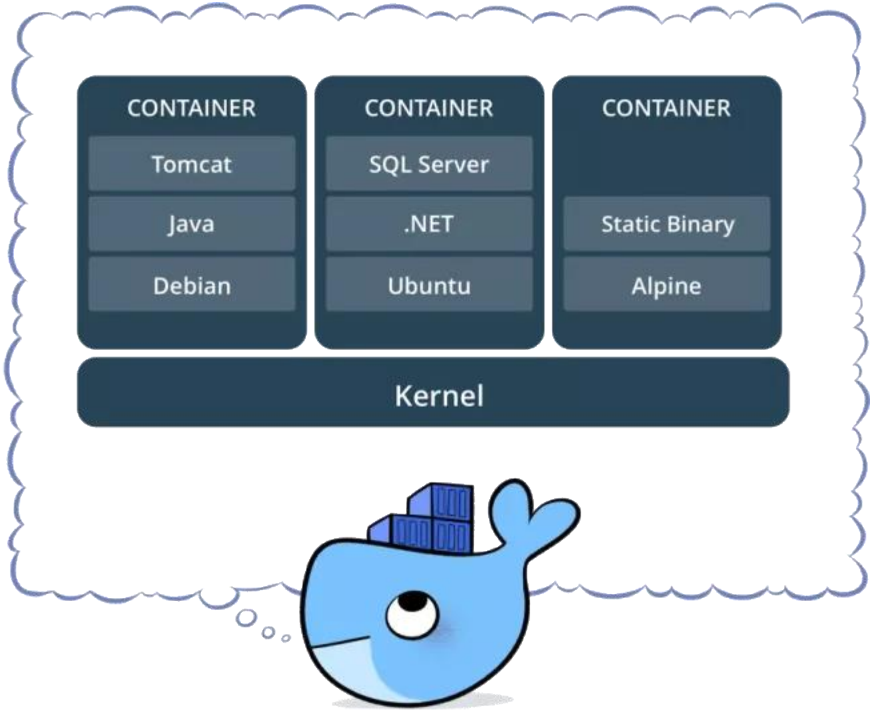
Demo



CONTAINERS OVERVIEW

- What Containers and Images are about
- Virtual Machines vs Containers
- Containers History
- Containers Underlying Technologies
- Docker Architecture
- Storage Drivers

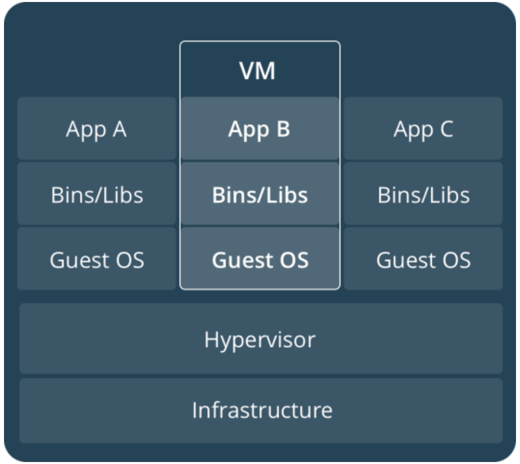
Virtual Machines vs Containers

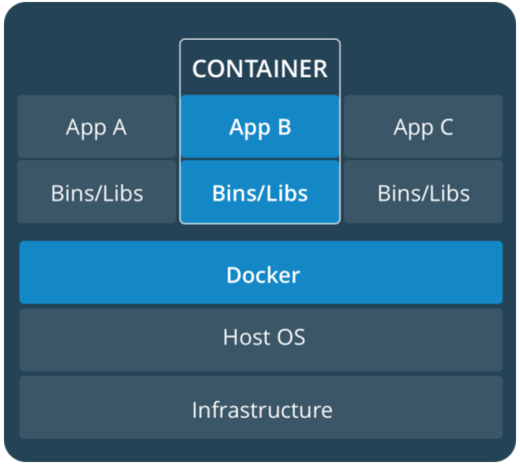


A **container** is something quite similar to a virtual machine, which can be used to **contain and execute** all the software required to run a particular program or set of programs.

The container includes an operating system (typically some flavor of Linux) as base, plus any software installed on top of the OS that might be needed. This container can therefore be run as a self-contained virtual environment, which makes it a lot easier to reproduce the same analysis on any infrastructure that supports running the container, from your laptop to a cloud platform, without having to go through the pain of identifying and installing all the software dependencies involved. You can even have multiple containers running on the same machine, so you can easily switch between different environments if you need to run programs that have incompatible system requirements.

Virtual Machines vs Containers





VIRTUAL MACHINES

Virtual machines (VMs) are an abstraction (emulation) of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, one or more apps, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.

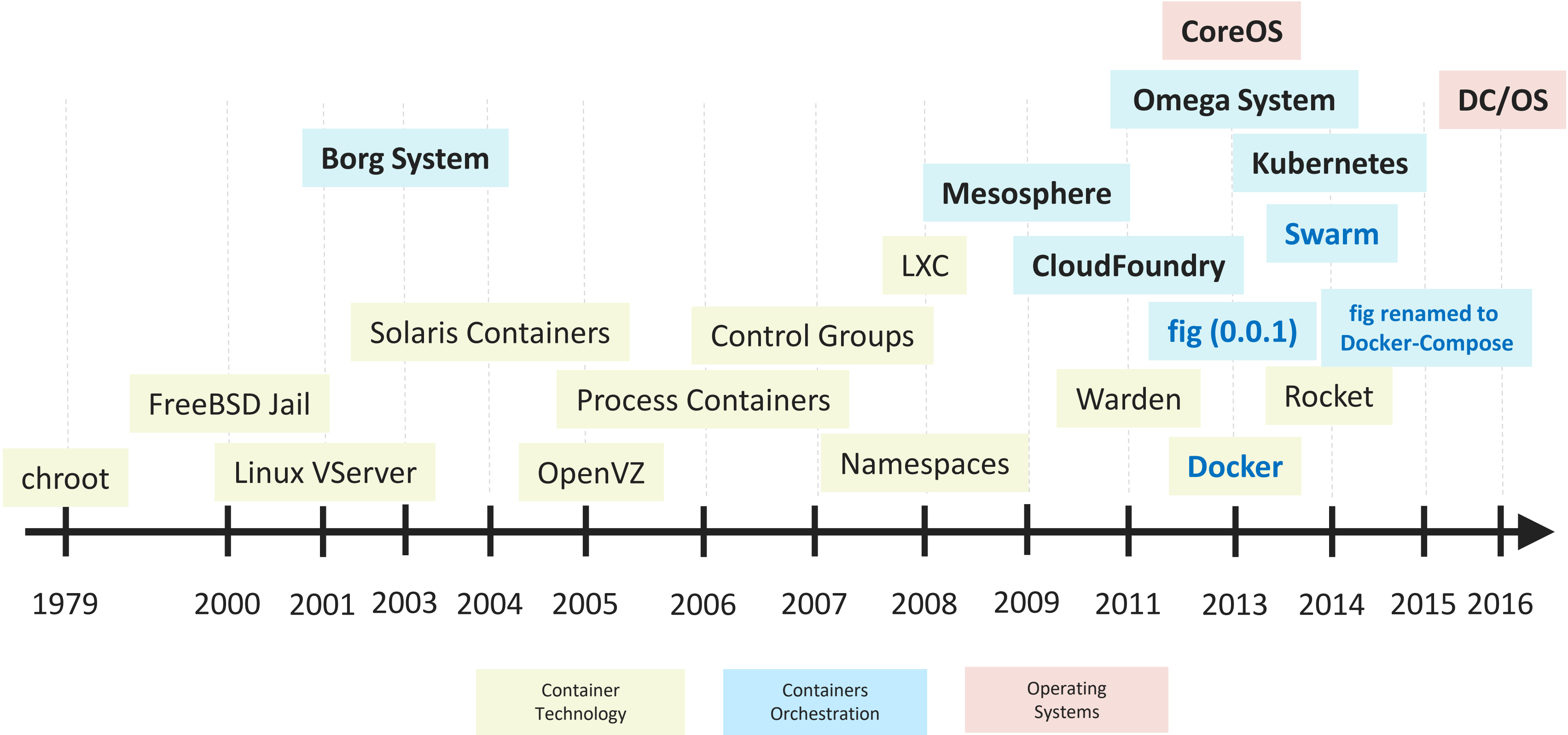
CONTAINERS

Containers are an abstraction at the app layer that packages code and dependencies together (worker process). Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), and start almost instantly.

Virtual Machines vs Containers

	Container	Hypervisor
Redundancy	Only Application Code and Data Edited Files	Computation and Memory redundancy
Flexibility	Less Flexible, Linux can host linux based containers, Windows - windows based containers, MacOS uses Linux VM to run contianers	More Flexible where Linux OS can host windows and vice verse
Security	Less Secure, DOS attacks can affect others Containers Container with root privileges could affect other Containers	Full Isolation, Hypervisor is responsible of limiting used resource by VM Cross Guest Access is prohibited
Creating Time	Almost instantaneous	Even Preexisting VMs need OS Load Time
Performance	Almost Native	Less than native due to middle ware
Consolidation	Limited by actual Application Usage	Limited By OS reserved
Memory	Allocation Memory	Allocated Files Disk, Allocated Files

Containers History



CoreOS

Omega System

DC/OS

Borg System

Kubernetes

Mesosphere

Swarm

LXC

CloudFoundry

fig renamed to Docker-Compose

Solaris Containers

Control Groups

fig (0.0.1)

FreeBSD Jail

Process Containers

Warden

Rocket

chroot

Linux VServer

OpenVZ

Namespaces

Docker

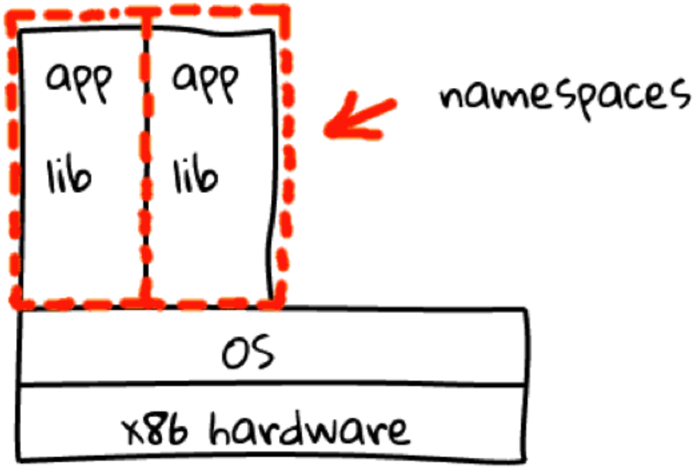
1979 2000 2001 2003 2004 2005 2006 2007 2008 2009 2011 2013 2014 2015 2016

Container Technology

Containers Orchestration

Operating Systems

How OS Manages Containers



Namespaces

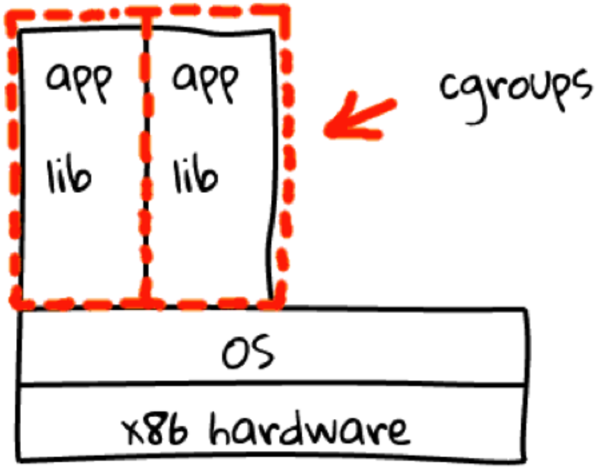
When you run a container, Docker creates a set of *namespaces* for that container.
This provides a layer of isolation: each aspect of a container runs in its own namespace and does not have access outside of it.

Some of the namespaces that Docker Engine uses on Linux are:

- ✓ The **pid** namespace: Process isolation (PID: Process ID).
- ✓ The **net** namespace: Managing network interfaces (NET: Networking).
- ✓ The **ipc** namespace: Managing access to IPC resources (IPC: InterProcess Communication).
- ✓ The **mnt** namespace: Managing mount-points (MNT: Mount).
- ✓ The **uts** namespace: Isolating kernel and version identifiers. (UTS: Unix Timesharing System).

Manpages

How OS Manages Containers

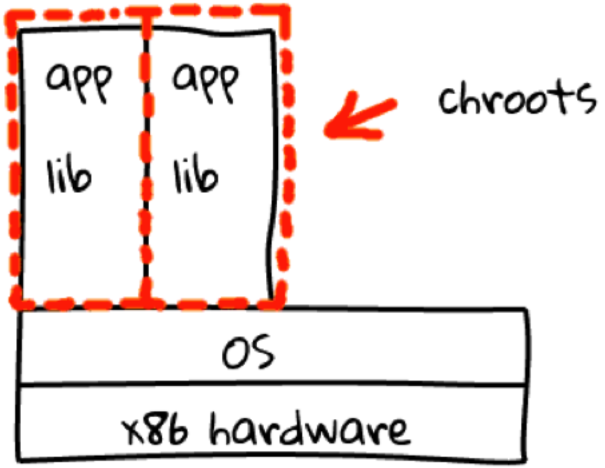


Control Groups

A key to running applications in isolation is to have them only use the resources you want. This ensures containers are good multi-tenant citizens on a host. Control groups allow Docker Engine to share available hardware resources to containers and, if required, set up limits and constraints. For example, limiting the memory available to a specific container.

- ✓ network.
- ✓ block i/o
- ✓ CPU
- ✓ Memory

How OS Manages Containers



chroot

choot is simply isolation on the filesystem

Union FS

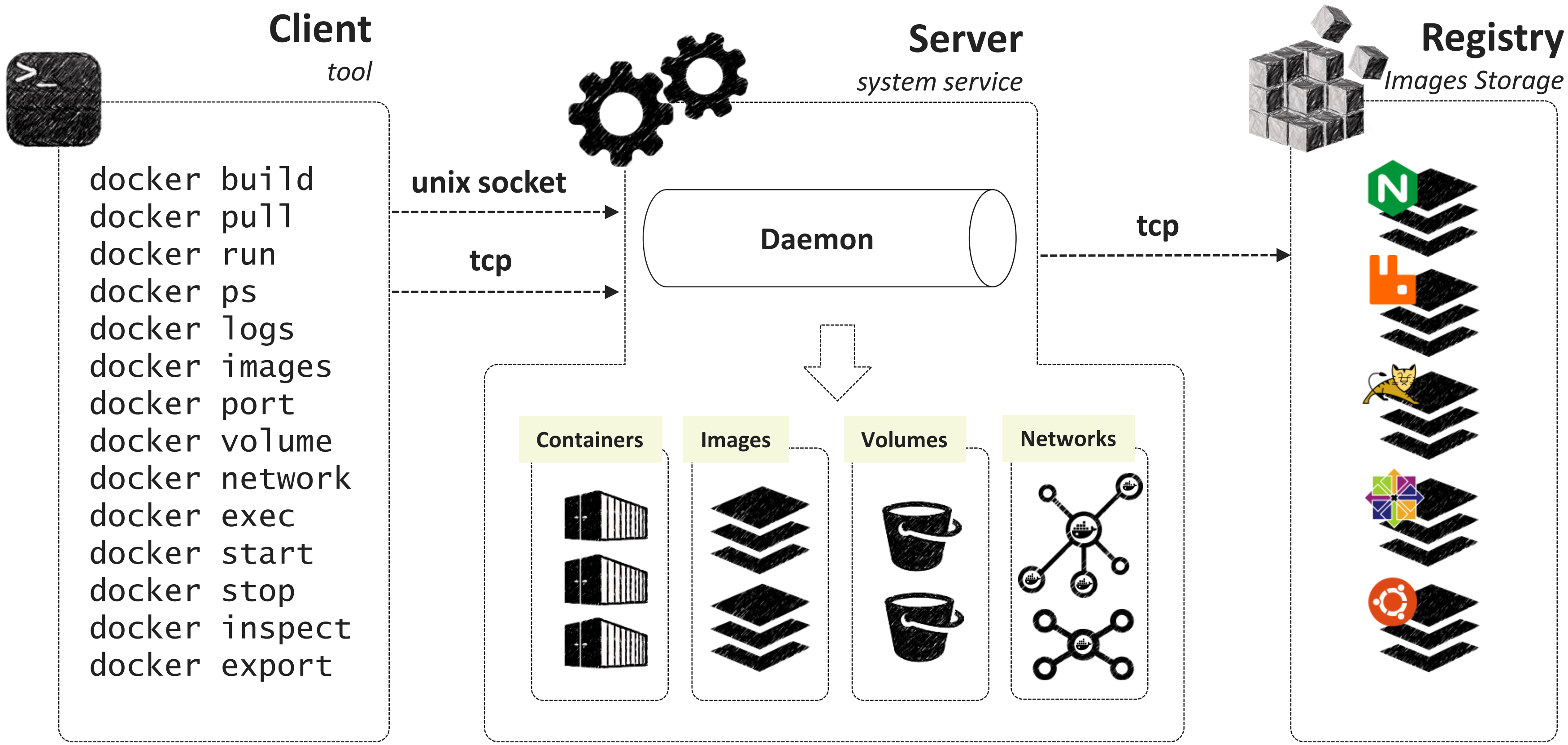
operate by creating layers, making them very lightweight and fast. Docker Engine uses union file systems to provide the building blocks for containers. Docker Engine can make use of several union file system variants including: AUFS, btrfs, vfs, and DeviceMapper

12factor

https://github.com/docker/labs/tree/master/12factor

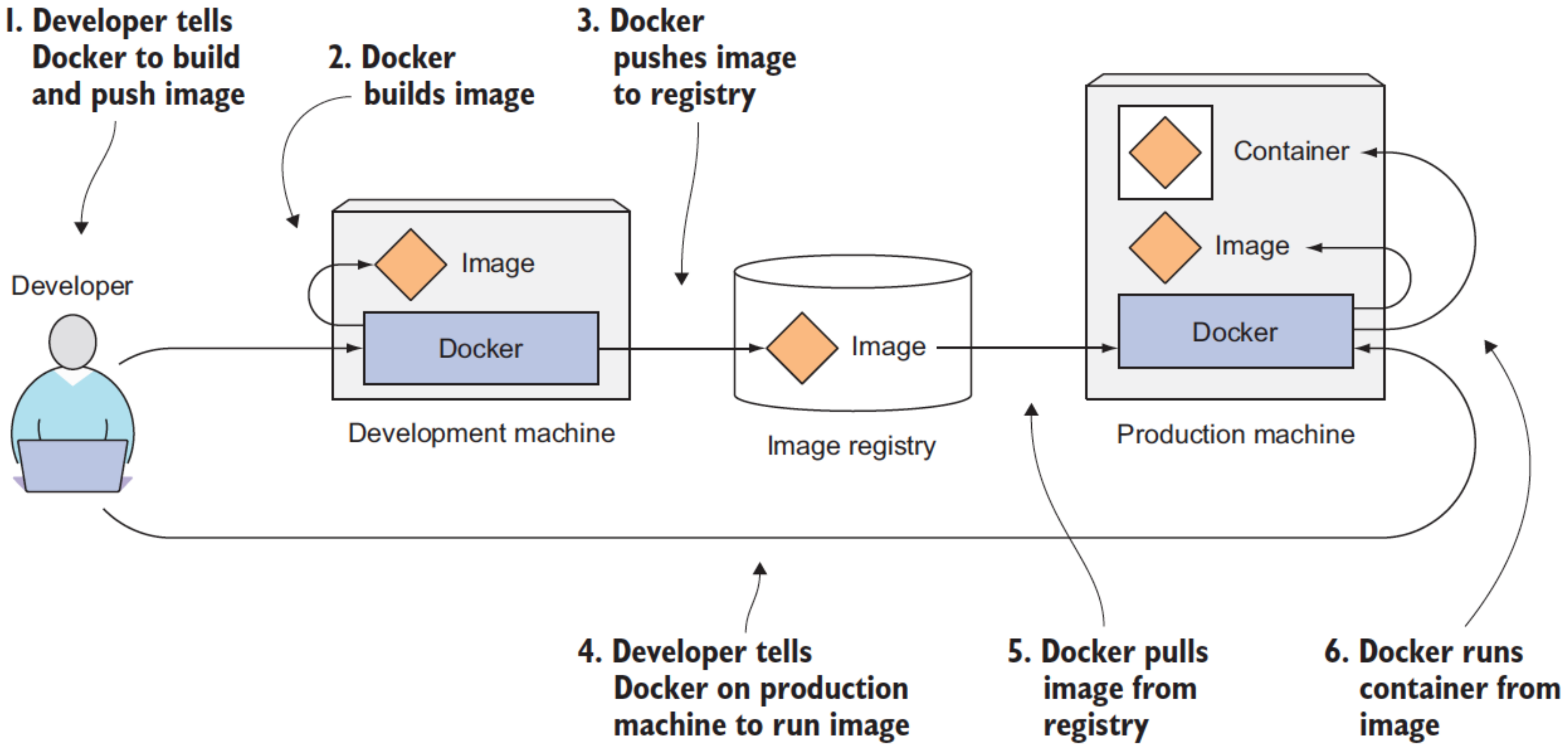
A great knowledge of what should be done to get cloud native application.
12 factor methodology is the result of their observations. As the name states, it presents 12 principles that will help application to be cloud ready, horizontally scalable, and portable.

Docker Architecture



```
docker build
docker pull
docker run
docker ps
docker logs
docker images
docker port
docker volume
docker network
docker exec
docker start
docker stop
docker inspect
docker export
```

Docker Workflow



1. Developer tells Docker to build and push image

2. Docker builds image

3. Docker pushes image to registry

4. Developer tells Docker on production machine to run image

5. Docker pulls image from registry

6. Docker runs container from image

Docker Components

The Docker daemon (dockerd)

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

The Docker client (docker)

is a tool to communicate with Docker daemon.

Docker images

is a read-only template with instructions for creating a Docker container. Often, an image is *based on* another image, with some additional customization

Docker containers

is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI

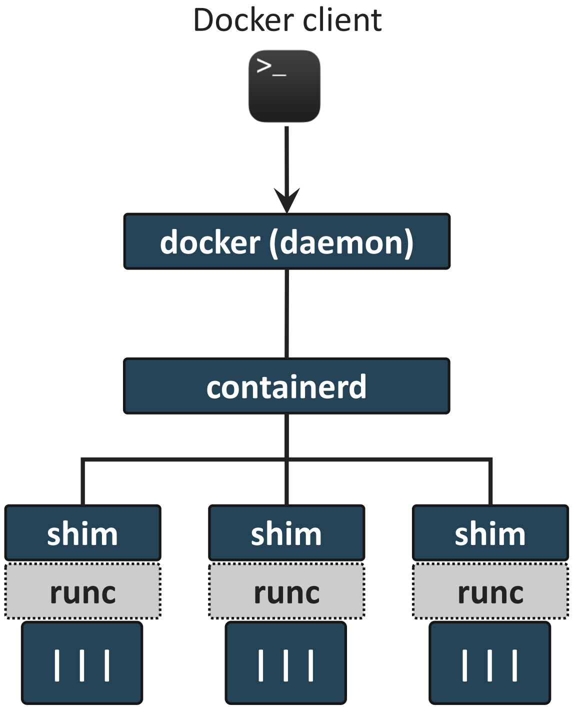
Docker registries

stores Docker images

Docker Services

allow you to scale containers across multiple Docker daemons, which all work together as a *swarm* with multiple *managers* and *workers*

Docker Architecture



(/usr/bin/)dockerd: The Docker daemon itself. The highest level component in your list and also the only 'Docker' product listed. Provides all the nice UX features of Docker.

(/usr/bin/)docker-containerd: Also a daemon, listening on a Unix socket, exposes gRPC endpoints. Handles all the low-level container management tasks, storage, image distribution, network attachment, etc...

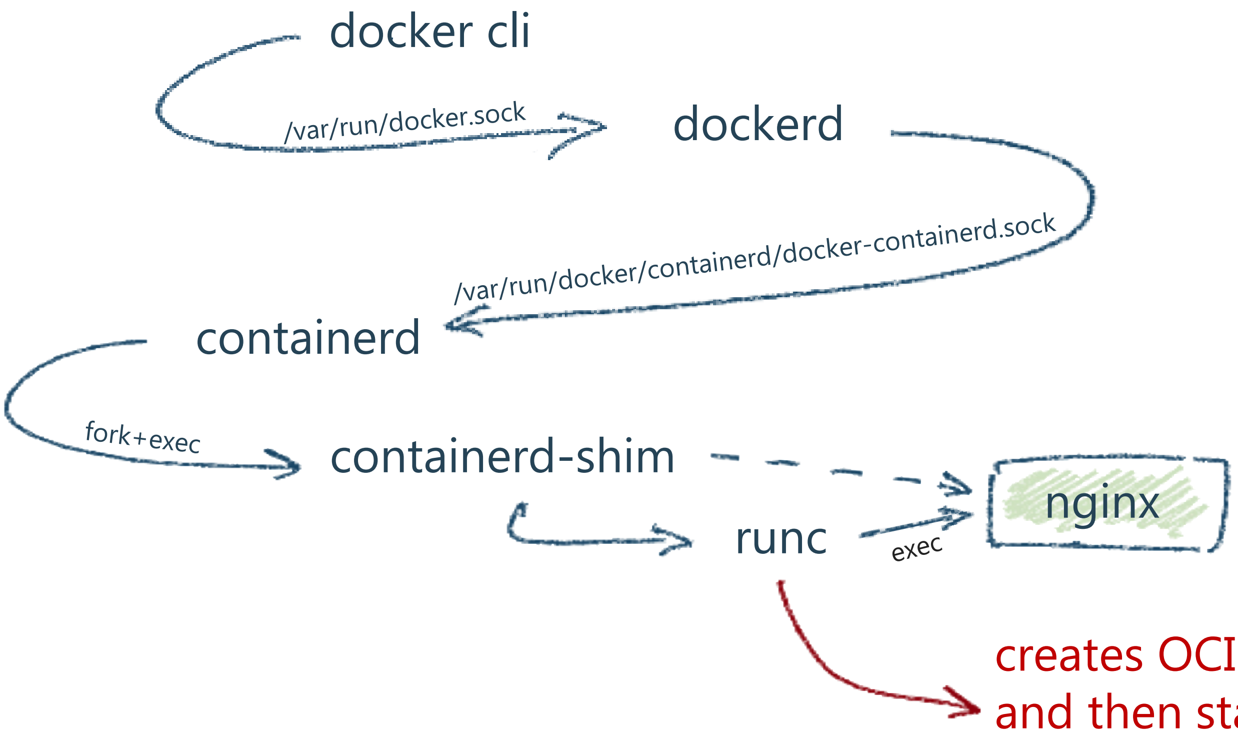
(/usr/bin/)docker-containerd-ctr: A lightweight CLI to directly communicate with containerd. Think of it as how 'docker' is to 'dockerd'.

(/usr/bin/)docker-containerd-shim: After runC actually runs the container, it exits (allowing us to not have any long-running processes responsible for our containers). The shim is the component which sits between containerd and runc to facilitate this.

(/usr/bin/)docker-runc: A lightweight binary for actually running containers. Deals with the low-level interfacing with Linux capabilities like cgroups, namespaces, etc...

(/usr/bin/)docker-proxy: A tool responsible for proxying container's ports to Host's interface

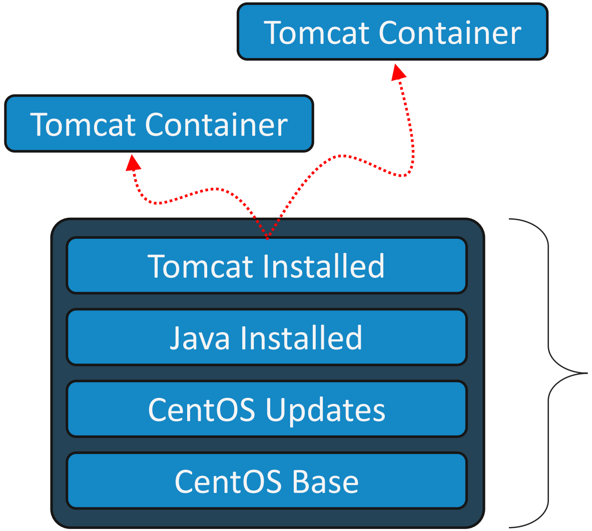
How Docker Spawns Containers



creates OCI bundle
and then starts,
leaving only NGINX

What is Image?

An image is an inert, immutable, file that's essentially a snapshot of a container. Images are created with the build command, and they'll produce a container when started with run. Images are stored in a Docker registry such as registry.hub.docker.com. Because they can become quite large, images are designed to be composed of layers of other images, allowing a minimal amount of data to be sent when transferring images over the network



Docker Container

Docker Container

Docker Image (layers)

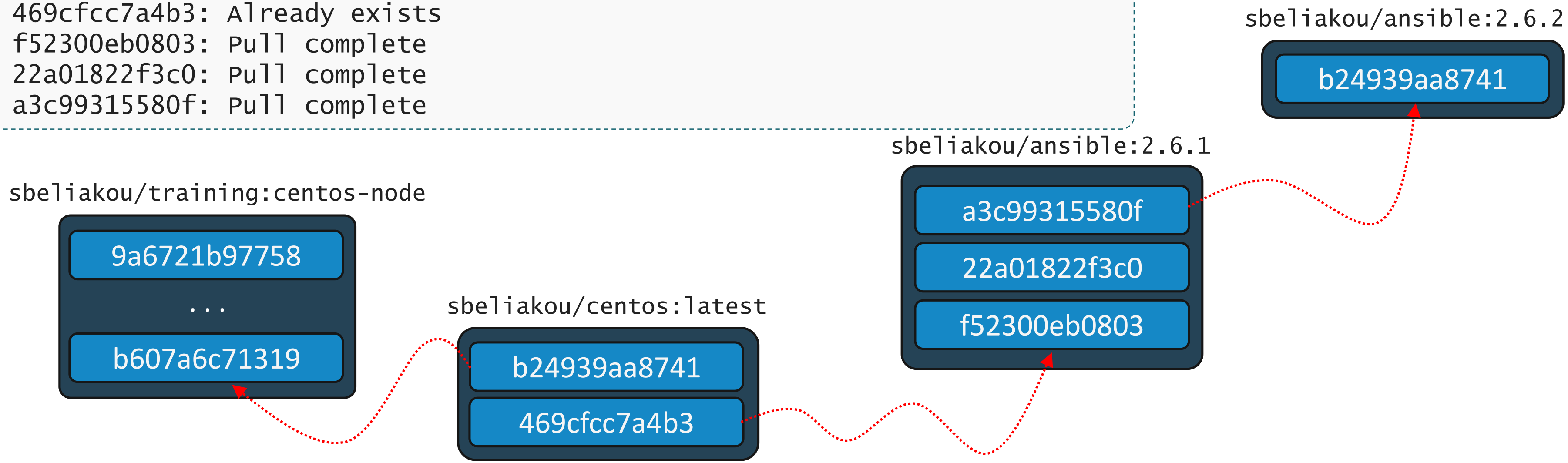
The copy-on-write (CoW) strategy

is a strategy of sharing and copying files for maximum efficiency. If a file or directory exists in a lower layer within the image, and another layer (including the writable layer) needs read access to it, it just uses the existing file. The first time another layer needs to modify the file (when building the image or running the container), the file is copied into that layer and modified. This minimizes I/O and the size of each of the subsequent layers. These advantages are explained in more depth below

What is Image?

```
[vagrant@docker-host ~]$ docker pull sbeliakou/centos
Using default tag: latest
latest: Pulling from sbeliakou/centos
469cfcc7a4b3: Pull complete
b24939aa8741: Pull complete

[vagrant@docker-host ~]$ docker pull sbeliakou/ansible:2.6.1
2.6.1: Pulling from sbeliakou/ansible
469cfcc7a4b3: Already exists
f52300eb0803: Pull complete
22a01822f3c0: Pull complete
a3c99315580f: Pull complete
```



Docker installation and Configuration

- Installing Docker on CentOS
- How Docker Works

Installing Docker Service on CentOS

VAGRANT

```
Vagrant.configure("2") do |config|
  config.vm.box = "sbeliakou/centos"
  config.vm.box_version = "7.5"
  config.vm.network :private_network, ip: "192.168.56.15"

  config.vm.provision "shell", inline: <<-SHELL
    yum install -y yum-utils jq net-tools
    yum-config-manager --add-repo \
      https://download.docker.com/linux/centos/docker-ce.repo
    yum-config-manager --enable docker-ce-edge

    yum install -y docker-ce
    systemctl enable docker
    systemctl start docker
    usermod -aG docker vagrant
  SHELL
end
```

```
$ vagrant up
$ vagrant ssh
$ vagrant halt
$ vagrant destroy
```

Installing Docker Service

Add application user to docker group

The docker daemon binds to a Unix socket instead of a TCP port. By default that Unix socket is owned by the user root and other users can access it with sudo. For this reason, docker daemon always runs as the root user.

To avoid having to use sudo when you use the docker command, create a Unix group called docker and add users to it. When the docker daemon starts, it makes the ownership of the Unix socket read/writable by the docker group

```
[root@localhost ~]# usermod -aG docker vagrant
```

Try Something Simple

```
[vagrant@localhost vagrant]$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
9db2ca6ccae0: Pull complete
Digest: sha256:4b8ff392a12ed9ea17784bd3c9a8b1fa3299cac44aca35a85c90c5e3c7afacdc
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

How It Works

Either by using the docker binary or via the API, the Docker client tells the Docker daemon to run a container.

```
[vagrant@localhost ~]$ docker run -i -t ubuntu bash
```

The Docker Engine client is launched using the ***docker*** tool with the ***run*** option running a new container.

The bare minimum the *docker client* needs to tell the Docker daemon to run the container is:

- ✓ What Docker image to build/create the container from, for example, ubuntu
- ✓ The command you want to run inside the container when it is launched, for example,/bin/bash

So what happens under the hood when we run this command?
In order, Docker Engine does the following:

- ✓ Pulls the ***ubuntu*** image.
- ✓ Creates a new container.
- ✓ Allocates a filesystem and mounts a read-write *layer*.
- ✓ Allocates a network / bridge interface.
- ✓ Sets up an IP address.
- ✓ Executes a process that you specify.
- ✓ Captures and provides application output.

You now have a running container! Now you can manage your container, interact with your application and then, when finished, stop and remove your container.

Docker Daemon, Hub/Registry Commands

Display Docker Version and Info

- docker --version
- docker version
- docker version --format '{{.Server.Version}}'
- docker version --format '{{json .}}'
- docker info
- docker info --format '{{json .}}'

Working with Docker Hub/Registry:

- docker login - to login to a registry.
- docker logout - to logout from a registry.
- docker search - searches registry for image.
- docker pull - pulls an image from registry to local machine.
- docker push - pushes an image to the registry from local machine.

Inspecting Docker Configuration

```
[vagrant@localhost ~]$ docker info
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 0
Server Version: 18.06.0-ce
Storage Driver: overlay2
 Backing Filesystem: xfs
Logging Driver: json-file
Cgroup Driver: cgroupfs
Kernel Version: 3.10.0-862.9.1.el7.x86_64
Operating System: CentOS Linux 7 (Core)
Docker Root Dir: /var/lib/docker
Registry: https://index.docker.io/v1/
```

Creating docker images

- Build own Image with Dockerfile
- Tagging Images
- CMD/Entrypoint
- Build Arguments
- Multistage Build

Working with Images

Lifecycle:

- docker images - shows all images.
- docker import - creates an image from a tarball.
- docker build - creates image from Dockerfile.
- docker commit - creates image from a container, pausing it temporarily if it is running.
- docker rmi - removes an image.
- docker load - loads an image from a tar archive as STDIN, including images and tags.
- docker save - saves an image to a tar archive stream to STDOUT with all parent layers, tags & versions.

Info:

- docker history - shows history of image.
- docker tag - tags an image to a name (local or registry).

Dockerfile

Dockerfile

```
FROM centos
LABEL maintainer="Siarhei Beliakou"
RUN yum install -y httpd web-assets-httpd && \
    yum clean all
RUN echo "my httpd container" > /var/www/html/index.html
# ADD/COPY index.html /var/www/html/
EXPOSE 80
ENTRYPOINT ["httpd"]
CMD ["-DFOREGROUND"]
```

```
# docker build [-t image_tag] [-f ./path/to/dockerfile] .
...
Status: Downloaded newer image for centos:latest
 ---> 49f7960eb7e4
...
Step 4/6 : RUN echo "my httpd container" > /var/www/html/index.html
 ---> Running in 37cc17740d0b
...
Successfully built 50a986f614d5
```

Dockerfile Instructions

- .dockerignore
- FROM - Sets the Base Image for subsequent instructions.
- RUN - execute any commands in a new layer on top of the current image and commit the results.
- CMD - provide defaults for an executing container.
- EXPOSE - informs Docker that the container listens on the specified network ports at runtime. NOTE: does not actually make ports accessible.
- ENV - sets environment variable.
- ADD - copies new files, directories or remote file to container. Invalidates caches.
- COPY - copies new files or directories to container.
- ENTRYPOINT - configures a container that will run as an executable.
- VOLUME - creates a mount point for externally mounted volumes or other containers.
- USER - sets the user name for following RUN / CMD / ENTRYPOINT commands.
- WORKDIR - sets the working directory.
- ARG - defines a build-time variable.
- ONBUILD - adds a trigger instruction when the image is used as the base for another build.
- STOPSIGNAL - sets the system call signal that will be sent to the container to exit.
- LABEL - apply key/value metadata to your images, containers, or daemons.

Tagging Images

```
$ docker build -t myhttpd .
...
Successfully built 50a986f614d5
Successfully tagged myhttpd:latest
```

```
$ docker tag 50a986f614d5 sbeliakou/myhttpd
```

```
$ docker tag myhttpd sbeliakou/myhttpd:1.0
```

```
$ docker tag myhttpd:latest sbeliakou/myhttpd:latest
```

CMD/ENTRYPOINT

```
FROM busybox
ENTRYPOINT ["echo", "hello"]
```

```
# docker build -t cmd_entrypoint:0.1 .
```

```
# docker run cmd_entrypoint:0.1
hello
```

```
# docker run cmd_entrypoint:0.1 world
hello world
```

```
FROM busybox
ENTRYPOINT echo hello
```

```
# docker build -t cmd_entrypoint:0.2 .
```

```
# docker run cmd_entrypoint:0.2
hello
```

```
# docker run cmd_entrypoint:0.2 world
hello
```

```
FROM busybox
CMD ["echo", "hello"]
```

```
# docker build -t cmd_entrypoint:0.3 .
```

```
# docker run cmd_entrypoint:0.3
hello
```

```
# docker run cmd_entrypoint:0.3 world
docker: Error response from daemon: OCI runtime create
failed: container_linux.go:348: starting container
process caused "exec: \"world\": executable file not
found in $PATH": unknown.
```

```
FROM busybox
CMD echo hello
```

```
# docker build -t cmd_entrypoint:0.4 .
```

```
# docker run cmd_entrypoint:0.4
hello
```

```
# docker run cmd_entrypoint:0.4 world
docker: Error response from daemon: OCI runtime create
failed: container_linux.go:348: starting container
process caused "exec: \"world\": executable file not
found in $PATH": unknown.
```

CMD/ENTRYPOINT

```
FROM busybox
ENTRYPOINT echo
CMD hello
```

```
# docker build -t cmd_entrypoint:0.5 .
```

```
# docker run cmd_entrypoint:0.5
```

```
# docker run cmd_entrypoint:0.5 world
```

```
FROM busybox
ENTRYPOINT ["echo"]
CMD hello
```

```
# docker build -t cmd_entrypoint:0.6 .
```

```
# docker run cmd_entrypoint:0.6
/bin/sh -c hello
```

```
# docker run cmd_entrypoint:0.6 world
world
```

```
FROM busybox
ENTRYPOINT echo
CMD ["hello"]
```

```
# docker build -t cmd_entrypoint:0.7 .
```

```
# docker run cmd_entrypoint:0.7
```

```
# docker run cmd_entrypoint:0.7 world
```

```
FROM busybox
ENTRYPOINT ["echo"]
CMD ["hello"]
```

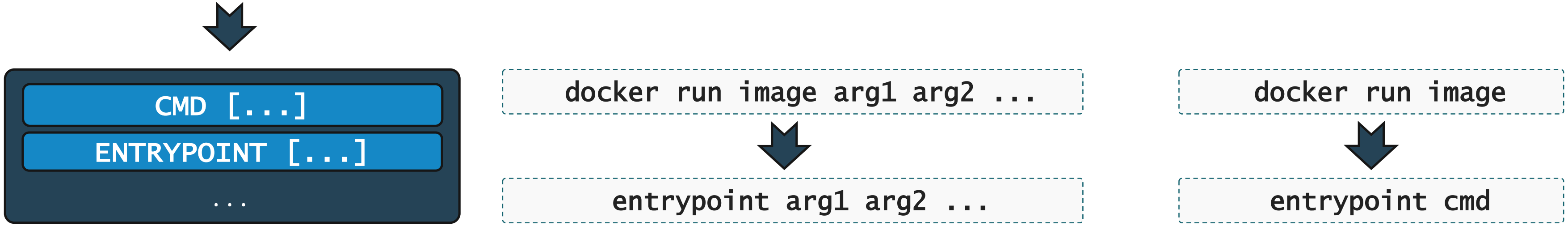
```
# docker build -t cmd_entrypoint:0.8 .
```

```
# docker run cmd_entrypoint:0.8
hello
```

```
# docker run cmd_entrypoint:0.8 world
world
```

CMD/ENTRYPOINT. The Best Choice

```
...
ENTRYPOINT ["/command/to/be/executed"]
CMD ["default", "args"]
```



CMD [...]

ENTRYPOINT [...]

...

```
docker run image arg1 arg2 ...
```

```
entrypoint arg1 arg2 ...
```

```
docker run image
```

```
entrypoint cmd
```

Dockerfile Example: Custom Packer Image

```
FROM centos:7
ENV PACKER_VERSION 1.2.3
RUN yum install -y \
      epel-release \
      yum-plugin-ovl \
      wget unzip \
      rsync \
      openssh openssh-clients && \
    yum install -y python-pip && \
    yum clean all

RUN wget -q https://releases.hashicorp.com/...${PACKER_VERSION}_linux_amd64.zip && \
    unzip -q packer_${PACKER_VERSION}_linux_amd64.zip -d /bin/ && \
    rm -f packer_${PACKER_VERSION}_linux_amd64.zip
RUN pip install -U ansible ansible-modules-hashivault

RUN useradd packer

USER packer
ENV USER packer

ENTRYPOINT ["/bin/packer"]
CMD ["--version"]
```

Some More Examples

```
FROM java:8-jre

ADD customer-contact-service.jar /
EXPOSE 4040

CMD ["java", "-jar", "/customer-contact-service.jar"]
```

```
FROM java:8

ENV PORT=8080
EXPOSE 8080
COPY wiremock /wiremock

CMD ["/wiremock/bin/startServer.sh"]
```

Docker Base Images

- `scratch` – this is the ultimate base image and it has 0 files and 0 size.
- `busybox` – a minimal Unix weighing in at 2.5 MB and around 10000 files.
- `debian:jessie` – the latest Debian is 122 MB and around 18000 files.
- `alpine:latest` – Alpine Linux, only 8 MB in size and has access to a package repository

```
FROM scratch

ADD   centos.tar.gz /

RUN   yum install -y epel-release && \
      yum update -y && \
      yum clean all

LABEL architecture="amd64" \
      OS="CentOS" \
      License=GPLv2 \
      maintainer="Siarhei Beliakou (sbeliakou@gmail.com)"

## Default command
CMD ["/bin/bash"]
```

More details

Useful Links and Examples

Many Samples: docs.docker.com/samples/
Examples: #dockerfile-examples
Jenkins: Dockerfile-alpine
Tomcat: jre8-alpine/Dockerfile
OpenJDK: jdk/alpine/Dockerfile
AmazonLinux: 2018.03/Dockerfile

Another example is here:

#define-a-container-with-dockerfile

Dockerfile Reference:

#/dockerfile-reference

Best practices for writing Dockerfiles:

docs.docker.com/dockerfile_best-practices/

Docker CLI Reference:

commandline/cli/

Build With Arguments

```
ARG BASE_IMAGE
FROM ${BASE_IMAGE}
...
```

```
# docker build --build-arg BASE_IMAGE=hashicorp/terraform:0.10.8.
...
```

```
# docker build --build-arg BASE_IMAGE=hashicorp/terraform:0.11.7.
...
```

```
...
ARG BUILD_ID
LABEL build_id="${BUILD_ID}"
...
```

Multistage Build

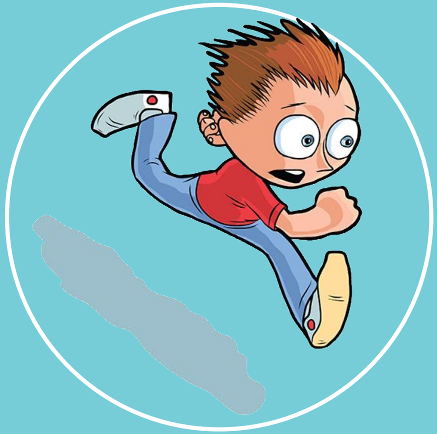
```
FROM maven:3.3-jdk-8 as builder
COPY . /build/
WORKDIR /build
RUN mvn clean install

FROM openjdk:8-jre
COPY --from=builder /build/target/demoapp.jar /opt/
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/opt/demoapp.jar"]
```

```
# docker build -t myapp .
...
```

```
# docker build -t myapp --target builder .
...
```

Explore example here: sbeliakou/springboot_example



RUNNING CONTAINERS

- Running in detached mode
- Exposing Ports
- Managing Restart Policy
- Changing Workspace Directory
- Changing Runtime User
- Providing Environment Variables

Working with Containers

Lifecycle:

- docker create - creates a container but does not start it.
- docker rename - allows the container to be renamed.
- docker run - creates and starts a container in one operation.
- docker rm - deletes a container.
- docker update - updates a container's resource limits.

Starting and Stopping:

- docker start - starts a container so it is running.
- docker stop - stops a running container.
- docker restart - stops and starts a container.
- docker pause - pauses a running container, "freezing" it in place.
- docker unpause - will unpause a running container.
- docker wait - blocks until running container stops.
- docker kill - sends a SIGKILL to a running container.
- docker attach - will connect to a running container.

Working with Containers

Info:

- docker ps - shows running containers.
- docker logs - gets logs from container.
- docker inspect - looks at all the info on a container.
- docker events - gets events from container.
- docker port - shows public facing port of container.
- docker top - shows running processes in container.
- docker stats - shows containers' resource usage statistics.
- docker diff - shows changed files in the container's FS.

Import / Export:

- docker cp - copies files or folders between a container and the local filesystem.
- docker export - turns container filesystem into tarball archive stream to STDOUT.

Executing Commands:

- docker exec - to execute a command in container.

Running the Container

```
# docker run 50a986f614d5
^C
```

```
# docker run -d myhttpd:1.0
9f761335efe268e9a82c4828d8f4be67b5824eb3266e8ba311343a7da45c67ff
```

```
# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS                    NAMES
9f761335efe2   50a986f614d5   "/bin/sh -c 'httpd -…"   5 seconds ago    Up 4 seconds    80/tcp                   trusting_kilby
```

```
# docker run -P -d myhttpd:1.0
74954ff14ec5e53ac9925bfd2873c654fe8978657764b4162ac494fc9afaab9f
```

```
# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS                    NAMES
74954ff14ec5   myhttpd:1.0    "/bin/sh -c 'httpd -…"   9 seconds ago    Up 18 seconds   0.0.0.0:32768->80/tcp    fervent_noyce
9f761335efe2   myhttpd:1.0    "/bin/sh -c 'httpd -…"   3 minutes ago    Up 3 minutes    80/tcp                   trusting_kilby
```

```
# curl localhost:32768
my httpd container
```

```
# docker run -d -p 8081:80 --name h8081  myhttpd:1.0
fca7f4525bc618e7c503b73bfa680c055300e8b5c767d48e33669831e0bc5bec
# docker run -d -p 127.0.0.1:8082:80 --name h8082 myhttpd:1.0
014e5efa5ca90d9b7e50eebef3c7f020f08f0b5238f98420681ee348a4097829
```

formatting

```
# docker ps --format "table {{.Names}}\t{{.Image}}\t{{.ID}}\t{{.Ports}}" -n2
NAMES               IMAGE               CONTAINER ID        PORTS
h8081               myhttpd:1.0         fca7f4525bc6        0.0.0.0:8081->80/tcp
h8082               myhttpd:1.0         014e5efa5ca9        127.0.0.1:8082->80/tcp
```

Running the Container: Restarting Policy

```
# docker run -d --restart=always --name sleeper centos sleep 5
6c3d24b3f89f13de92e710fcbb5b343b4cb81e7454ecc79445e94f0c5ba31a49
```

```
# docker ps --format "table {{.Names}}\t{{.Image}}\t{{.ID}}\t{{.RunningFor}}\t{{.Status}}" -n1
NAMES          IMAGE         CONTAINER ID        CREATED             STATUS
sleeper        centos        33675bff7f47        2 seconds ago       Up 1 second
```

```
# docker ps --format "table {{.Names}}\t{{.Image}}\t{{.ID}}\t{{.RunningFor}}\t{{.Status}}" -n1
NAMES          IMAGE         CONTAINER ID        CREATED             STATUS
sleeper        centos        33675bff7f47        11 seconds ago      Up 4 seconds
```

```
# docker ps --format "table {{.Names}}\t{{.Image}}\t{{.ID}}\t{{.RunningFor}}\t{{.Status}}" -n1
NAMES          IMAGE         CONTAINER ID        CREATED              STATUS
sleeper        centos        33675bff7f47        About a minute ago   Restarting (0) 6 seconds ago
```

```
# docker inspect -f "{{ .RestartCount }}" sleeper
19
```

More details

Policy	Result
no	Do not automatically restart the container when it exits. This is the default.
on-failure[:max-retries]	Restart only if the container exits with a non-zero exit status. Optionally, limit the number of restart retries the Docker daemon attempts.
always	Always restart the container regardless of the exit status. When you specify always, the Docker daemon will try to restart the container indefinitely. The container will also always start on daemon startup, regardless of the current state of the container.
unless-stopped	Always restart the container regardless of the exit status, including on daemon startup, except if the container was put into a stopped state before the Docker daemon was stopped.

Running Containers in Interactive Mode

```
[root@localhost ~]# docker run centos cat /etc/redhat-release
CentOS Linux release 7.5.1804 (Core)
```

```
[root@localhost ~]# docker run ubuntu cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=18.04
DISTRIB_CODENAME=bionic
DISTRIB_DESCRIPTION="Ubuntu 18.04.1 LTS"
```

```
[root@localhost ~]# docker run -it centos bash
[root@dfc1b0d4f6a5 /]# cat /etc/redhat-release
CentOS Linux release 7.5.1804 (Core)
[root@dfc1b0d4f6a5 /]# yum install epel-release
...
[root@dfc1b0d4f6a5 /]# exit
exit
[root@localhost ~]#
```

```
[root@localhost ~]# docker ps --format "table {{.ID}}\t{{.RunningFor}}\t{{.Status}}" -n1 -a
CONTAINER ID        CREATED             STATUS
dfc1b0d4f6a5        7 minutes ago       Exited (1) 1 minutes ago
```

```
[root@localhost ~]# docker rm dfc1b0d4f6a5
dfc1b0d4f6a5
```

```
[root@localhost ~]# docker run --rm centos cat /etc/redhat-release
CentOS Linux release 7.5.1804 (Core)
```

Executing Commands Inside Running Container

```
# docker run -d centos sleep infinity
9626a94669c935ea140bcb9ea83339bd325b28195fc088a690b620eb12902b33
```

```
# docker ps -l
CONTAINER ID   IMAGE    COMMAND            CREATED          STATUS          PORTS   NAMES
9626a94669c9   centos   "sleep infinity"   11 seconds ago   Up 10 seconds           priceless_einstein
```

```
# docker exec -it 9626a94669c9 bash
[root@9626a94669c9 /]# ps -ef
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 13:43 ?        00:00:00 sleep infinity
root          34     0  2 13:48 pts/0    00:00:00 bash
root          47    34  0 13:48 pts/0    00:00:00 ps -ef
[root@9626a94669c9 /]# exit
exit
#
```

Stopping/Deleting Containers

```
# docker ps --format "table {{.Image}}\t{{.Names}}\t{{.ID}}\t{{.RunningFor}}\t{{.Status}}"
IMAGE          NAMES    CONTAINER ID        CREATED              STATUS
myhttpd:1.0    h8081    fca7f4525bc6        About an hour ago    Up About an hour
myhttpd:1.0    h8082    014e5efa5ca9        About an hour ago    Up About an hour
```

```
# docker stop h8082
014e5efa5ca9
```

```
# docker rm 014e5efa5ca9
014e5efa5ca9
```

```
# docker rm $(docker stop 014e5efa5ca9)
014e5efa5ca9
```

```
# docker rm $(docker stop $(docker ps -a -q))
014e5efa5ca9
fca7f4525bc6
```

☆

```
# docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N]
```

☆

```
# docker image prune
WARNING! This will remove all dangling images.
Are you sure you want to continue? [y/N]
you want to continue? [y/N]
```

Changing Container's Build Defaults

1. Default User

```
# docker run -it jenkins id
uid=1000(jenkins) gid=1000(jenkins) groups=1000(jenkins)
```

--user == -u

```
# docker run -it --user 0 jenkins id
uid=0(root) gid=0(root) groups=0(root)
```

```
# docker run -it --user 1000:0 jenkins id
uid=1000(jenkins) gid=0(root) groups=0(root)
```

```
# docker run -it --group-add 123 jenkins id
uid=1000(jenkins) gid=1000(jenkins) groups=1000(jenkins),123
```

2. Default Workdir

```
# docker run -it jenkins pwd
/
```

--workdir == -w

```
# docker run -it --workdir /var/jenkins_home jenkins pwd
/var/jenkins_home
```

```
# docker run -it -w $(pwd) -v $(pwd):$(pwd) maven clean package
```

Changing Container's Build Defaults

3. Default Entrypoint

```
# docker run -it myhttpd:1.0 bash
# echo $?
1
```

```
# docker run -it --entrypoint=/bin/bash myhttpd:1.0
[root@e4e7976cf838 /]# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  1 19:27 pts/0    00:00:00 /bin/bash
root        14     1  0 19:27 pts/0    00:00:00 ps -ef
[root@e4e7976cf838 /]# exit
#
```

4. Environment Variables

```
# docker run -it -e MYVAR="My Variable" centos env | grep MYVAR
MYVAR=My Variable
```

```
# docker run -it --env-file <(env| grep ARM | cut -f1 -d=) centos env | grep ARM
ARM_SUBSCRIPTION_ID=64a3f30f-xxx-xxxx-xxxx-yyyyfe63fe9a
ARM_TENANT_ID=bd5c6713-xxx-yyyy-xxxx-78f2d078e543
ARM_CLIENT_SECRET=xxxxxxxxxxxxxxxx
ARM_CLIENT_ID=808f38ed-xxxx-xxxx-yyyy-ebbce91bcfee
```

MOUNTS

- Mounting Data from Host

Mounting Data from Host

We have a directory on the host with valuable data which we need to share into the container

```
# ls -l ./
total 40
-rw-r--r--   1 sbeliakou  wheel  119 July 25 22:05 index.html

# docker run -d -P -v $(pwd):/var/www/html httpd
# docker run -d -P -v $(pwd):/var/www/html:ro httpd

# docker run -d -p 127.0.0.1:8080:80 -v $(pwd):/var/www/html:ro httpd
```

The -v flag can also be used to mount a single file - instead of *just* directories - from the host machine

```
# docker run --rm -it -v ~/.bash_history:/root/.bash_history ubuntu /bin/bash
```

```
# ls -l ./
total 40
-rw-r--r--   1 root  root  119 July 25 22:05 index.html

# docker run -P -v $(pwd):$(pwd) -w $(pwd) busybox ls -l
total 40
-rw-r--r--   1 root  root  119 July 25 22:05 index.html
```

Mounting Data from Host

```
# ls
playbook.yml

# which ansible-playbbok
/usr/bin/which: no ansible-playbbok in (/sbin:/bin:/usr/sbin:/usr/bin)

# alias ansible-playbook='docker run -v $(pwd):$(pwd) -w $(pwd) ansible:2.6.1'

# ansible-playbook playbook.yml -vv
PLAYBOOK: playbook.yml *********************************************************
1 plays in playbook.yml

PLAY [localhost] ***************************************************************

TASK [Gathering Facts] *********************************************************
task path: /vagrant/playbook.yml:1
ok: [localhost]

TASK [debug] *******************************************************************
task path: /vagrant/playbook.yml:4
ok: [localhost] => {
    "ansible_host": "localhost"
}

PLAY RECAP *********************************************************************
localhost                  : ok=2    changed=0    unreachable=0    failed=0
```

GETTING LOGS FROM CONTAINERS

- In-container Logs
- Container Log Drivers

Container Logs

By default, **docker logs** shows the command’s **STDOUT** and **STDERR**

```
# docker run hello-world
# docker logs $(docker ps -aql)
```

Let's check container from *myhttpd:latest* image:

```
# docker run -d -P myhttpd:latest
791c65b5aed05a11a40a953779675b5b94d090e691730c5c0bceaa33143fcbc4

# docker logs $(docker ps -lq)
AH00558: httpd: Could not reliably determine the server's fully qualified domain name,
using 172.17.0.17. Set the 'ServerName' directive globally to suppress this message
```

```
# curl localhost:$(docker port $(docker ps -lq) | cut -d: -f2)
my httpd container

# docker logs $(docker ps -lq)
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using
172.17.0.17. Set the 'ServerName' directive globally to suppress this message
```

Container Logs

Let’s create “Dockerfile” file with following content

```
FROM centos
LABEL maintainer="Siarhei Beliakou"
RUN yum install -y httpd web-assets-httpd && yum clean all
RUN echo "logs are sending to stdout" > /var/www/html/index.html
# ADD index.html /var/www/html/
RUN ln -s /dev/stdout /var/log/httpd/access_log && \
    ln -s /dev/stderr /var/log/httpd/error_log
EXPOSE 80
CMD httpd -DFOREGROUND
```

```
FROM myhttpd:1.0
RUN echo "logs are sending to stdout" > /var/www/html/index.html
RUN ln -s /dev/stdout /var/log/httpd/access_log && \
    ln -s /dev/stderr /var/log/httpd/error_log
```

And build it:

```
# docker build -t myhttpd:2.0 .
...
Successfully built 5ffd0ffc1780
Successfully tagged myhttpd:2.0
```

Container Logs

```
# docker run -d -P myhttpd:2.0
1b0c3a82d0687519ffcd2ee06d345a4d3ad8d475dc0d7710fb279bdd836e5e88

# docker logs $(docker ps -lq)
AH00558: httpd: Could not reliably determine the server's fully qualified domain name,
using 172.17.0.17. Set the 'ServerName' directive globally to suppress this message
```

```
# curl localhost:$(docker port $(docker ps -lq) | cut -d: -f2)
logs are sending to stdout

# docker logs $(docker ps -lq)
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using
172.17.0.18. Set the 'ServerName' directive globally to suppress this message
172.17.0.1 - - [29/Jul/2018:22:07:24 +0000] "GET / HTTP/1.1" 200 19 "-" "curl/7.29.0"
```

Configuring Log Driver

```
# docker run -d -P --name=myhttpd --log-driver=journald myhttpd:2.0
5748f3078b647c43a335bdc1f8bc7c8d9db31a491e635effd58b31b1429c8ee7

# curl localhost:$(docker port $(docker ps -lq) | cut -d: -f2)
logs are sending to stdout
```

```
# journalctl -b CONTAINER_NAME=myhttpd
-- Logs begin at Sat 2018-07-28 19:14:07 BST, end at Sun 2018-07-29 23:19:52 BST. --
Jul 29 23:19:29 localhost.localdomain 5748f3078b64[2806]: AH00558: httpd: Could not reliably
determine the server's fully qualified domain name, using 172.17.
Jul 29 23:19:50 localhost.localdomain 5748f3078b64[2806]: 172.17.0.1 - - [29/Jul/2018:22:19:50
+0000] "GET / HTTP/1.1" 200 19 "-" "curl/7.29.0"
```

Configuring Log Driver

```
# docker run -d -P --log-driver=journald --log-opt tag=myhttpd myhttpd:latest
0555d7a41ab6af098dfea296e2fa7b4f1c6b73424e2156c387930b13bfcefb24

# curl localhost:$(docker port $(docker ps -lq) | cut -d: -f2)
logs are sending to stdout
```

```
# journalctl -b CONTAINER_TAG=myhttpd
-- Logs begin at Sat 2018-07-28 19:14:07 BST, end at Sun 2018-07-29 23:27:27 BST. --
Jul 29 23:26:02 localhost.localdomain myhttpd[2806]: AH00558: httpd: Could not reliably determine
the server's fully qualified domain name, using 172.17.0.21.
Jul 29 23:26:26 localhost.localdomain myhttpd[2806]: 172.17.0.1 - - [29/Jul/2018:22:26:26 +0000]
"GET / HTTP/1.1" 200 19 "-" "curl/7.29.0"
```

Supported Log Drivers

Driver	Description
none	No logs are available for the container and docker logs does not return any output.
json-file	The logs are formatted as JSON. The default logging driver for Docker.
syslog	Writes logging messages to the syslog facility. The syslog daemon must be running on the host machine.
journald	Writes log messages to journald. The journald daemon must be running on the host machine.
gelf	Writes log messages to a Graylog Extended Log Format (GELF) endpoint such as Graylog or Logstash.
fluentd	Writes log messages to fluentd (forward input). The fluentd daemon must be running on the host machine.
awslogs	Writes log messages to Amazon CloudWatch Logs.
splunk	Writes log messages to splunk using the HTTP Event Collector.
etwlogs	Writes log messages as Event Tracing for Windows (ETW) events. Only available on Windows platforms.
gcplogs	Writes log messages to Google Cloud Platform (GCP) Logging.
logentries	Writes log messages to Rapid7 Logentries.

Dockerd: Configuring Log Driver

```
# docker info --format '{{.LoggingDriver}}'
json-file

# cat << EOF > /etc/docker/daemon.json
{
  "log-driver": "journald"
}
EOF

# systemctl daemon-reload
# systemctl restart docker.service
# docker info --format '{{.LoggingDriver}}'
journald
```

```
# docker run -d -P --name=myhttpd_2.0 myhttpd:2.0
```

```
# journalctl -b CONTAINER_NAME=myhttpd_2.0
```

Demo

https://github.com/dbuhtiyarov/docker-demo/tree/master/java/DemoSpringBootApp

Thank you for your attention!

Dmitry Buhtiyarov, Siarhei Beliakou
2018