Dockers for developers

DevOps Course

About me

OpsGuru

https://www.opsguru.io/

lev@opsguru.io

Lev Epshtein

Technology enthusiast with 10 years of industry experience in DevOps and IT. Industry experience with back-end architecture.

Solutions architect with experience in big scale systems hosted on AWS/GCP, end-to-end DevOps automation process.

DevOps, and Big Data instructor at John Bryce.

Partner & Senior Engineering Consultant at Opsguru.

Clone Git

https://github.com/lev-tmp/seminars.git

Objectives

JOHN BRYCE
Leading in IT Education
matrix company

- By the end of this session
  - You’ll be familiar with Docker Concepts & Base Commands
  - Configure Dockers using DockerFile And Passing Properties To It
  - Run Standalone Jar in docker
  - Operate Docker Hub (Push)
  - Build Docker Image with Maven
  - Advance Docker - Network and Docker compose
  - Utilize docker compose for your own CD in your local development env

Questions for you...

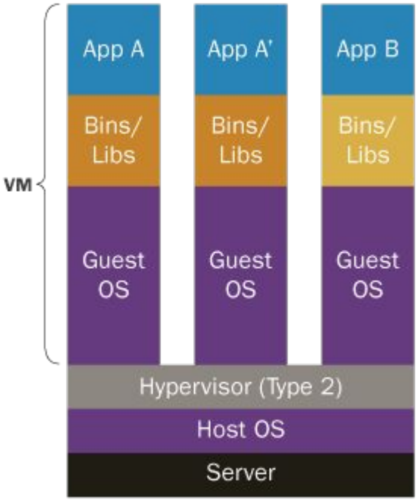
- What Do You Know About Docker?
- Who Used Docker For Development / QA / STG / PROD?
- Who Tried & Failed Implementing Docker

What is Docker

Docker is an open platform for developing, shipping, and running applications.

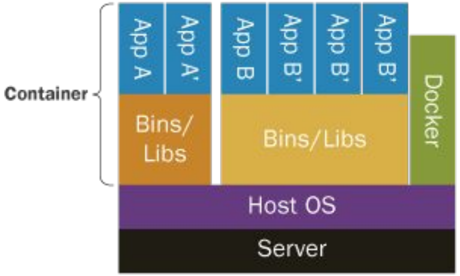
Docker allows you to package an application with all of its dependencies into a standardized unit for software development.

Containers VS. VMs



Virtual Machines

Containers are isolated, but share OS and, where appropriate, bins/libraries



Containers

Docker Benefits Upon VMs

- Small to tiny images - Few hundred MB’s for OS + Application (5MB for full OS - Alpine) VS. Gigabytes in VM’s
- Very small footprint on the host machine (CPU, RAM Impact) as Docker only use what it required instead of building a complete Operating system per VM.
- Containers use up only as many system resources as they need at a given time. VMs usually require some resources to be permanently allocated before the virtual machine starts.
- Direct hardware access. Applications running inside virtual machines generally cannot access hardware like graphics cards on the host in order to speed processing. Containers Can (ex. Nvidia)
- Microservice in nature and integrations (API’s) for whatever task required.
- Portable, Fast (Deployments , Migration , Restarts and Rollbacks) and Secure
- Can run anywhere and everywhere
- Simplify DevOps
- Version controlled
- Open Source

Common Use Cases for Docker

- CI / CD
- Fast Scaling application layers for overcoming application performance limitations.
- For Sandboxed environments (Development, Testing , Debugging)
- Local development environment (no more “ It ran on my laptop…”)
- Infrastructure as a CODE made easy with docker
- Multi-Tier applications (Front End , Mid Tier (Biz Logic) , Data Tier) / Microservices
- Building PaaS , Saas

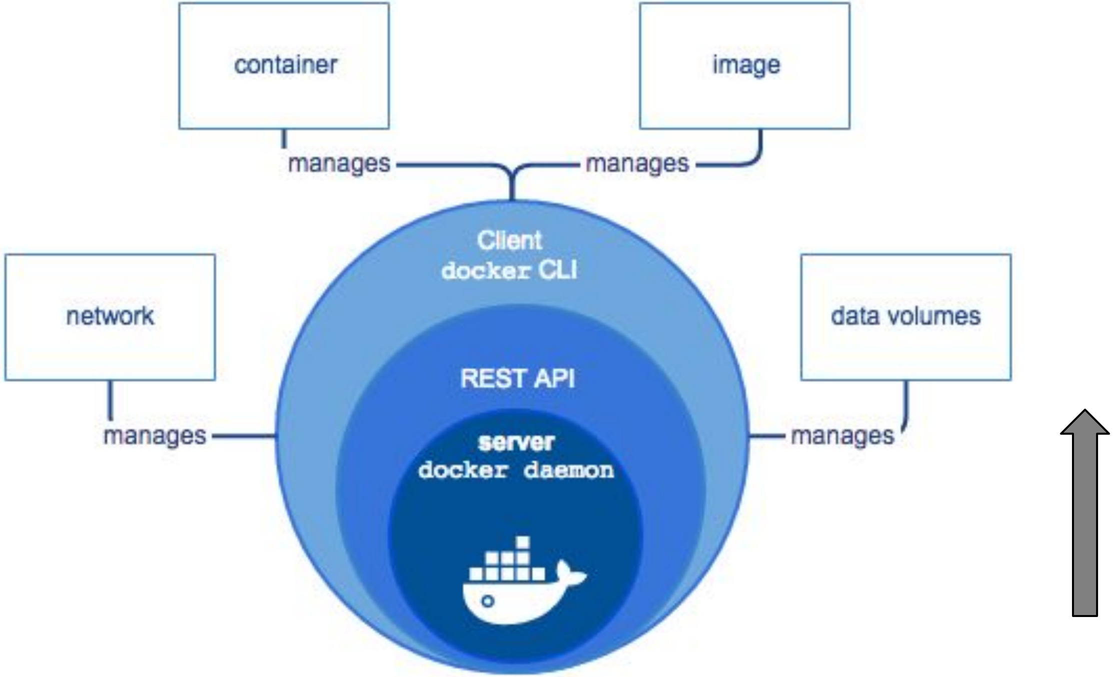
Under The Hood

- Architecture: Linux X86-64
- Written in: GoLang (On March 13, 2014, with the release of version 0.9, Docker dropped LXC as the default execution environment which is an operating system level virtualization and replaced it with its own libcontainer library written in the Go programming language)
- Engine: Client - Server (Daemon) Architecture
- Namespace: Isolation of process in linux where one process cant “See” the other process
- Control Groups: Linux Kernel capability to limit and isolate the resource usage (CPU, RAM, disk I/O, network etc..) of a collection of process
- Container format: libcontainer - Go implementation for creating containers with namespaces, control groups and File system capabilities access control

Docker Architecture

JOHN BRYCE
Leading in IT Education
matrix company

Overview



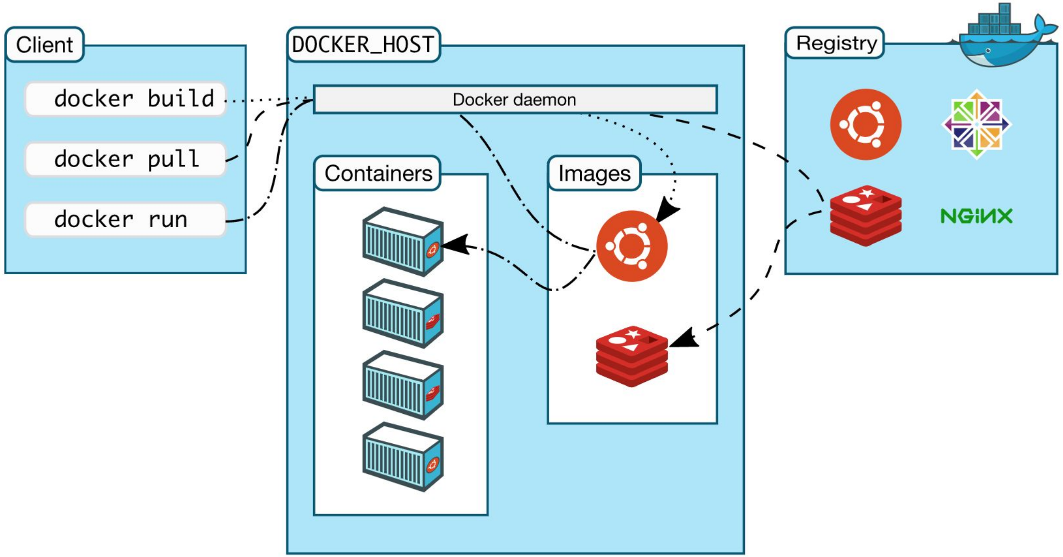
What is docker - Technical Aspect

Docker Architecture

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

Docker Architecture

Docker Architecture



Docker Components

- Daemon
- (Docker) Client
- Docker Registries
- Docker Objects
- Machine
- Compose
- Swarm

Docker Components

Daemon

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

Docker Components

Docker Client

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

Docker Components

Docker Registries

- A Docker registry stores Docker images. Docker Hub and Docker Cloud are public registries that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.
- When one use “docker pull / push / run” commands, the required images are pulled from the configured registry.

Docker Components

Docker Objects

- Images

  a. Read Only template with instruction for creating a Docker Container. Often, an Image is based on another image with some additional customization.

  b. Self own images that are fully created by you using DockerFile with a simple syntax where every instruction control a different Layer in the image. Once a change is made to a specific layer, a rebuild of the image will change only the updated layers. This what makes images small, fast and lightweight in compared to other virtualization solutions

Docker Components

Docker Objects

- Containers

    a. A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can **connect a container to one or more networks**, **attach storage to it,** or even create a new image based on its current state.

    b. Container is defined by its image as a well as any configuration options we provide to it when created or when we start it

Docker Components

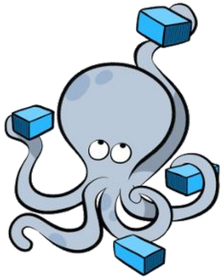
Docker Objects

- Services

  a. Allow you to scale containers across multiple Docker daemons, which all work together as a **swarm** with multiple managers and workers. Each member of a swarm is a Docker daemon, and the daemons all communicate using the Docker API. A service allows you to define the desired state, such as the number of replicas of the service that must be available at any given time. By default, the service is load-balanced across all worker nodes. To the consumer, the Docker service appears to be a single application. Docker Engine supports swarm mode in Docker 1.12 and higher.

Docker Components

Docker Compose



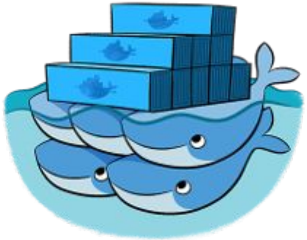
A tool for defining and running complex applications with

Docker (eg multi-container application ex. LAMP)

With a single file

Docker Components

Docker Swarm



A Native Clustering tool for Docker. Swarm pools together several Docker hosts and exposed them as a single virtual Docker host. It scale up to multiple hosts

Docker Components

Good to know:

Docker Machine

A Tool which makes it easy to create Docker Hosts on

Operating systems that does not support docker natively, or

on cloud providers and inside your datacenter.

INSTALLING DOCKER

Windows 10 Enterprise / Educational

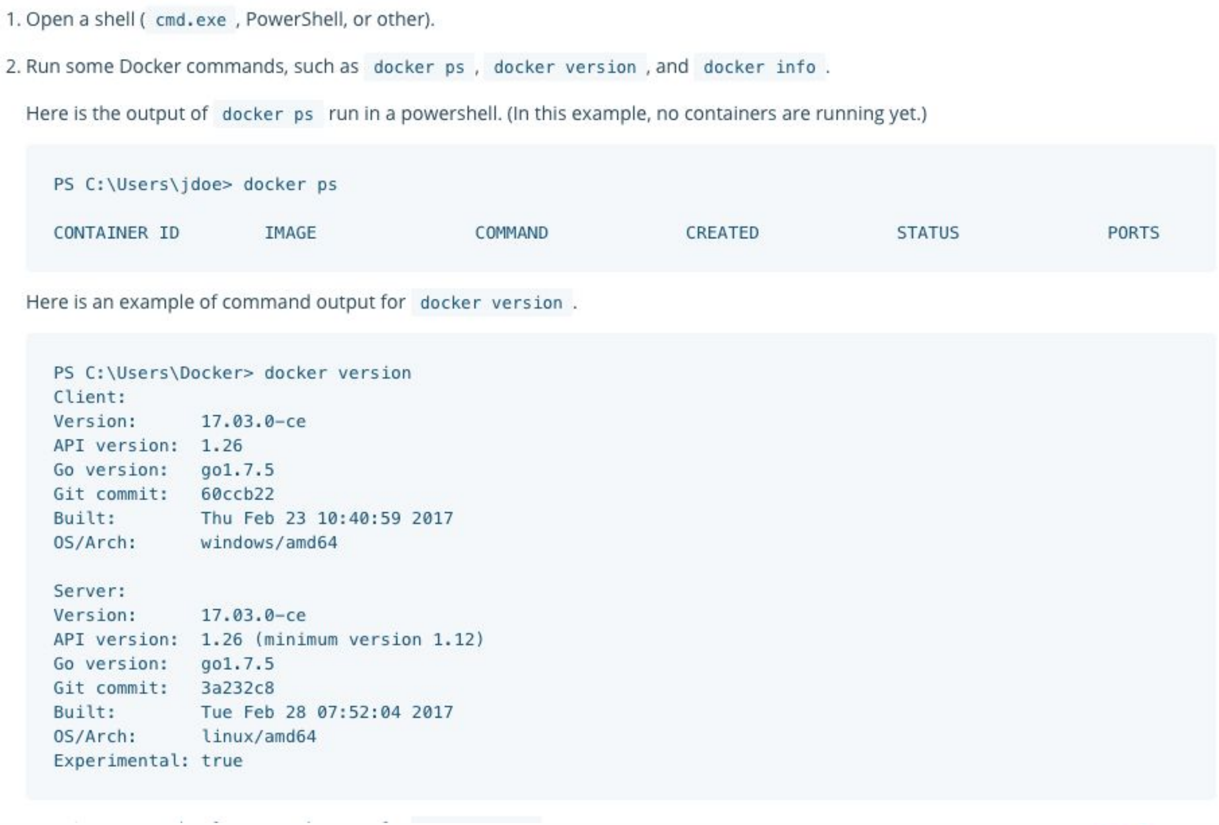
DOWNLOAD HERE

https://docs.docker.com/docker-for-windows/

Windows 10 Enterprise / Educational

1. Turn windows features on or off
   a. Enable HYPER V
   b. Restart

Windows 10 Check Functionality



1. Open a shell (`cmd.exe` , PowerShell, or other).

2. Run some Docker commands, such as `docker ps` , `docker version` , and `docker info` .

   Here is the output of `docker ps` run in a powershell. (In this example, no containers are running yet.)

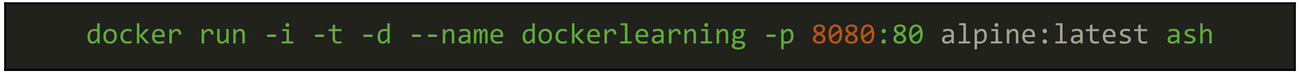
```
PS C:\Users\jdoe> docker ps

CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
```

   Here is an example of command output for `docker version` .

```
PS C:\Users\Docker> docker version
Client:
Version:      17.03.0-ce
API version:  1.26
Go version:   go1.7.5
Git commit:   60ccb22
Built:        Thu Feb 23 10:40:59 2017
OS/Arch:      windows/amd64

Server:
Version:      17.03.0-ce
API version:  1.26 (minimum version 1.12)
Go version:   go1.7.5
Git commit:   3a232c8
Built:        Tue Feb 28 07:52:04 2017
OS/Arch:      linux/amd64
Experimental: true
```

Let's Start

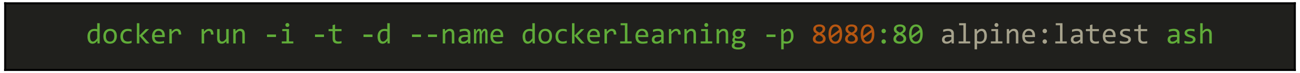
Docker Flow



```
docker run -i -t -d --name dockerlearning -p 8080:80 alpine:latest ash
```

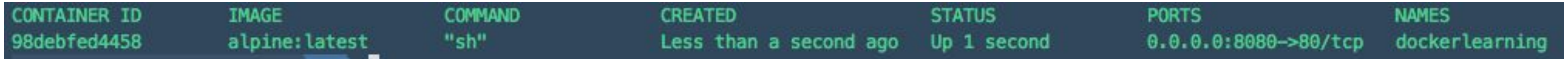
- ‘docker run’ will run the container
- This will not restart an already running container, just create a new one
- docker run [options] IMAGE [command] [arguments]
  a. [options] modify the docker process for this container
  b. IMAGE is the image to use
  c. [command] is the command to run inside the container (entry point to hold the container running)
  d. [arguments] are arguments for the command

Docker Flow



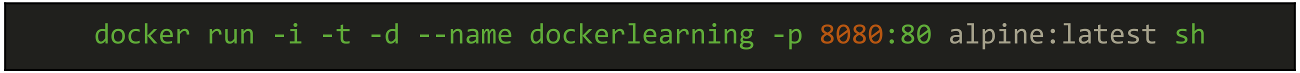
```
docker run -i -t -d --name dockerlearning -p 8080:80 alpine:latest ash
```

- ‘docker run’ will run the container
  a. -i - Interactive mode
  b. -t - Allocate pseudo TTY - or not Terminal will be available
  c. -d - Run in the background (Daemon style)
  d. --name - Give the container a name or let Docker to name it
  e. -p [local port] : [container port] - Forward local port to the container port



```
CONTAINER ID   IMAGE           COMMAND   CREATED                  STATUS         PORTS                    NAMES
98debfed4458   alpine:latest   "sh"      Less than a second ago   Up 1 second    0.0.0.0:8080->80/tcp     dockerlearning
```

Docker Flow



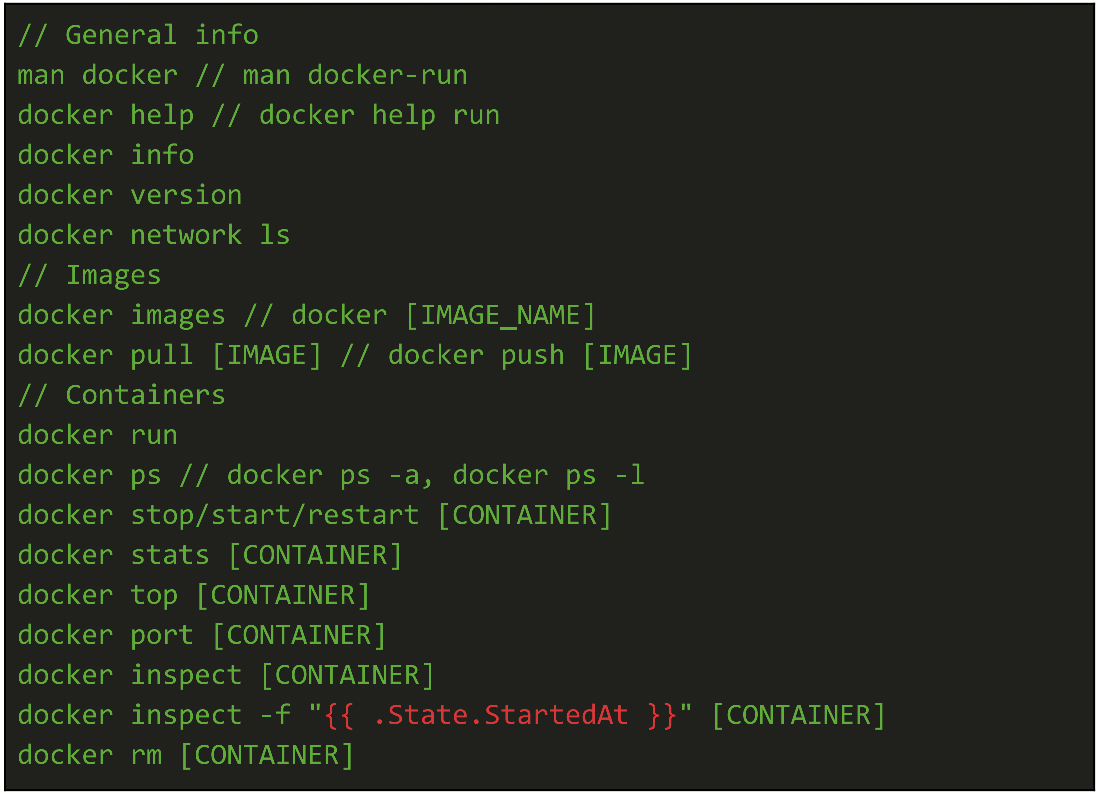
```
docker run -i -t -d --name dockerlearning -p 8080:80 alpine:latest sh
```

- Pulls the alpine:latest image from the registry (if not existed on our station)
  a. Run “docker images” to see what images already downloaded / in use locally
- Creates new container
- Allocate FS and Mounts a read-write Layer
- Allocates network/bridge interface
- Set up an IP Address
- Executes a process that we specify (in this scenario - “sh” as alpine release doesn't have bash)
- Captures and provides application outputs

Docker Examples

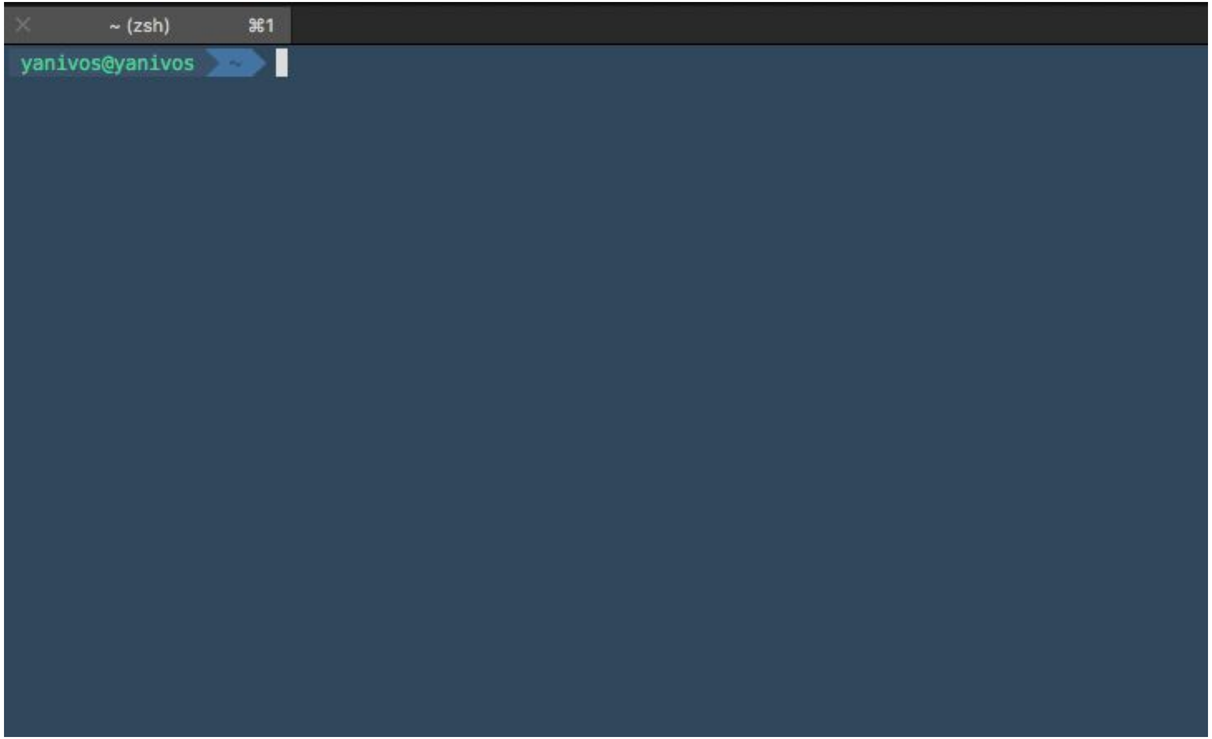
- Pull / Run an image
- SSH into a container
- View Logs
- Docker Volume
- Using Dockerfile - Building our own Jar
- Package an app and push it to a repo

Common Docker Commands



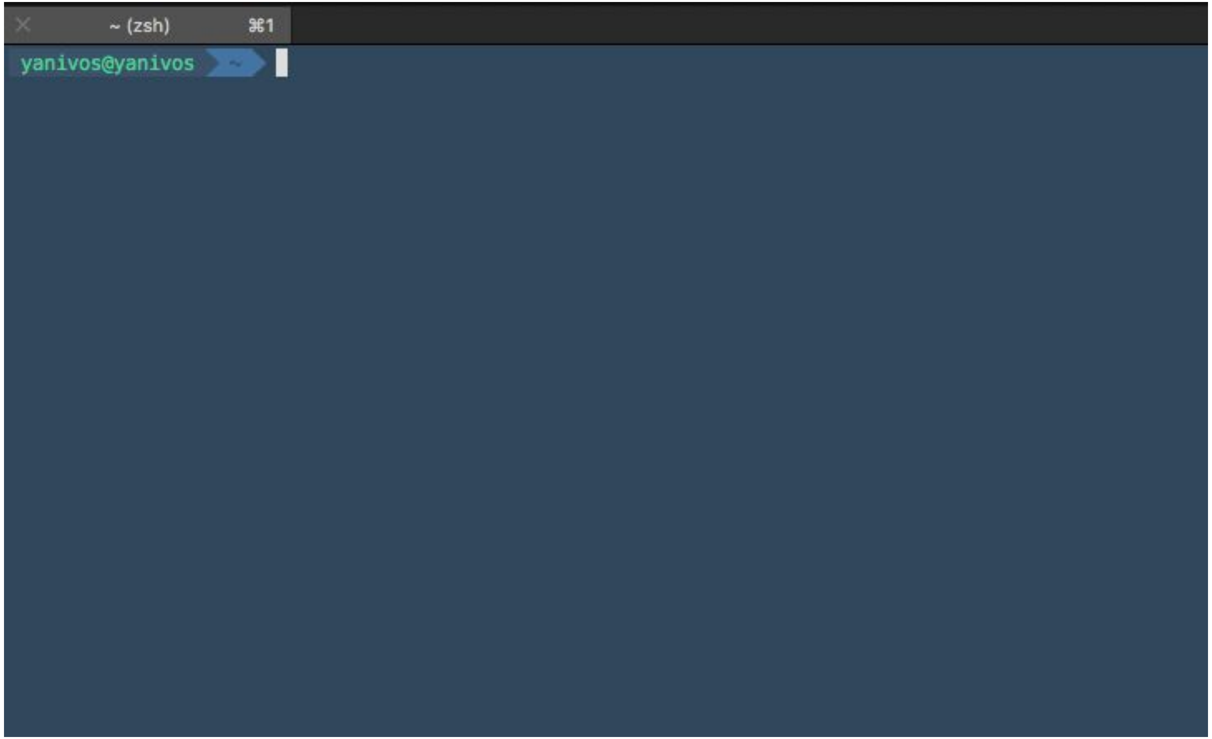
```
// General info
man docker // man docker-run
docker help // docker help run
docker info
docker version
docker network ls
// Images
docker images // docker [IMAGE_NAME]
docker pull [IMAGE] // docker push [IMAGE]
// Containers
docker run
docker ps // docker ps -a, docker ps -l
docker stop/start/restart [CONTAINER]
docker stats [CONTAINER]
docker top [CONTAINER]
docker port [CONTAINER]
docker inspect [CONTAINER]
docker inspect -f "{{ .State.StartedAt }}" [CONTAINER]
docker rm [CONTAINER]
```

Running simple shell



```
yanivos@yanivos ~
```

Building & Running Mysql On docker



```
yanivos@yanivos  ~ 
```

Why not to run SSH inside a container

- We can…
- Docker is designed for one command per container - Now we run two
- If any update or modification is needed, We need to change our setup and not the docker image..
- If you still want to review something… SSH it.

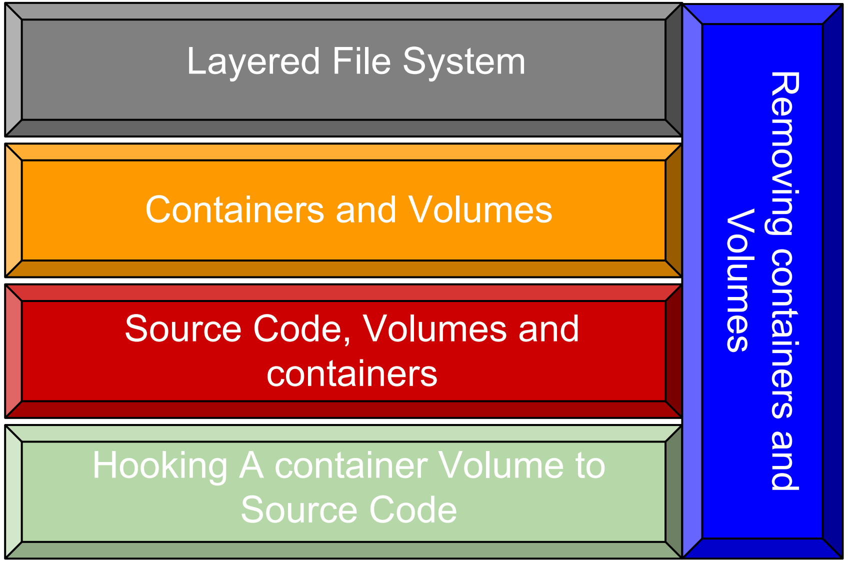
Docker Advanced

- Volumes - Hooking Source code into a container
- Networking and communications
- Building Custom Images with DockerFile
- Building Custom images with Docker Compose (v3 YAML)
- Working with images
- Building a Microservice Project
- Working with Private Registries

HOOKING SOURCE CODE

Module Agenda

To understand how we can hook our source code into a container,
We will go over the following:



Layered File System

Containers and Volumes

Source Code, Volumes and containers

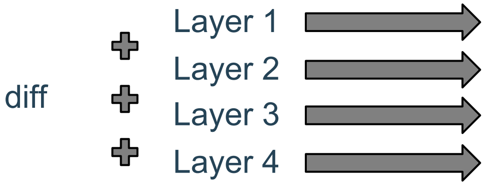
Hooking A container Volume to Source Code

Removing containers and Volumes

Layered FS

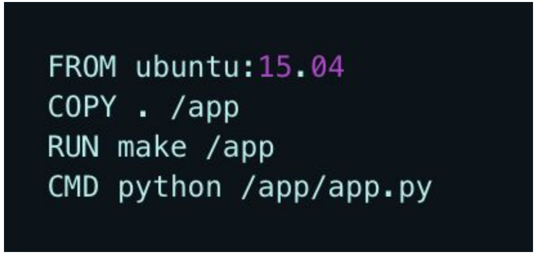
- **Images and Layers**

  A Docker image is built up from a series of layers. Each layer represents an instruction in the image’s Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile



diff

Layer 1
+ Layer 2
+ Layer 3
+ Layer 4

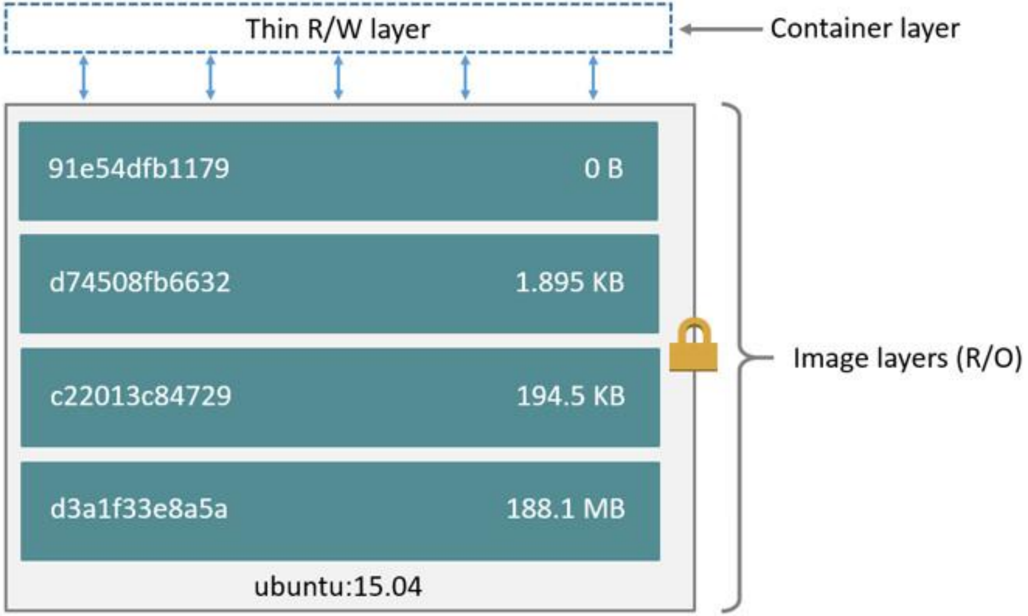


```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

Each layer is only a set of differences from the layer before it. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the **“container layer”.** All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer.

Layered FS

R/O !



Container
(based on ubuntu:15.04 image)

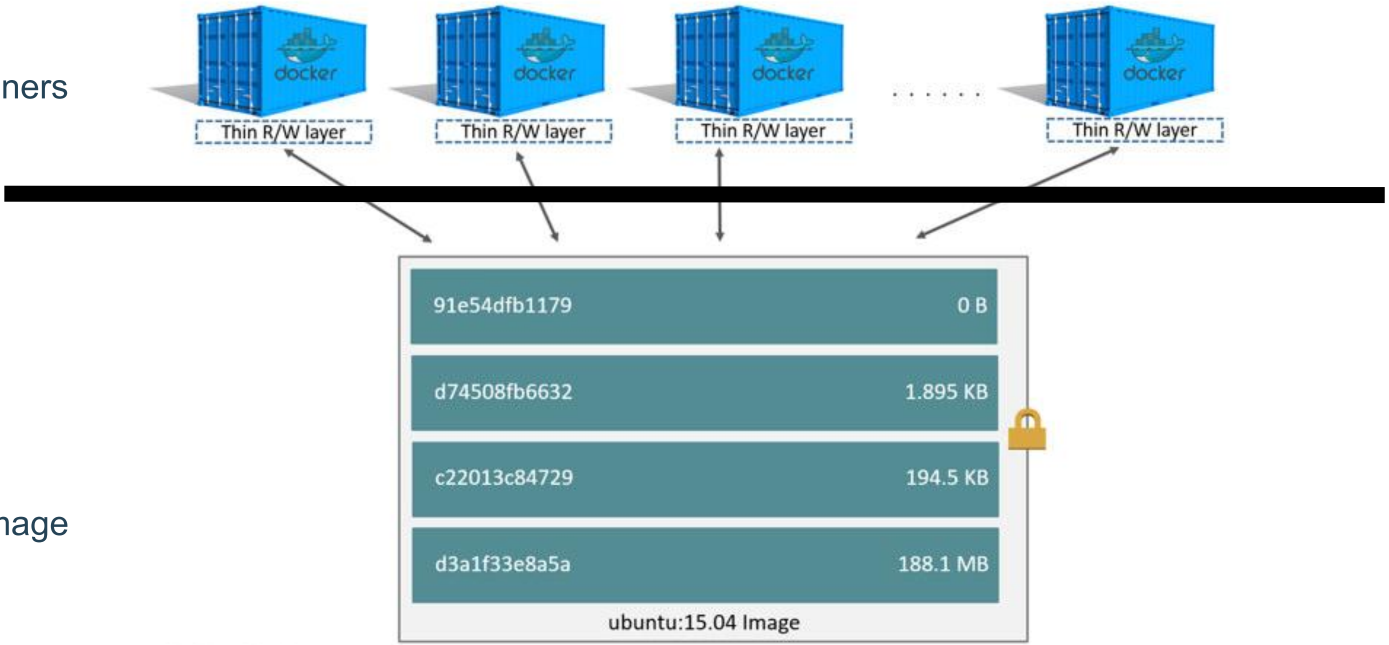
Layered FS

- **Containers and Layers**

  The major difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged. Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.

Layered FS

Different Containers



91e54dfb1179	0 B
d74508fb6632	1.895 KB
c22013c84729	194.5 KB
d3a1f33e8a5a	188.1 MB

ubuntu:15.04 Image

Using same Image

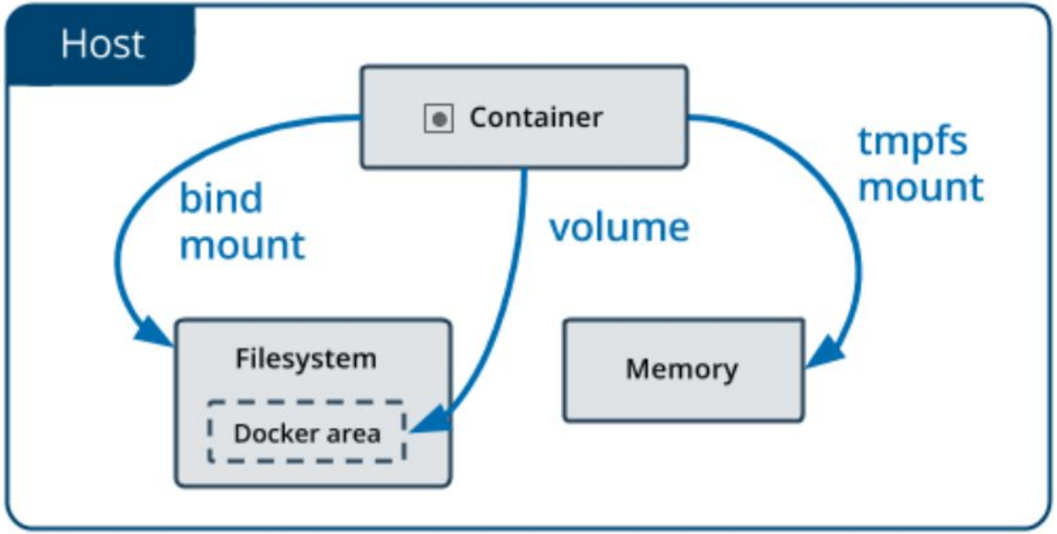
Note: If we need multiple images to have shared access to the exact same data, we store this data in a Docker **volume** and mount it into your containers.

SO HOW DO WE GET OUR SOURCE CODE INTO A CONTAINER?

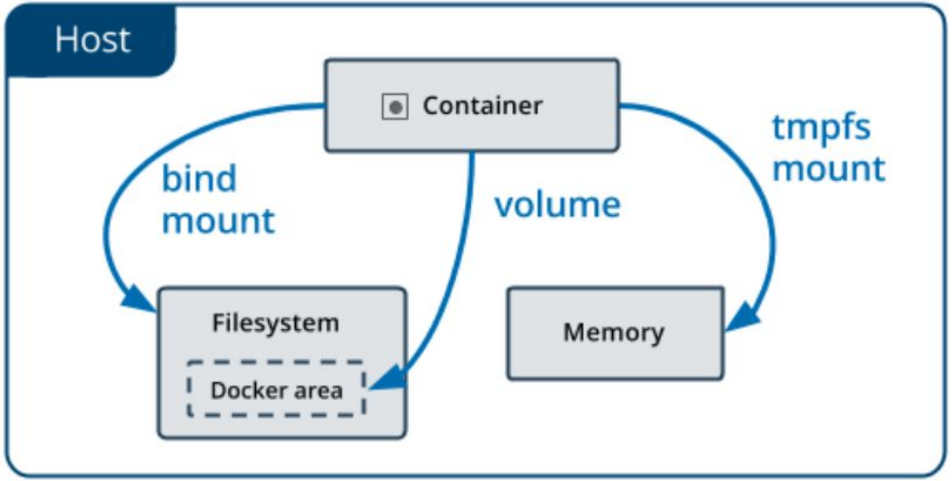
Manage Data in Docker

Manage Data in Docker

An easy way to visualize the difference among volumes, bind mounts, and `tmpfs` mounts is to think about where the data lives on the Docker host.

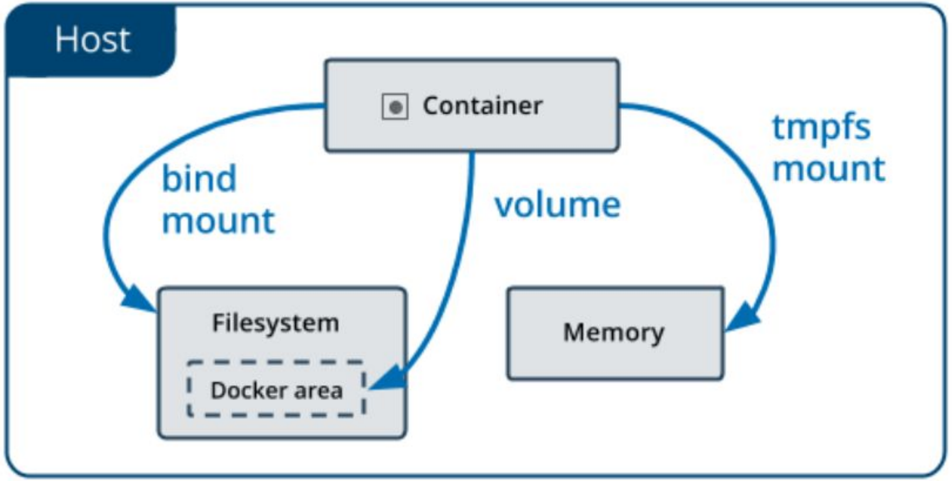


Manage Data in Docker



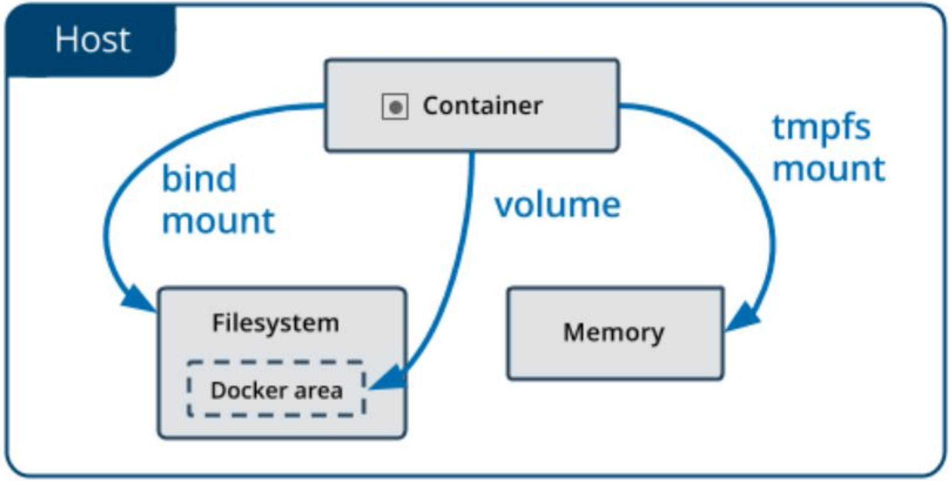
- **Volumes** are stored in a part of the host filesystem which is *managed by Docker* (`/var/lib/docker/volumes/` on Linux). Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker.

Manage Data in Docker



- **Bind mounts** may be stored *anywhere* on the host system. They may even be important system files or directories. Non-Docker processes on the Docker host or a Docker container can modify them at any time.

Manage Data in Docker



- **`tmpfs` mounts** are stored in the host system's memory only, and are never written to the host system's filesystem.

Containers and Volumes

Volumes

- **What is a Volume**

  Special type of directory in a container typically referred to as a “data volume”

  - Can be shared and reused among one or many containers
  - Updates to an image won’t affect a data volume
  - Data volumes are persisted even after container deletion
  - Volumes are OS agnostic. They can run on Linux and windows containers
  - Volumes drivers allow us to store volumes on remote hosts or cloud providers.
  - Volumes can be encrypted or to add other functionality
  - A new volume content can be pre-populated by a container

Containers and Volumes

Follow through

Volumes

- **Create and manage volumes:**

  What will we achieve in the following follow through session:

  - Creating new volume
    - Inspacting
    - Removing
  - Start a container[s] with a volume

Volumes

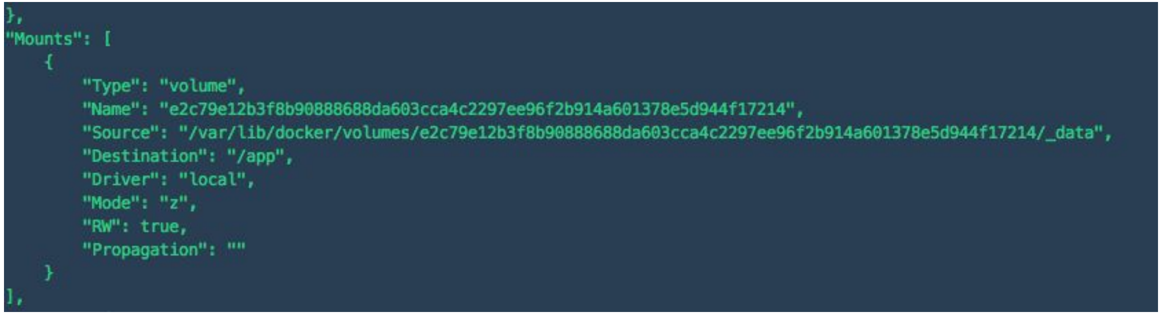
Follow through

RUN

```
docker run -dti --name alpine1 --mount target=/app alpine ash
```

INSPECT

```
docker inspect alpine1
```



```
},
"Mounts": [
    {
        "Type": "volume",
        "Name": "e2c79e12b3f8b90888688da603cca4c2297ee96f2b914a601378e5d944f17214",
        "Source": "/var/lib/docker/volumes/e2c79e12b3f8b90888688da603cca4c2297ee96f2b914a601378e5d944f17214/_data",
        "Destination": "/app",
        "Driver": "local",
        "Mode": "z",
        "RW": true,
        "Propagation": ""
    }
],
```

STOP AND DELETE CONTAINER

```
docker stop alpine1 && docker rm alpine1
```

Volumes

Follow through 2

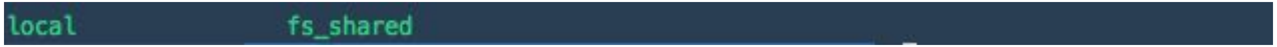
Creating a VOLUME managed by docker FS and share it with multiple containers

RUN

```
docker volume create fs_shared
```

LIST VOLUMES

```
docker volume ls
```



```
local               fs_shared
```

RUN AND MOUNT

```
docker run --rm -tdi --name alpine1 --mount source=fs_shared,target=/app alpine ash
```

```
docker run --rm -tdi --name alpine2 --mount source=fs_shared,target=/app alpine ash
```

```
docker run --rm -tdi --name alpine3 --mount source=fs_shared,target=/app alpine ash
```

Volumes

LAB

Attach to running containers, create files and verify files gets updated on all containers

Disconnect sequence Ctrl + p + Ctrl + q

Containers and Volumes

BIND MOUNTS

Volumes

- **Bind Mounts**

  Bind mounts have been around since the early days of Docker. Bind mounts have limited functionality compared to volumes. When you use a bind mount, a file or directory on the host machine is mounted into a container. The file or directory is referenced by its full or relative path on the host machine. By contrast, when you use a volume, a new directory is created within Docker’s storage directory on the host machine, and Docker manages that directory’s contents.

COMMAND EXAMPLE

-v

```
docker run --rm -tdi -v “$(pwd)”/source:/app [image] [CMD]
```

--mount

```
docker run --rm -tdi --mount type=bind,source=”$(pwd)”/source,target=/app [image] [CMD]
```

Volumes

- WINDOWS MAP VOLUME BIND

-v

```
docker run --rm -tdi -v C:/folder/name:/data [image] [CMD]
```

Volumes

- **BIND MOUNTS USING -V OR --MOUNT ?**
  - Both will provide the same outcome but as -v /--volume exists since day 1 in docker and --mount was introduced since docker 17.06 it became normal and easier to use --mount.

Containers and Volumes

LAB: BIND MOUNTS

Volumes

- **Create and manage bind mount:**
  - Create new host local project folder called “jb_docker” and cd into it
    - Create 2 alpine nodes and share new local folder called source1 using --mount
    - Create 2 alpine nodes and share new local folder called source2 using -v
    - What happened when you tried creating a shared host folder with --mount without first creating the folder manually ? and what happened when you were using -v
  - Inspect the new volumes and containers
  - Validate shared folder by creating files and make sure the exists on both containers
  - Stop all docker containers and Make sure containers got deleted

Containers and Volumes

LAB: Running BootStrap app in a container

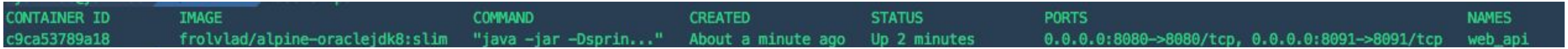
Volumes

- **Hook SpringBoot Jar into a container:**
  - Cd into your “jb_docker” folder
    - Copy from your cloned git the demo artifact to ./source

      seminars/docker//artifacts/**spring-music.jar**
  - Run 1 new container
    - Name: web_api
    - Mount Using -v or --mount
      - source: ./source
      - Target: /app
    - Image: **frolvlad/alpine-oraclejdk8:slim**
    - CMD: `java -jar -Dspring.profiles.active  /app/spring-music.jar`

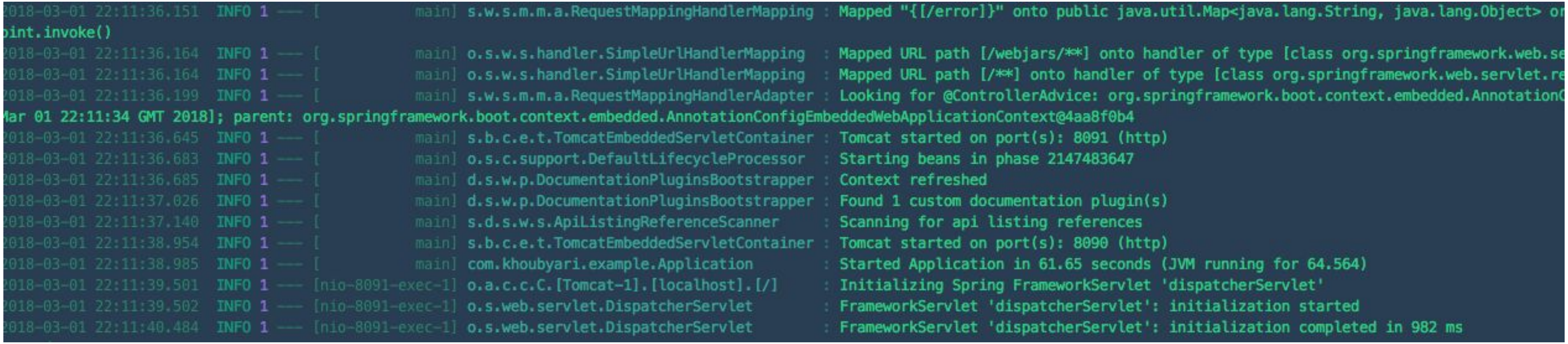
Volumes

- **Validate your work:**
  - Run docker ps and expect to see the following



```
CONTAINER ID   IMAGE                              COMMAND                  CREATED              STATUS          PORTS                                            NAMES
c9ca53789a18   frolvlad/alpine-oraclejdk8:slim    "java -jar -Dsprin..."   About a minute ago   Up 2 minutes    0.0.0.0:8080->8080/tcp, 0.0.0.0:8091->8091/tcp   web_api
```

  - Run docker logs OR attach and expect seeing the following (remember ctrl+p+ctrl+q to disconnect)



```
2018-03-01 22:11:36.151  INFO 1 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto public java.util.Map<java.lang.String, java.lang.Object> or
oint.invoke()
2018-03-01 22:11:36.164  INFO 1 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.se
2018-03-01 22:11:36.164  INFO 1 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handler of type [class org.springframework.web.servlet.re
2018-03-01 22:11:36.199  INFO 1 --- [           main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.context.embedded.AnnotationC
Mar 01 22:11:34 GMT 2018]; parent: org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@4aa8f0b4
2018-03-01 22:11:36.645  INFO 1 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8091 (http)
2018-03-01 22:11:36.683  INFO 1 --- [           main] o.s.c.support.DefaultLifecycleProcessor  : Starting beans in phase 2147483647
2018-03-01 22:11:36.685  INFO 1 --- [           main] d.s.w.p.DocumentationPluginsBootstrapper : Context refreshed
2018-03-01 22:11:37.026  INFO 1 --- [           main] d.s.w.p.DocumentationPluginsBootstrapper : Found 1 custom documentation plugin(s)
2018-03-01 22:11:37.140  INFO 1 --- [           main] s.d.s.w.s.ApiListingReferenceScanner     : Scanning for api listing references
2018-03-01 22:11:38.954  INFO 1 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8090 (http)
2018-03-01 22:11:38.985  INFO 1 --- [           main] com.khoubyari.example.Application        : Started Application in 61.65 seconds (JVM running for 64.564)
2018-03-01 22:11:39.501  INFO 1 --- [nio-8091-exec-1] o.a.c.c.C.[Tomcat-1].[localhost].[/]     : Initializing Spring FrameworkServlet 'dispatcherServlet'
2018-03-01 22:11:39.502  INFO 1 --- [nio-8091-exec-1] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization started
2018-03-01 22:11:40.484  INFO 1 --- [nio-8091-exec-1] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization completed in 982 ms
```

Volumes

Try Browsing from your host browser: http://locahost:8080/

Did it worked?

- What do you need to do to forward request to port 8080 and 8080 to your docker web_api ?

FINAL SOLUTION

HOOKING YOUR OWN SOURCE CODE

Volumes

```
docker run -tdi --name web_api -v "$(pwd)"/source:/app -p 8080:8080 frolvlad/alpine-oraclejdk8:slim java -jar
-Dspring.profiles.active  /app/spring-music.jar
```

STOP AND REMOVE

CLEAN UP

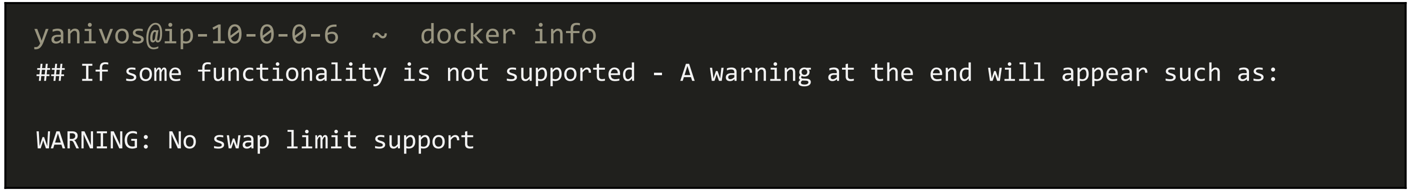
CONTAINERS ADVANCE

Limit a container's resources before we continue to Network

CONTAINERS ADVANCE

- By default, a container has no resource constraints and can use as much of a given resource as the host’s kernel scheduler allows. 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.

Verify support by running:



```
yanivos@ip-10-0-0-6  ~  docker info
## If some functionality is not supported - A warning at the end will appear such as:

WARNING: No swap limit support
```

CONTAINERS ADVANCE: RAM

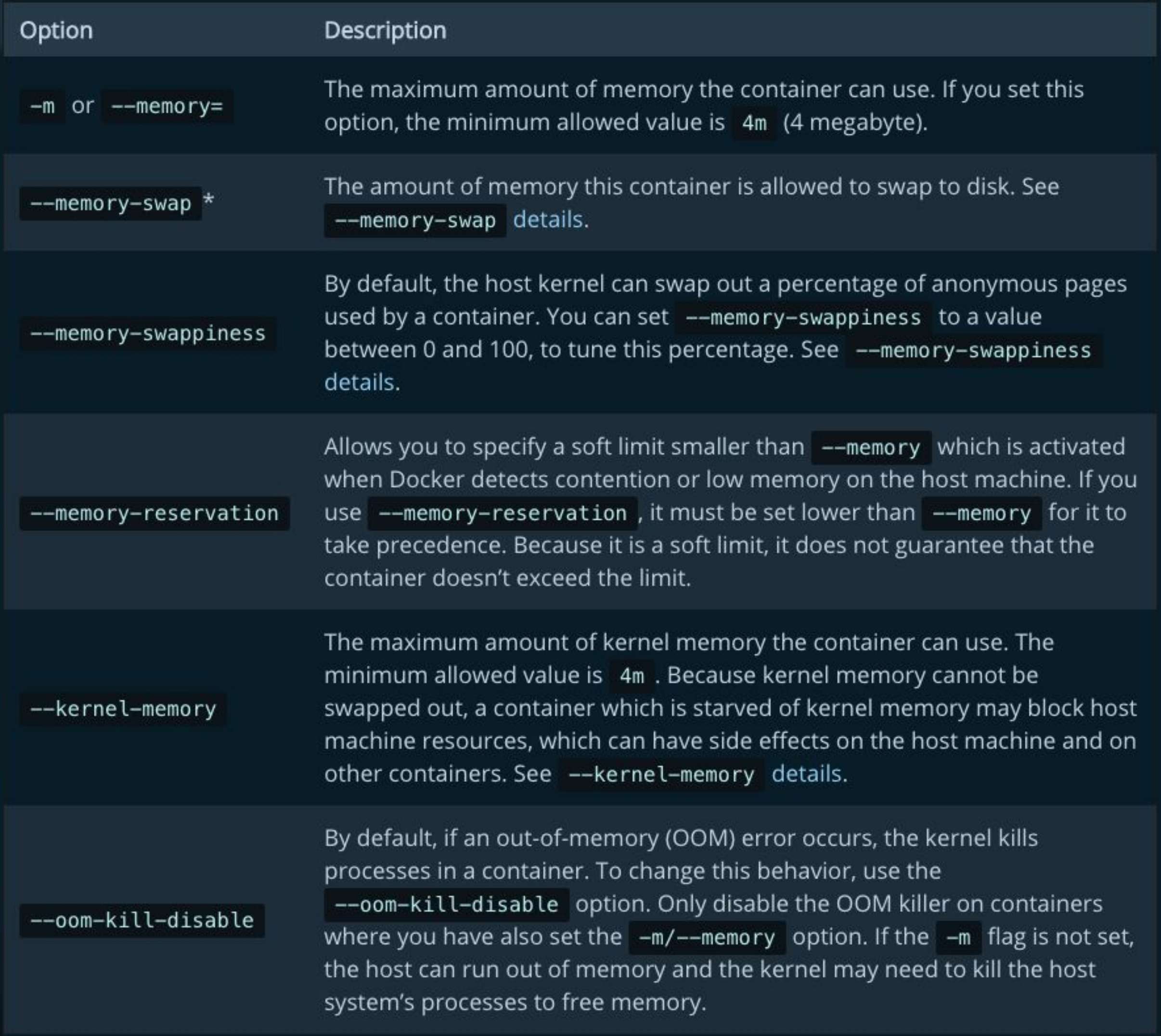
Understand the risks of running out of memory

It is important not to allow a running container to consume too much of the host machine’s memory.
On Linux hosts, if the kernel detects that there is not enough memory to perform important system functions, it throws an OOME, or Out Of Memory Exception, and starts killing processes to free up memory.
Any process is subject to killing, including Docker and other important applications. This can effectively bring the entire system down if the wrong process is killed.

CONTAINERS ADVANCE: RAM

Mitigate the risk of system instability

- Perform tests to understand the memory requirements of your application before placing it into production.
- Ensure that your application runs only on hosts with adequate resources.
- Limit the amount of memory your container can use, as described below.
- Be mindful when configuring swap on your Docker hosts. Swap is slower and less performant than memory but can provide a buffer against running out of system memory.
- Consider converting your container to a service, and using service-level constraints and node labels to ensure that the application runs only on hosts with enough memory



Option	Description
`-m` or `--memory=`	The maximum amount of memory the container can use. If you set this option, the minimum allowed value is `4m` (4 megabyte).
`--memory-swap` *	The amount of memory this container is allowed to swap to disk. See `--memory-swap` details.
`--memory-swappiness`	By default, the host kernel can swap out a percentage of anonymous pages used by a container. You can set `--memory-swappiness` to a value between 0 and 100, to tune this percentage. See `--memory-swappiness` details.
`--memory-reservation`	Allows you to specify a soft limit smaller than `--memory` which is activated when Docker detects contention or low memory on the host machine. If you use `--memory-reservation`, it must be set lower than `--memory` for it to take precedence. Because it is a soft limit, it does not guarantee that the container doesn't exceed the limit.
`--kernel-memory`	The maximum amount of kernel memory the container can use. The minimum allowed value is `4m`. Because kernel memory cannot be swapped out, a container which is starved of kernel memory may block host machine resources, which can have side effects on the host machine and on other containers. See `--kernel-memory` details.
`--oom-kill-disable`	By default, if an out-of-memory (OOM) error occurs, the kernel kills processes in a container. To change this behavior, use the `--oom-kill-disable` option. Only disable the OOM killer on containers where you have also set the `-m/--memory` option. If the `-m` flag is not set, the host can run out of memory and the kernel may need to kill the host system's processes to free memory.

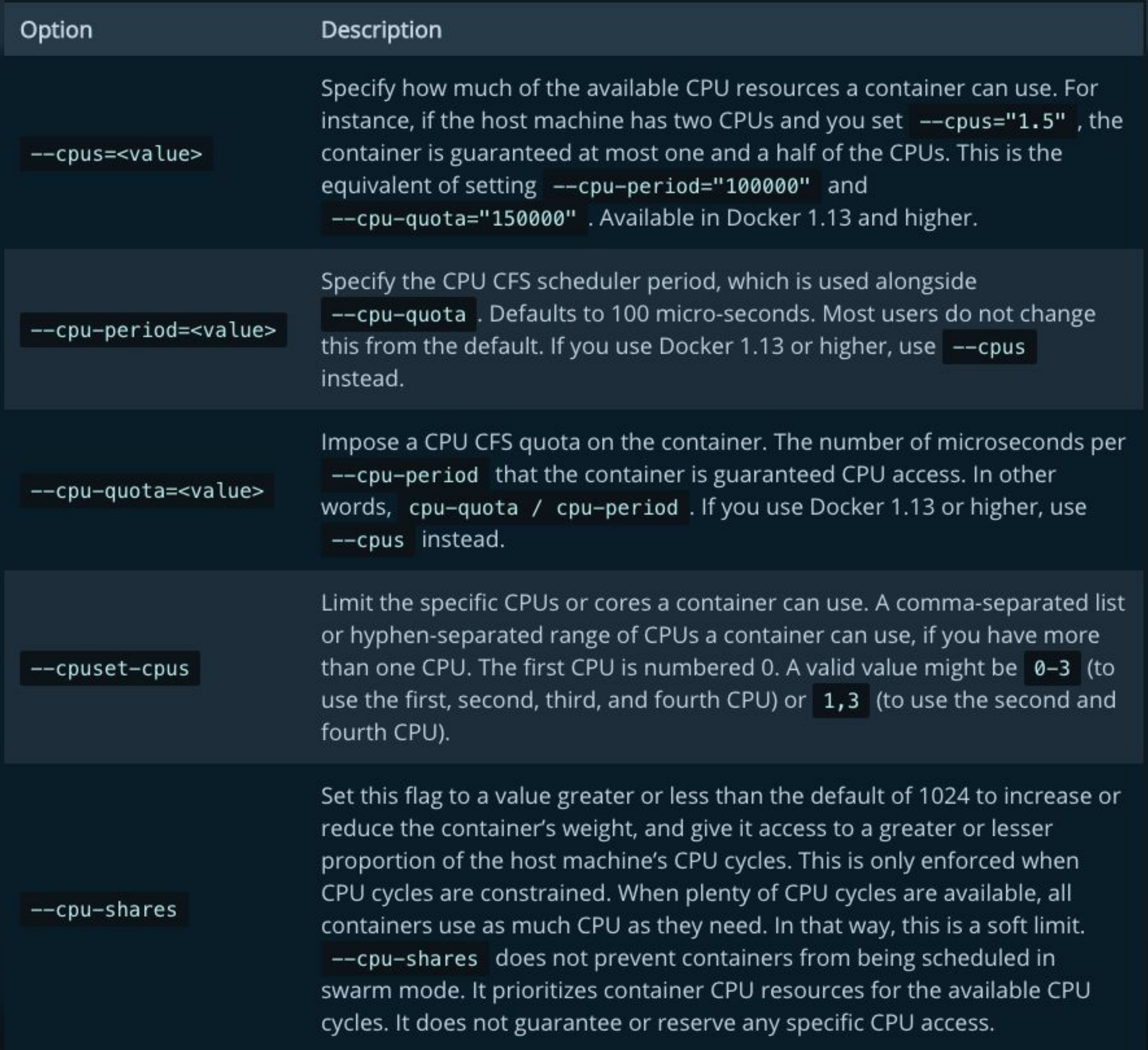
CONTAINERS ADVANCE: CPU

CPU

By default, each container’s access to the host machine’s CPU cycles is unlimited. You can set various constraints to limit a given container’s access to the host machine’s CPU cycles. Most users use and configure the default CFS scheduler. In Docker 1.13 and higher, you can also configure the realtime scheduler.

CFS scheduler:

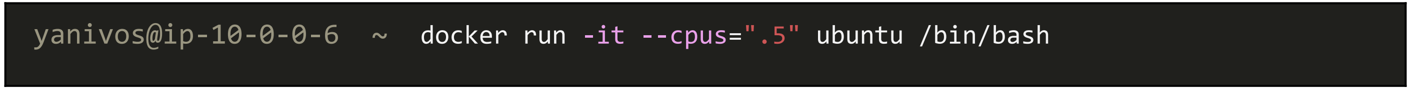
The CFS is the Linux kernel CPU scheduler for normal Linux processes. Several runtime flags allow you to configure the amount of access to CPU resources your container has (Containers uses cgroup)



Option	Description
`--cpus=<value>`	Specify how much of the available CPU resources a container can use. For instance, if the host machine has two CPUs and you set `--cpus="1.5"` , the container is guaranteed at most one and a half of the CPUs. This is the equivalent of setting `--cpu-period="100000"` and `--cpu-quota="150000"` . Available in Docker 1.13 and higher.
`--cpu-period=<value>`	Specify the CPU CFS scheduler period, which is used alongside `--cpu-quota` . Defaults to 100 micro-seconds. Most users do not change this from the default. If you use Docker 1.13 or higher, use `--cpus` instead.
`--cpu-quota=<value>`	Impose a CPU CFS quota on the container. The number of microseconds per `--cpu-period` that the container is guaranteed CPU access. In other words, `cpu-quota / cpu-period` . If you use Docker 1.13 or higher, use `--cpus` instead.
`--cpuset-cpus`	Limit the specific CPUs or cores a container can use. A comma-separated list or hyphen-separated range of CPUs a container can use, if you have more than one CPU. The first CPU is numbered 0. A valid value might be `0-3` (to use the first, second, third, and fourth CPU) or `1,3` (to use the second and fourth CPU).
`--cpu-shares`	Set this flag to a value greater or less than the default of 1024 to increase or reduce the container's weight, and give it access to a greater or lesser proportion of the host machine's CPU cycles. This is only enforced when CPU cycles are constrained. When plenty of CPU cycles are available, all containers use as much CPU as they need. In that way, this is a soft limit. `--cpu-shares` does not prevent containers from being scheduled in swarm mode. It prioritizes container CPU resources for the available CPU cycles. It does not guarantee or reserve any specific CPU access.

CONTAINERS ADVANCE: CPU

Set container to use 50% of our CPU every second



```
yanivos@ip-10-0-0-6  ~  docker run -it --cpus=".5" ubuntu /bin/bash
```

CONTAINERS IN PRODUCTION

CONS / PROS AND INBETWEEN

And No...deploying your app inside a container - does not change it’s monolith architecture to microservices

CONTAINERS IN PRODUCTION

Containers

Containers are amazing peace of technology but like anything and everything else,
There are no such thing as a free lunch.

There are many benefits that we learned about using containers and how it can make our development / deployment / CI / CD easy and fast but a question should be asked…

What's the catch?

CONTAINERS IN PRODUCTION:PROS

PROS 101

- Containers makes our applications "virtually look" the same on most infrastructures (Physical/Cloud/VM’s)
- Containers makes our application runtime dependencies the developers's responsibility - **splendid**!
- Containers require the application developers to consider application state and persistence.
- Containers, once built, provide a (mostly) consistent behavior between dev, staging, and production environments. Immutable delivery mechanism out of the box
- Blazing fast scaling our ecosystem
- Delivery time - Days and hours becomes minutes / seconds
- Handoff - Developers <> Operators - Wall of confusion ? **breached**!

The above makes using Containers a no brainer for production - But there are flaws we should know about

CONTAINERS IN PRODUCTION:CONS

CONS 101

- Containers do not make your applications more secure.
- Containers do not make your applications more scalable - **This is a common misconception**
- Containers do not make your applications more portable - **Shared / Common libraries in your code?**
- Network - NAT managed by Dockerd is not how we want to work in production

All of the above becomes absolute once we move to K8S and Swarm (new Pros & cons but different...)

NETWORKING

Network - Basics

Intro

One of the reasons Docker containers and services are so powerful is that you can connect them together, or connect them to non-Docker workloads. Docker containers and services do not even need to be aware that they are deployed on Docker, or whether their peers are also Docker workloads or not. Whether your Docker hosts run Linux, Windows, or a mix of the two, you can use Docker to manage them in a platform-agnostic way.

Network - Basics

What are the common network drivers types?

- **Bridge**

  The basic and default driver which is used for standalone containers setup that need to communicate.

- **Overlay**

  Connect multiple docker daemons together and enable swarm (cluster) services to communicate with each other. This can be used to facilitate communication between swarm and standalone container or between two standalone containers on different docker daemons.

- **macVLAN**

  Macvlan network allow us to assign a MAC address to a container for making it appear as physical device on our network. Usually to be used with legacy or HW required product that must have a MAC and being directly connected to the physical network to operate.

Network - Basics

Network Driver Summary

- **Bridge**

  User-defined bridge networks are best when you need multiple containers to communicate on the same Docker host.

- **Overlay**

  are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services - Works with Swarm only

- **macVLAN**

  Macvlan network are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address

NETWORK

Common practice for user defined bridge setup

HANDS ON LAB

Network - Basics

Follow through

Default Bridge network

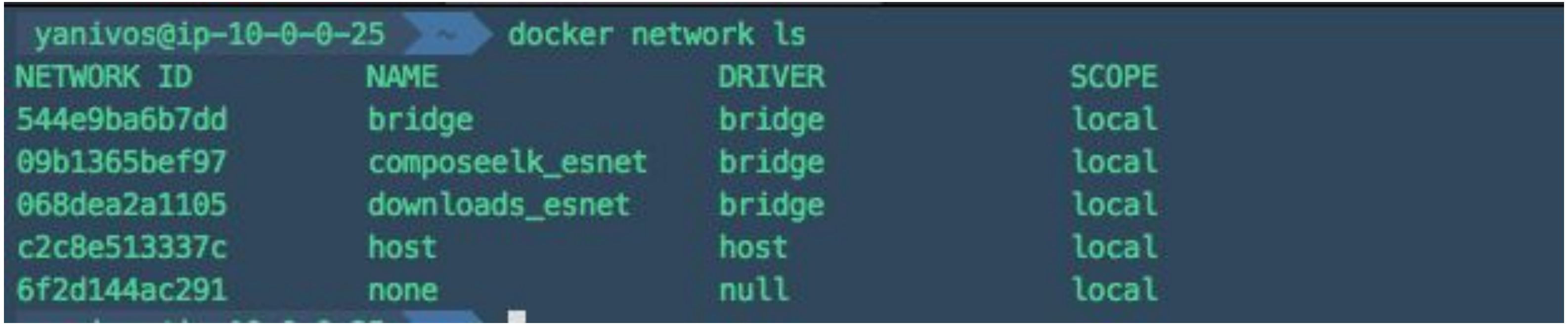
The default bridge network is what Docker setup for us automatically.

It’s a great way to start but this is **not suitable for production use**

Network - Basics

Follow through

We start by inspecting the current network



```
yanivos@ip-10-0-0-25  ~  docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
544e9ba6b7dd        bridge              bridge              local
09b1365bef97        composeelk_esnet    bridge              local
068dea2a1105        downloads_esnet     bridge              local
c2c8e513337c        host                host                local
6f2d144ac291        none                null                local
```

The default **bridge** network is listed, along with host and none. The latter two are not fully-fledged networks, **but are used to start a container connected directly to the Docker daemon host's networking stack, or to start a container with no network devices**. This follow through will connect two containers to the bridge network

Network - Basics

Follow through

Add two new alpine containers with ash as entry point

```
docker run --rm -tdi --name alpine1 alpine ash
docker run --rm -tdi --name alpine2 alpine ash
```

As you recall: The -tdi flags means start the container detached (in the background), interactive (with the ability to type into it), and with a TTY (so you can see the input and output). Because we did not specified any --network flags, the containers connect to the **default bridge network**

Network - Basics

Follow through

Next:

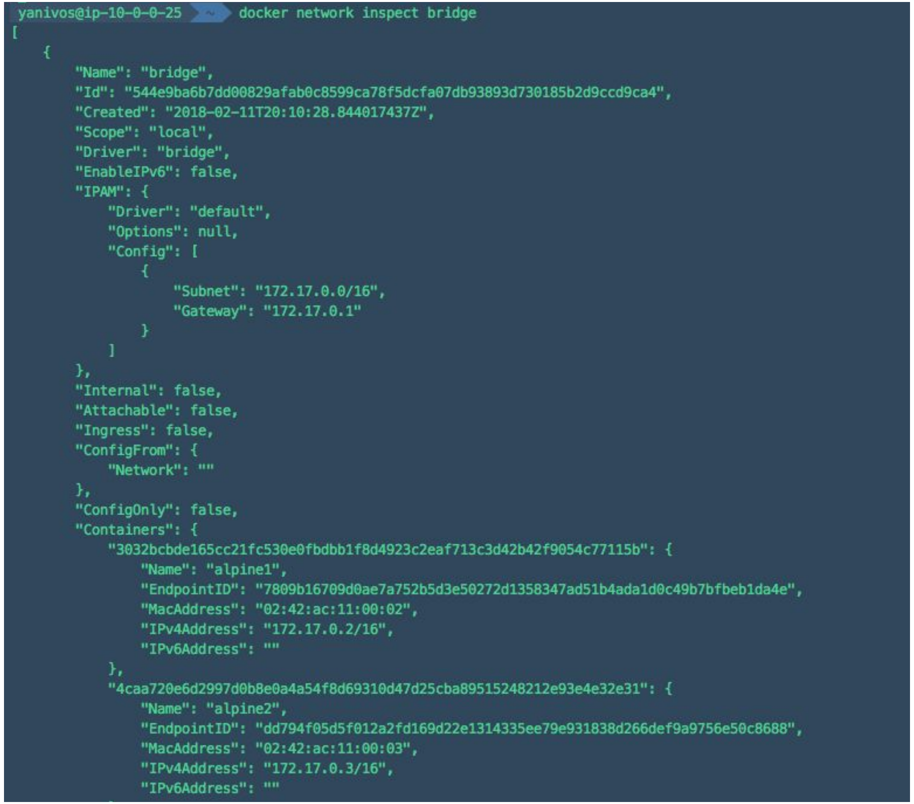
1. Check that the containers are actually running
2. Inspect the network and see what containers are connected to it using

   docker network inspect bridge
3. Connect to one of the Alpine containers using **docker attach** and ping the other container with IP and than with it’s name.

   What happened ?

Network - Basics

Inspect example



```
yanivos@ip-10-0-0-25  ~  docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "544e9ba6b7dd00829afab0c8599ca78f5dcfa07db93893d730185b2d9ccd9ca4",
        "Created": "2018-02-11T20:10:28.844017437Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "3032bcbde165cc21fc530e0fbdbb1f8d4923c2eaf713c3d42b42f9054c77115b": {
                "Name": "alpine1",
                "EndpointID": "7809b16709d0ae7a752b5d3e50272d1358347ad51b4ada1d0c49b7bfbeb1da4e",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            },
            "4caa720e6d2997d0b8e0a4a54f8d69310d47d25cba89515248212e93e4e32e31": {
                "Name": "alpine2",
                "EndpointID": "dd794f05d5f012a2fd169d22e1314335ee79e931838d266def9a9756e50c8688",
                "MacAddress": "02:42:ac:11:00:03",
                "IPv4Address": "172.17.0.3/16",
                "IPv6Address": ""
```

Network - Basics

Follow through

Pinging containers with IP worked while with name it Failed.

Default Bridge driver does not allow name linking / resolution

Network - Basics

Follow through

User Define Bridge network

User define Bridge network provide us with a way to better arrange / build our network topology and communication across containers that connected to the same User Define bridge network along with a DNS Resolution.

Network - Basics

Follow through

Creating user Define Bridge network

CREATE NEW BRIDGE NETWORK

```
docker network create dmz
```

INSPECT NETWORK

```
docker network inspect dmz
```

RUN CONTAINER IN DMZ

```
docker run -tdi --rm --name network_test --network dmz alpine ash
```

INSPECT CONTAINER

```
docker inspect network_test
```

NETWORK BRIDGES

LAB

Network - Basics

User Define Bridge network

LAB

1. Delete the previous containers (stop and then remove)
2. Create a newly user Defined network bridge named “**alpine-net”** and verify creation with **network ls** and than **Inspect** the network to see that no containers are connected
3. Create 4 new alpine containers with -dit and --network to the following network configuration
   a. First two to: alpine-net
   b. 3rd one to the **default bridge**
   c. 4th one to **alpine-net & to the bridge network (trickey…)**
   Tip: network connect...
4. Inspect Network bridge and user defined network

Network - Basics

User Define Bridge network

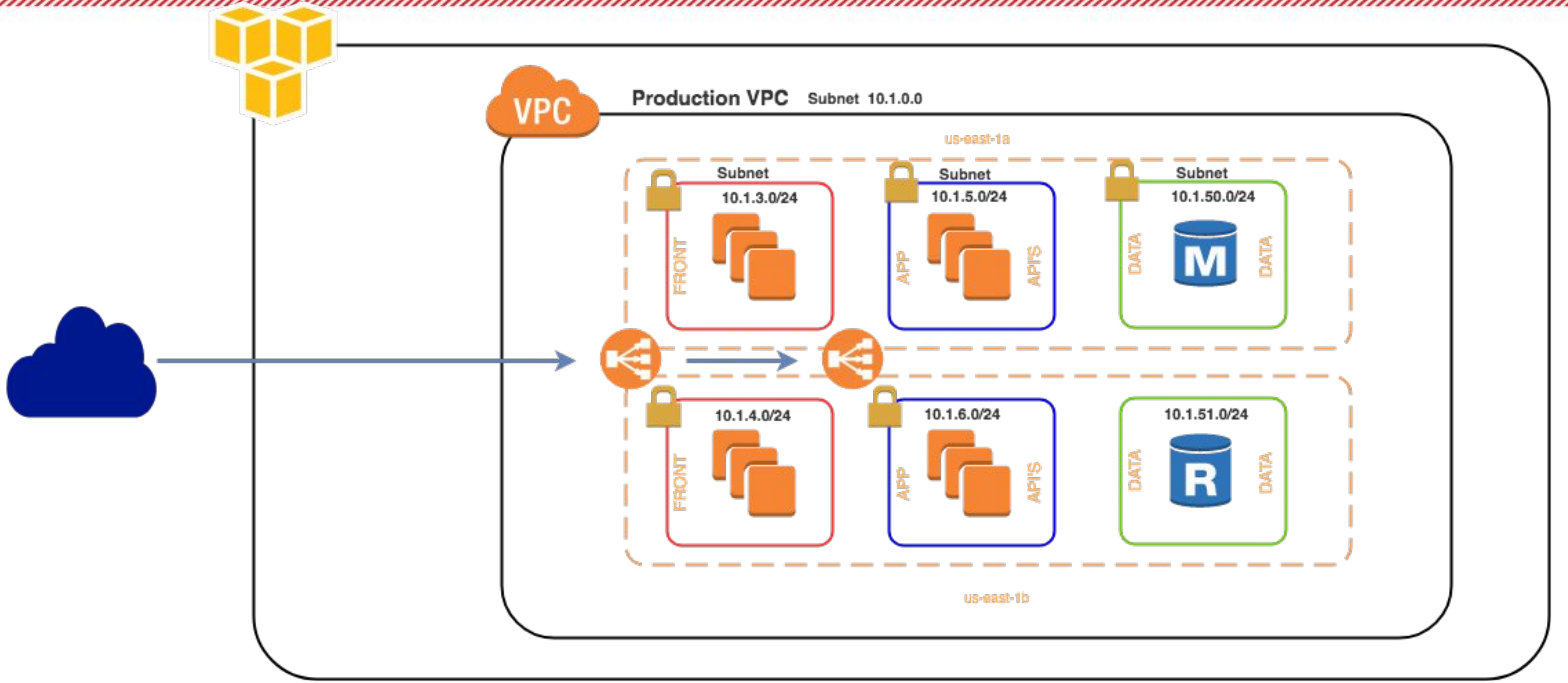
LAB:

5. Connect to alpine1 and try pinging to alpine1,2,3,4 with IP and DNS - What happened ?
6. Connect to alpine4 and try pinging to alpine1,2,3,4 with IP and DNS - What happened ?
7. Why?
8. Stop all containers , Remove them and delete the user defined network you created

NETWORK BRIDGES

LAB: CONNECTING APP AND DB

Network


VPC
Production VPC
Subnet 10.1.0.0
us-east-1a
Subnet
10.1.3.0/24
FRONT
Subnet
10.1.5.0/24
APP
API'S
Subnet
10.1.50.0/24
DATA
M
DATA
10.1.4.0/24
FRONT
10.1.6.0/24
APP
API'S
10.1.51.0/24
DATA
R
DATA
us-east-1b

Network - LAB

APP + DB Network layer and Source code hook

LAB:

1. Create two network bridges
   a. db_layer
   b. app_layer
2. Copy spring-music.jar from `/seminars/docker/artifacts` to a new folder of your choose.
3. Run MYSQL Container as followed and with the following: (1 line)

Network - LAB

```
docker run --rm -itd --name db_mysql -e MYSQL_ALLOW_EMPTY_PASSWORD=yes -e
MYSQL_DATABASE=music wangxian/alpine-mysql
```

And add the following:

4. Mount: `"$(pwd)":/app`
5. Networks: db_layer
6. Expose port: 3306:3306
7. Name: db_mysql

Network - LAB

9. **Inspect db_mysql**
   a. Verify MYSQL is working by connecting to the container and running mysql
   b. Verify a new local folder on your host called mysql created
   c. Inspect network and container that it indeed connected to db_layer bridge
10. **Create new java web application container that will run your local spring-music jar**
    a. Image: `frolvlad/alpine-oraclejdk8:slim`
    b. CMD: `java -jar -Dspring.profiles.active=mysql /source/spring-music.jar`
    c. Mount: `"$(pwd)":/source`
    d. Networks: app_layer & db_layer
    e. Expose port: 8080:8080
    f. Name: web_app

Network - LAB

11. **Validate application is working with the DB container in db_layer**
    a. Connect to db_mysql and run:
        i. “Mysql” and select database “music” and view table “albums”
    b. Browse http://localhost:80 and change a value and than review table “albums” again
        i. issues ?
            1. Check logs using docker logs [container]
            2. Make sure web_app is connected to both db_layer and app_layer
            3. Inspect network and contianer
12. **Once everything is woking -**
    a. stop containers & delete bridges.

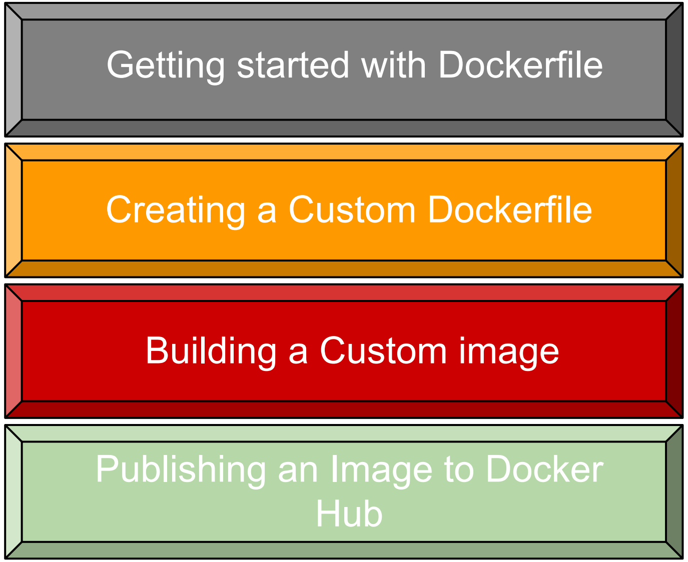
Docker Advanced

Dockerfile - Custom images

INTRO

Module Agenda

“Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build users can create an automated build that executes several command-line instructions in succession “



Getting started with Dockerfile

Creating a Custom Dockerfile

Building a Custom image

Publishing an Image to Docker Hub

Dockerfile

What will we do in this module?

Get our source code into a **custom built image (vs pre-built images)** to share with others

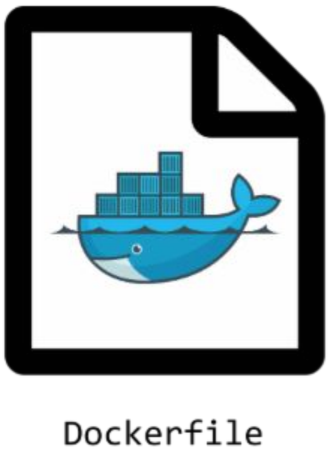
Dockerfile

What is a dockerfile and how it create an Image

Developers use .java or pom file to describe / develop - we use Dockerfile

Dockerfile

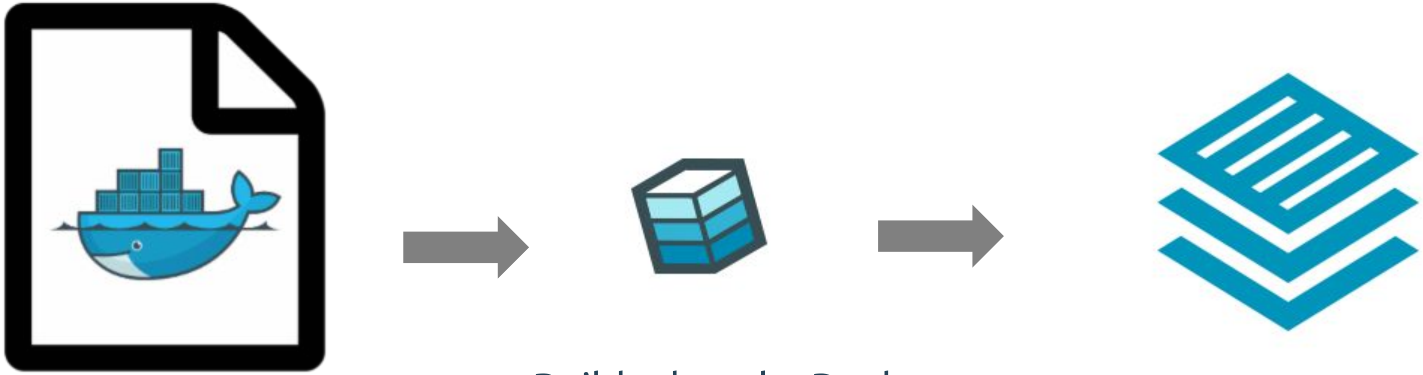
Dockerfile is a … FILE with instructions and descriptions of an image



BUILD	Boot	Run
FROM	WORKDIR	CMD
	USER	ENV
COPY		EXPOSE
ADD		VOLUME
RUN		ENTRYPOINT
ONBUILD		
.dockerignore		

Dockerfile

Dockerfile flow



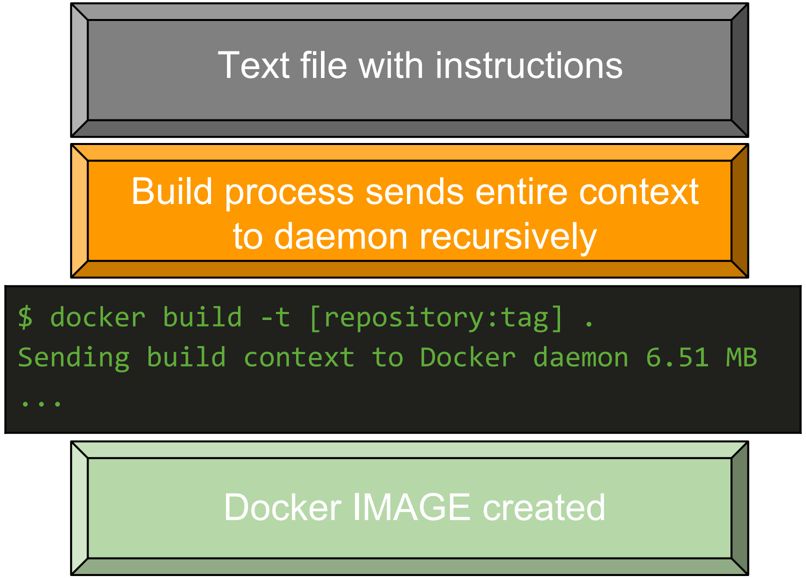
dockerfile

Build - done by Docker daemon and not CLI

image

Dockerfile

Dockerfile flow Overview



Text file with instructions

Build process sends entire context to daemon recursively

```
$ docker build -t [repository:tag] .
Sending build context to Docker daemon 6.51 MB
...
```

Docker IMAGE created

Warning: Do not use your root directory, /, as the PATH as it causes the build to transfer the entire contents of your hard drive to the Docker daemon.

Dockerfile - Example

```
ARG VERSION=latest
# Java8 Alpine Release
FROM frolvlad/alpine-oraclejdk8:slim
ARG VERSION
# RUN will execute shell commands
RUN echo $VERSION > image_version
# Label Use Labels for descriptions and view it with docker inspect
LABEL multi.label1="value1" \
      description="Bug fix x.0 for client y"
# configure WorkDir inside the container
WORKDIR /app
# Mount HOST Folder
VOLUME ["./spring-boot-rest-example/dockerfile/artifact/"]
# Copy Spring Boot File to target
COPY spring-boot-rest-example-0.4.0.war /app/spring-boot-rest-example-0.4.0.war
#Expose Ports - ONLY EXPOSED - IT'S NOT Mapped. -p will be needed on run
EXPOSE 8091
EXPOSE 8090

# The main purpose of a CMD is to provide defaults for an executing container
CMD java -jar /app/spring-boot-rest-example-0.4.0.war -Dspring.profiles.active=test
```

Dockerfile

CMD VS ENTRYPOINT?

```
FROM alpine:latest
CMD ping localhost
```

```
docker build -t playground:latest .
….
docker run -ti playground:latest
PING localhost (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: seq=0 ttl=64 time=0.051 ms
64 bytes from 127.0.0.1: seq=1 ttl=64 time=0.080 ms
# in CMD - Override IS ALLOWED
#### docker run -ti playground:latest [command]
docker run -ti playground:latest hostname
93d4a120e1ff
```

Dockerfile

CREATING A CUSTOM BOOTSPRING

DOCKERFILE

Dockerfile

MAKE SURE YOU CLONED

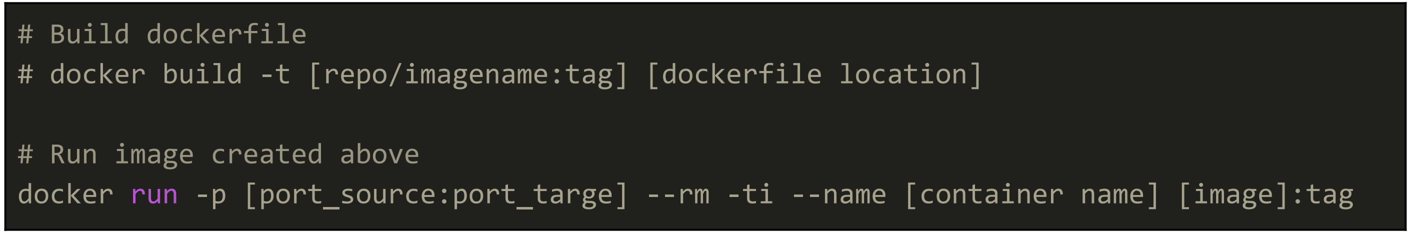
https://github.com/lev-tmp/seminars.git

Dockerfile - LAB

1. Make a new folder in your project directory called `jb_dockerfile`
   a. Copy spring-music.jar from `/seminars/docker/artifacts` to a new folder `jb_dockerfile/artifacts`
   b. Create an empty dockerfile
2. SPEC
   a. From: `frolvlad/alpine-oraclejdk8:slim`
   b. Workdir `/app`
   c. Copy: artifact to `/app`
   d. Expose: `8080`
   e. CMD: `java -jar -Dspring.profiles.active=none spring-music.jar`
3. Build && Run image

Dockerfile - LAB

Building the dockerfile in CLI



```
# Build dockerfile
# docker build -t [repo/imagename:tag] [dockerfile location]

# Run image created above
docker run -p [port_source:port_targe] --rm -ti --name [container name] [image]:tag
```

Dockerfile - LAB

Browse

http://localhost:8080/

Dockerfile - LAB

DOCKER HUB

Push our docker image to docker hub

1. Create new Repo in docker hub
2. Register your newly created repo and login to it in CLI

   “docker login”
3. Push your created image to your repo

   “docker push repo/image:tag”

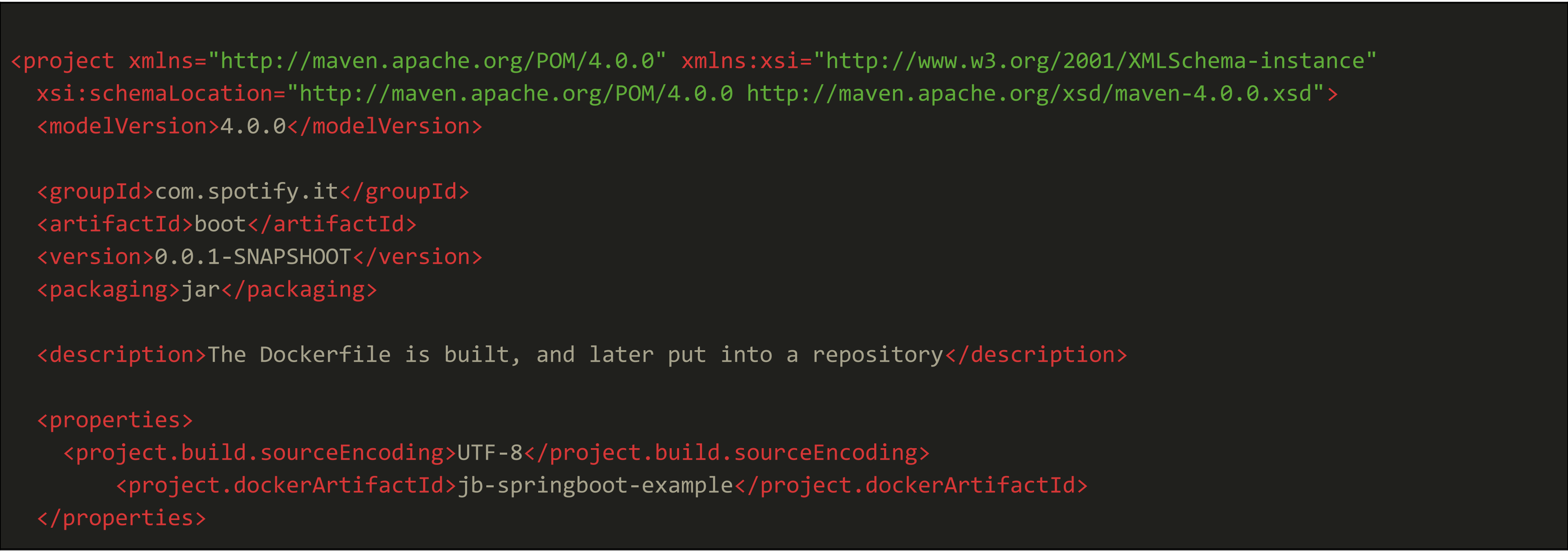
Maven style

Building and Pushing Docker image to automate build process

MAVEN & DOCKER

Using Spotify Maven Plugin, Build , Deploy and Push Docker Image post build becomes extremely easy

Once configured we can run: mvn clean package docker:build



```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.spotify.it</groupId>
  <artifactId>boot</artifactId>
  <version>0.0.1-SNAPSHOOT</version>
  <packaging>jar</packaging>

  <description>The Dockerfile is built, and later put into a repository</description>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.dockerArtifactId>jb-springboot-example</project.dockerArtifactId>
  </properties>
```

```
<build>
                <resources>
                        <resource>
                                <!--  <directory>src/main/resources</directory> -->
                                <directory>artifact</directory>
                                <filtering>true</filtering>
                        </resource>
                </resources>
                        <!-- Docker Build -->
                <plugins>
                        <plugin>
                                <artifactId>maven-war-plugin</artifactId>
                                <version>3.0.0</version>
                        </plugin>
                        <plugin>
                                <groupId>com.spotify</groupId>
                                <artifactId>docker-maven-plugin</artifactId>
                                <version>0.4.10</version>
                                <configuration>
                                        <imageName>yanivomc/${project.dockerArtifactId}</imageName>
                                        <dockerDirectory>dockerfile</dockerDirectory>
                                        <resources>
                                        <resource>
                                                        <targetPath>/</targetPath>
                                                        <directory>${project.build.directory}</directory>
                                                        <include>${project.artifactId}-${project.version}.${project.packaging}</include>
                                        </resource>
                                        </resources>
                                </configuration>
                        </plugin>
                </plugins>
        </build>
</project>
```

Before we continue

DOCKERS Q&A

REVIEW / STRATEGY / CI/CD

Docker Advanced

Docker Compose

Docker Compose

Intro

“ Compose is a tool for defining and running multi-container Docker applications…

…. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration. “

Docker Compose

MODULE AGENDA

Getting started

Docker-compose.yml

Docker compose commands

Docker compose in action

Setting up dev env services

Creating custom docker-compose to manage our services

Docker compose

When should we use it?

Development environment
When you’re developing software, the ability to run a full fledged application (role and all of its dependencies) in an isolated environment and interact with it is crucial. The Compose command line tool can be used to create the environment and interact with it.

Using the Compose file - we can document and configure all of the application (role) depenendencies (DB, Queues, Caches, web services API and many other components) in one of multiple containers per component in a **single command** (docker-compose up)

Compose can provide a convenient way for developers to focus on developing and not on requesting servers or waiting for IT to provide VM’s , EC2’s or physical servers to develop on top.

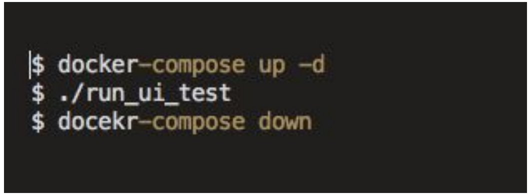
Docker compose

When should we use it?

Automated Tests environment (as part of a ci/cd or standalone)

With compose we can run end-to-end testing that requires a full environment for it to run.
Compose provides a convenient way to create , destroy an isolated testing environments for our test suite.

Vision this:



```
$ docker-compose up -d
$ ./run_ui_test
$ docekr-compose down
```

Docker compose

When should we use it?

Production use….

Possibly but we got Kubernetes for that purpose (and no… we are not learning about K8S today…)

Compose with Swarm

- **Cluster management integrated with Docker Engine:** Use the Docker Engine CLI to create a swarm of Docker Engines where you can deploy application services. You don’t need additional orchestration software to create or manage a swarm.
- **Decentralized design:** Instead of handling differentiation between node roles at deployment time, the Docker Engine handles any specialization at runtime. You can deploy both kinds of nodes, managers and workers, using the Docker Engine. This means you can build an entire swarm from a single disk image.

Compose with Swarm

- **Declarative service model:** Docker Engine uses a declarative approach to let you define the desired state of the various services in your application stack. For example, you might describe an application comprised of a web front end service with message queueing services and a database backend.
- **Scaling:** For each service, you can declare the number of tasks you want to run. When you scale up or down, the swarm manager automatically adapts by adding or removing tasks to maintain the desired state.

Compose with Swarm

- **Desired state reconciliation:** The swarm manager node constantly monitors the cluster state and reconciles any differences between the actual state and your expressed desired state. For example, if you set up a service to run 10 replicas of a container, and a worker machine hosting two of those replicas crashes, the manager creates two new replicas to replace the replicas that crashed. The swarm manager assigns the new replicas to workers that are running and available.
- **Multi-host networking:** You can specify an overlay network for your services. The swarm manager automatically assigns addresses to the containers on the overlay network when it initializes or updates the application.

Compose with Swarm

- **Service discovery:** Swarm manager nodes assign each service in the swarm a unique DNS name and load balances running containers. You can query every container running in the swarm through a DNS server embedded in the swarm.
- **Load balancing:** You can expose the ports for services to an external load balancer. Internally, the swarm lets you specify how to distribute service containers between nodes.

Compose with Swarm

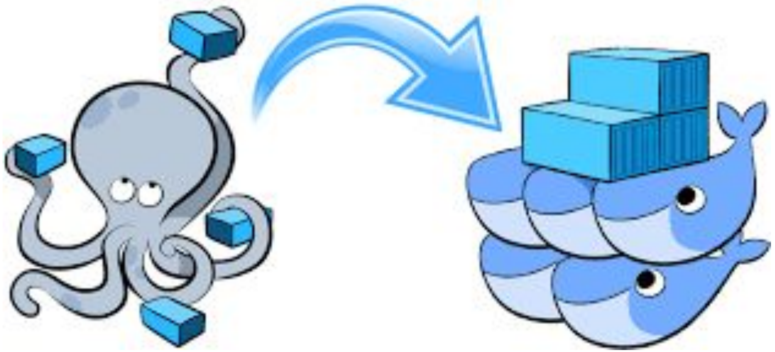
- **Secure by default:** Each node in the swarm enforces TLS mutual authentication and encryption to secure communications between itself and all other nodes. You have the option to use self-signed root certificates or certificates from a custom root CA.
- **Rolling updates:** At rollout time you can apply service updates to nodes incrementally. The swarm manager lets you control the delay between service deployment to different sets of nodes. If anything goes wrong, you can roll-back a task to a previous version of the service.

Docker Compose

Layout

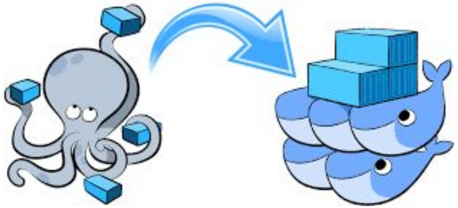
Docker compose

Docker compose manges our application lifecycle



Docker compose

Docker compose manges our application lifecycle

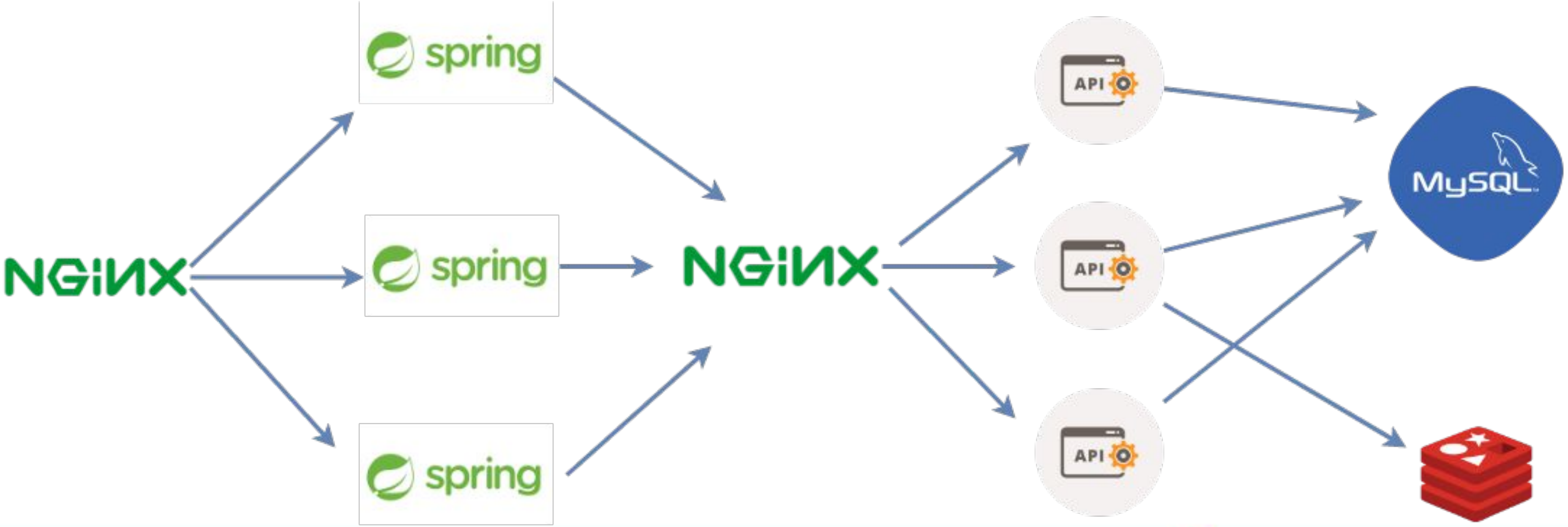


- Start, stop , rebuild of our services
- View status of our running services
- Stream the log output of running services
- Run a one-off command on a service

Docker compose

Why do we need docker compose?

Imagine managing this manually



Docker compose

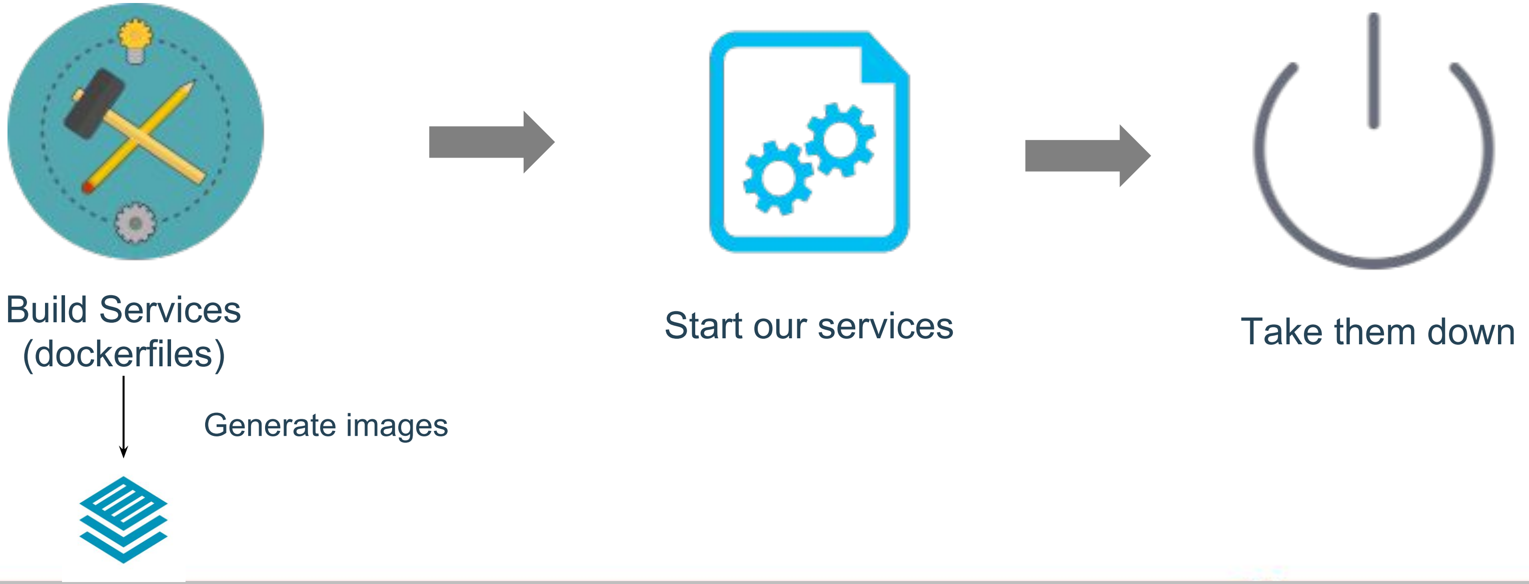
Why do we need docker compose?

Using docker-compose.yml we can define

- Networking
- Dependencies between services
- Environments
- What makes up a role and its components
- Manage each application services we got

Docker compose

Docker compose flow



Build Services (dockerfiles)

Generate images

Start our services

Take them down

Docker compose

Layout

Docker compose is basically 3 steps process

1. First we define our app/role environment with a dockerfile as we did earlier
2. Define the services/components that makes our app/role a whole in our compose file “docker-compose.yml”
3. Run “docker-compose up”
4. Multiple images are up and running

Docker Compose

.yml example

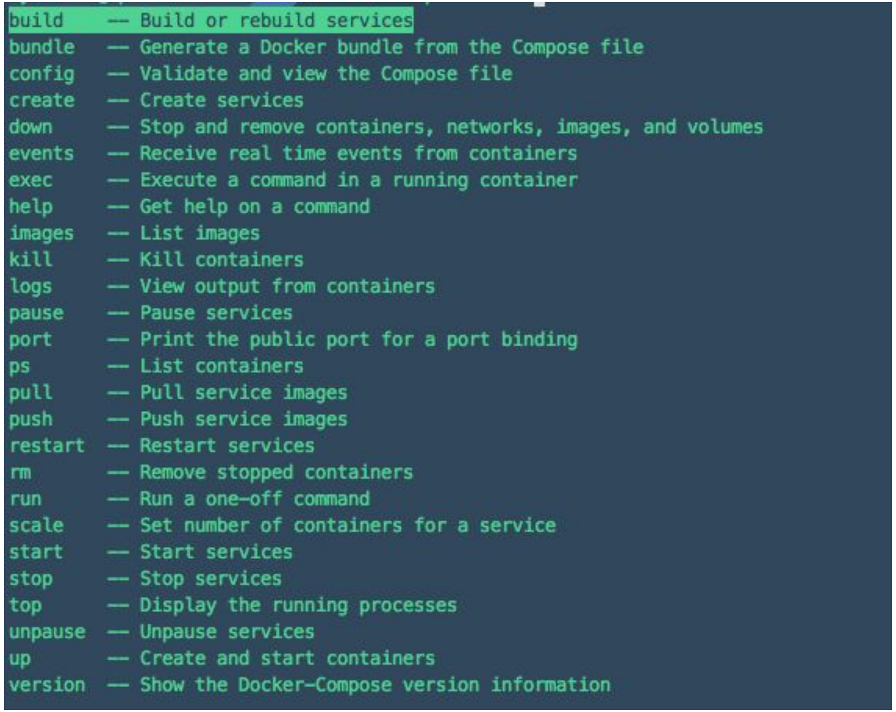
```yaml
version: '3' # Docker-compose yml version for docker compose builder
services: # Here we define our architecture services / roles
 nginx: # or web_role  , api_role , db_mysql , db_redis ....
   build:  # Dockerfile to build or use image: to pull from DockerHub (public/private) or git
     context: . # folder where the dockerfile is located
     dockerfile: #[dockerfile-name]
   ports: # Ports we wish to expose source:target
     - 8080:8080
   volumes: # Which Bind mounts or Volumes to mount source:target
     - type: bind         # Shortsyntax -> ./source:/code
       source: /source
       target: /code
     -type: volume        # Shortsyntax -> logicalVolume01:/var/log
       source: logicalVoloume01
       target: /var/log
   network: # Define netwroks per service / role
     - db_layer
     - app_layer
 services: #.....
   image: redis
networks: # custome configuration per network defined
 db_layer:
 app_layer:

volumes:
 logicalVoloume01 :
```

Compose file version 3 reference

https://docs.docker.com/compose/compose-file/

Docker compose

Common cli for docker compose lifecycle



```
build     —— Build or rebuild services
bundle    —— Generate a Docker bundle from the Compose file
config    —— Validate and view the Compose file
create    —— Create services
down      —— Stop and remove containers, networks, images, and volumes
events    —— Receive real time events from containers
exec      —— Execute a command in a running container
help      —— Get help on a command
images    —— List images
kill      —— Kill containers
logs      —— View output from containers
pause     —— Pause services
port      —— Print the public port for a port binding
ps        —— List containers
pull      —— Pull service images
push      —— Push service images
restart   —— Restart services
rm        —— Remove stopped containers
run       —— Run a one-off command
scale     —— Set number of containers for a service
start     —— Start services
stop      —— Stop services
top       —— Display the running processes
unpause   —— Unpause services
up        —— Create and start containers
version   —— Show the Docker-Compose version information
```

Docker Compose

Follow Through

Docker compose

Simple Project with docker compose

Project Description:
We will build and run an application with two roles.

Front: Web - our spring-music spring app using the dockerfile we created

Backend: db_sql - mysql db using the dockerfile we created

Docker compose

Step 1 - prep

- Create new folder name: jb_dockercompose
- Copy your mymusic-spring dockerfile we created earlier
    - or from seminars/docker/playground/labs/labdockercompose/roles/web/dockerfile
        - **Into: jb_dockercompose/roles/web/dockerfile**
- Update the location of the artifact spring-music.jar in the dockerfile
    - You can use the one in: seminars/docker/artifacts/spring-music.jar

Docker compose

Step 2 - WEB Roles - Dockerfile

```
FROM frolvlad/alpine-oraclejdk8:slim

WORKDIR /code

EXPOSE 8080

CMD java -jar -Dspring.profiles.active /code/spring-music.jar
```

Docker compose

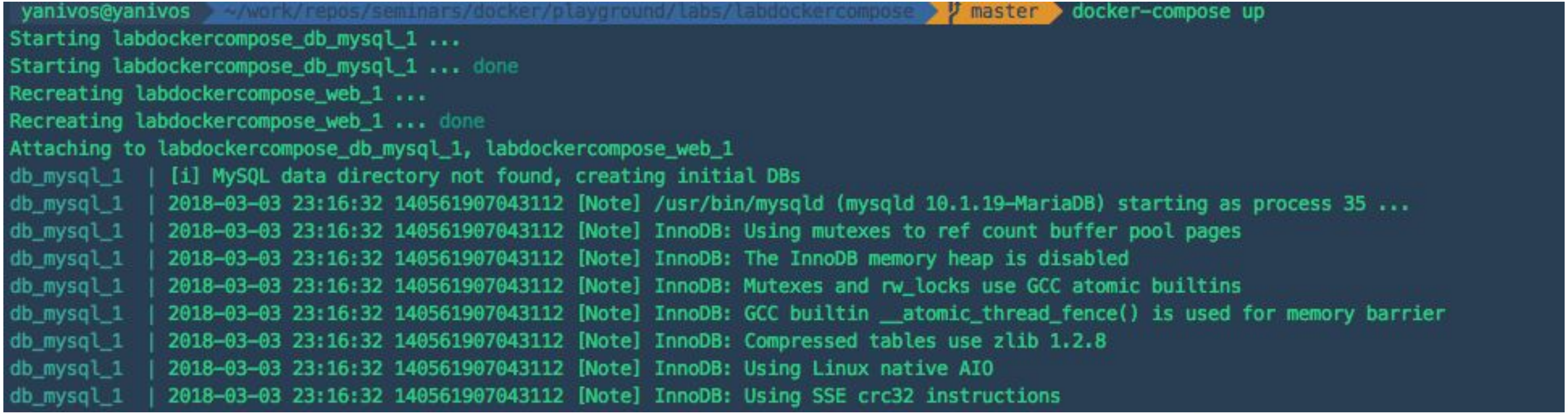
```
#NOTE THAT THERE ARE ERRORS IN THE YML FILE FOR YOU TO FIX
version: '3' # Docker-compose yml version for docker compose builder
services:
 web:
  build: ./roles/web/
  depends_on:
    - db_mysql
……
# View the original file in
seminars/docker/playground/labs/labdockercompose/docker-compose.yml
```

Docker compose

Step 4 - Try Building and running the app with compose

Run docker-compose build and than docker-compose up and check logs:

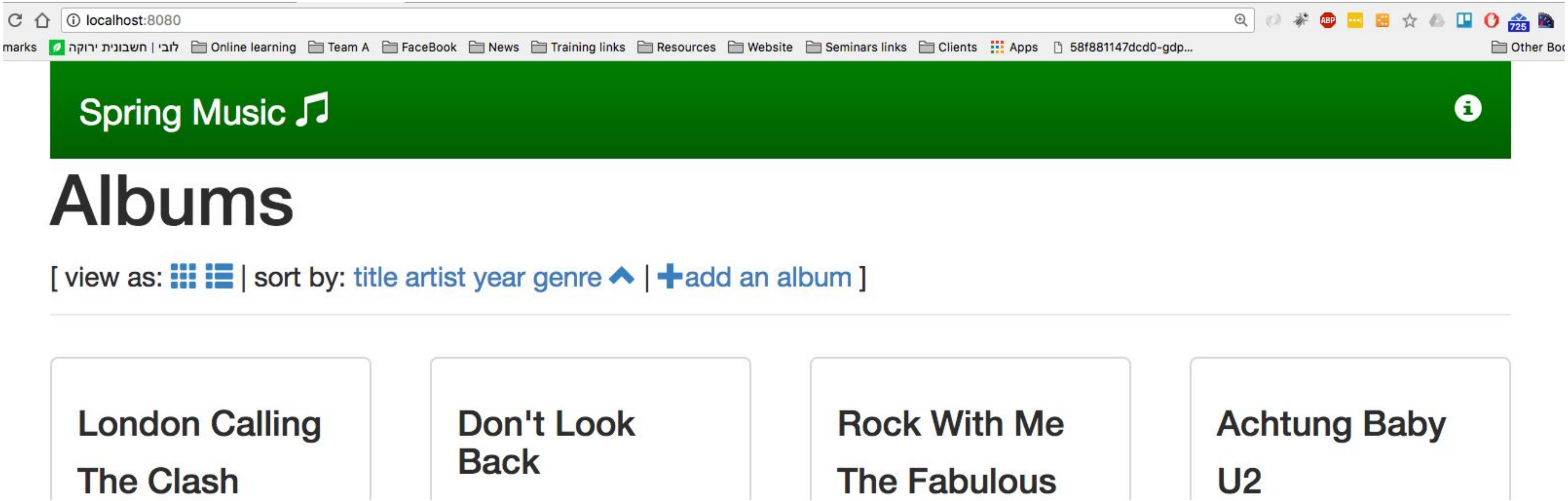
- docker-compose logs -f
- docker-compose ps
- Try fixing the errors and than run it again until you see the below and able to Reach http://localhost:[port]



```
 yanivos@yanivos  ~/work/repos/seminars/docker/playground/labs/labdockercompose  master  docker-compose up
Starting labdockercompose_db_mysql_1 ...
Starting labdockercompose_db_mysql_1 ... done
Recreating labdockercompose_web_1 ...
Recreating labdockercompose_web_1 ... done
Attaching to labdockercompose_db_mysql_1, labdockercompose_web_1
db_mysql_1   | [i] MySQL data directory not found, creating initial DBs
db_mysql_1   | 2018-03-03 23:16:32 140561907043112 [Note] /usr/bin/mysqld (mysqld 10.1.19-MariaDB) starting as process 35 ...
db_mysql_1   | 2018-03-03 23:16:32 140561907043112 [Note] InnoDB: Using mutexes to ref count buffer pool pages
db_mysql_1   | 2018-03-03 23:16:32 140561907043112 [Note] InnoDB: The InnoDB memory heap is disabled
db_mysql_1   | 2018-03-03 23:16:32 140561907043112 [Note] InnoDB: Mutexes and rw_locks use GCC atomic builtins
db_mysql_1   | 2018-03-03 23:16:32 140561907043112 [Note] InnoDB: GCC builtin __atomic_thread_fence() is used for memory barrier
db_mysql_1   | 2018-03-03 23:16:32 140561907043112 [Note] InnoDB: Compressed tables use zlib 1.2.8
db_mysql_1   | 2018-03-03 23:16:32 140561907043112 [Note] InnoDB: Using Linux native AIO
db_mysql_1   | 2018-03-03 23:16:32 140561907043112 [Note] InnoDB: Using SSE crc32 instructions
```

Docker compose

Browse http://localhost:8080



Spring Music ♫

Albums

[view as: | sort by: title artist year genre | +add an album]

London Calling
The Clash

Don't Look Back

Rock With Me
The Fabulous

Achtung Baby
U2

Docker compose

CLEAN UP

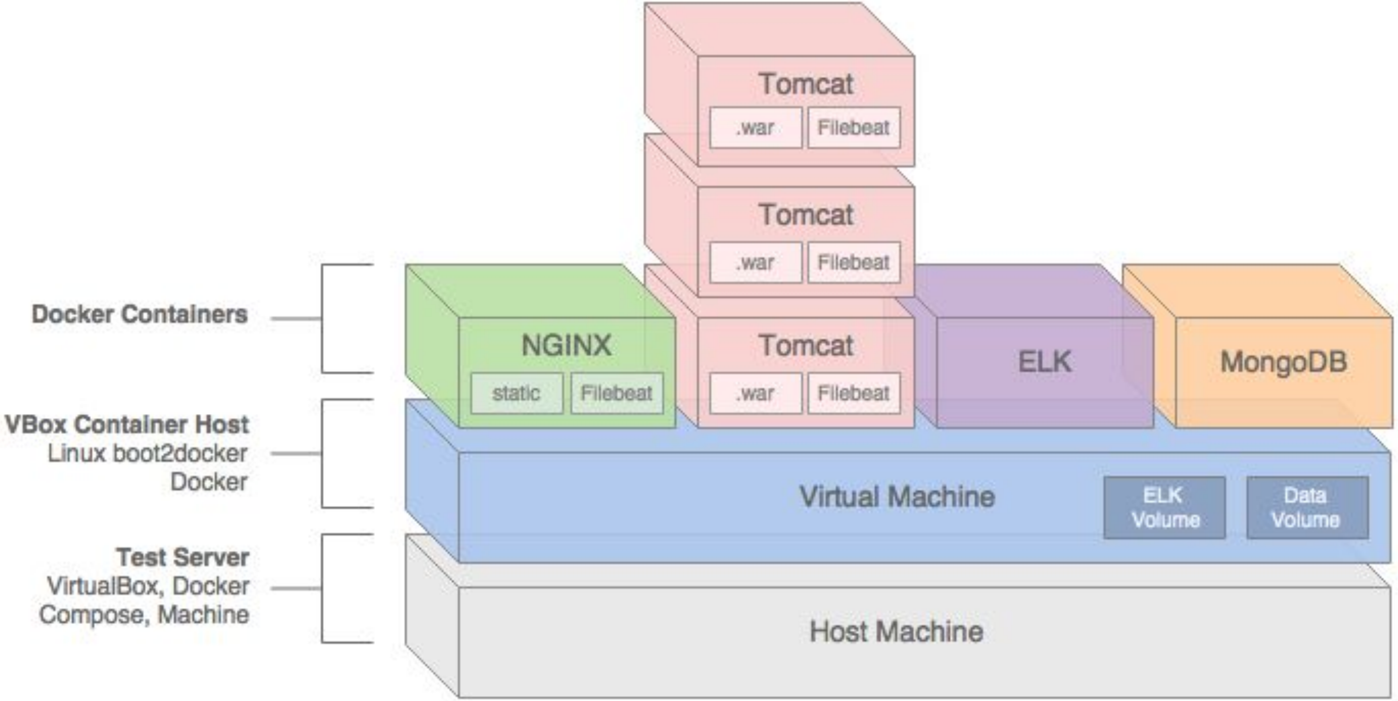
```
docker-compose rm -f -v -s
```

Docker compose

LETS GET CRAZY

Full fledged docker compose architecture project

Docker compose



Docker compose

Introduction

This tutorial demonstrates how to build, deploy, and monitor a Java Spring web application, hosted on Apache Tomcat, load-balanced by NGINX, monitored by ELK, and all containerized with Docker.

Application Architecture

The Java Spring Music application uses the following technologies: Java 8, Spring Framework, NGINX, Apache Tomcat, MongoDB, and the ELK Stack with Filebeat.

Docker compose

NGINX

To increase the application's performance, the application's static content, including CSS, images, JavaScript, and HTML files, is hosted by NGINX. To further increase application performance, NGINX is configured for browser caching of the static content

Tomcat

The application's WAR file is hosted by Apache Tomcat. Requests for non-static content are proxied through NGINX on the front-end, to a set of three load-balanced Tomcat instances on the back-end.

Docker compose

MongoDB

The Spring Music application was designed to work with a number of data stores, including MySQL, Postgres, Oracle, MongoDB, Redis, and H2, an in-memory Java SQL database. Given the choice of both SQL and NoSQL databases, we will select MongoDB.

The Spring Music application, hosted by Tomcat, will store and modify record album data in a single instance of MongoDB. MongoDB will be populated with a collection of album data from a JSON file, when the Spring Music application first creates the MongoDB database instance.

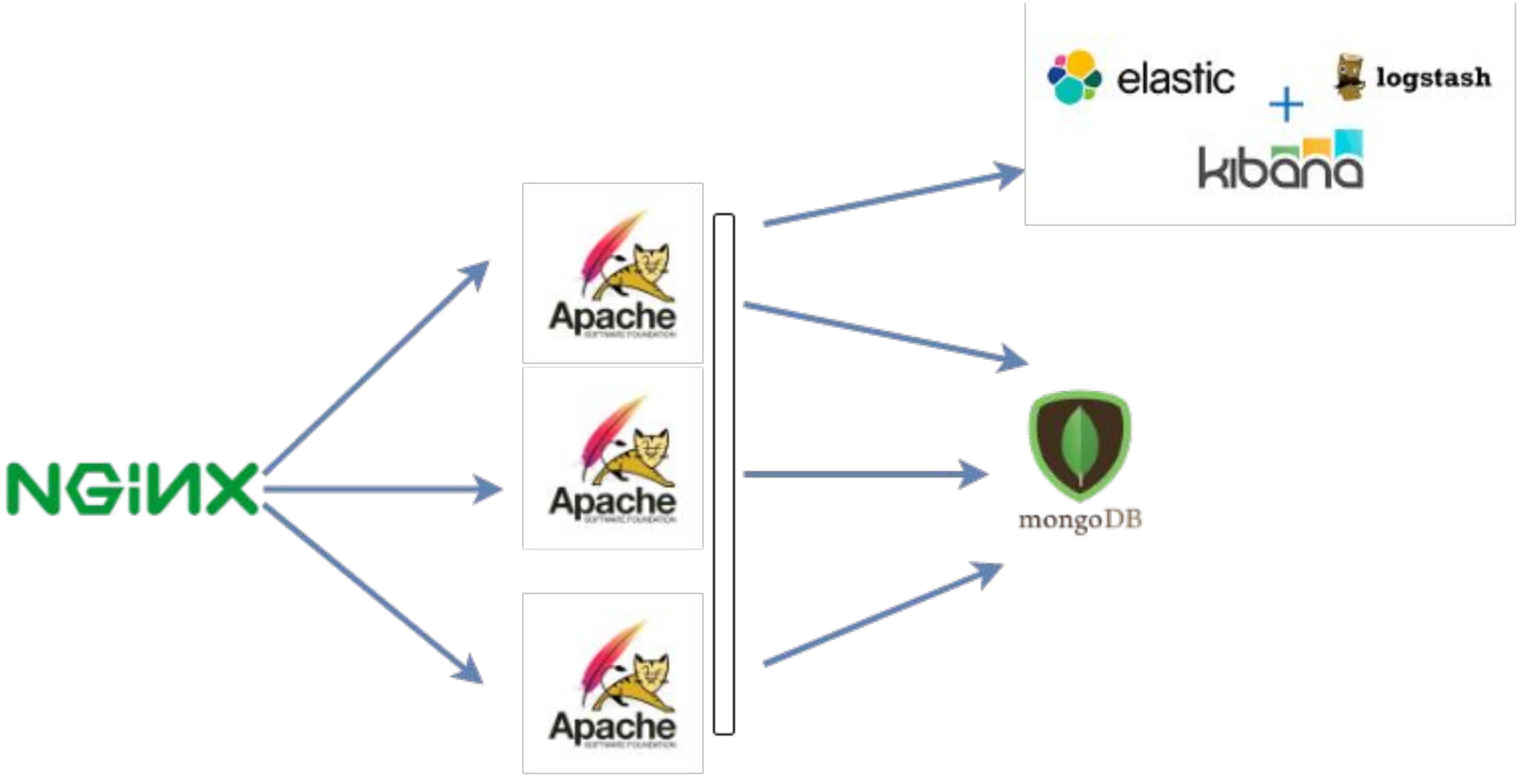
ELK

Lastly, the ELK Stack with Filebeat, will aggregate both Docker and Java Log4j log entries, providing debugging and analytics to our demonstration. A similar method for aggregating logs, using Logspout instead of Filebeat

Docker compose

PROJECT ARCHITECTURE

Docker compose



Docker compose

LET'S DIVE IN

Docker compose

PART 1

BUILDING OUR ELK STACK

Docker compose

ELK

Installing ELK on Docker….
Not a scenario that we use to see - Why ?

One of the reasons for this could be a contradiction between what is required from a data pipeline architecture — persistence, robustness, security — and the ephemeral and distributed nature of Docker. Having said that, and as demonstrated in the instructions below — Docker can be an extremely easy way to set up the stack.

Docker compose

ELK

Using Docker compose to build:

Elasticsearch | Logstash | Kibana

- 3 Docker containers
- Port forwarding
- Data volume for persisting Elasticsearch Data

Docker compose

ELK

Folder Layout for our project

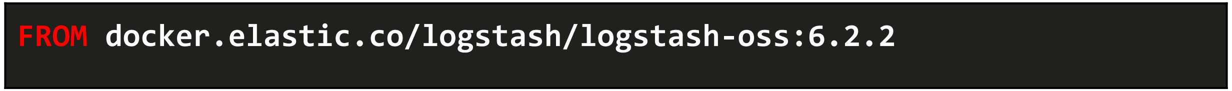
- Docker-elk
  - elasticsearch
    - config
  - extensions
    - logspout # Is a plugin that collects all Docker logs using the Docker logs API, and forwards them to Logstash without any additional configuration.
  - kibana
    - config
  - logstash
    - config
    - pipeline

Docker compose

Elasticsearch | Logstash | Kibana

Architecture:

- Logstash + Kibana need to communicate with ES using port 9200
- **LOGSTASH** dockerfile need to use the following configuration in his root folder:



```
FROM docker.elastic.co/logstash/logstash-oss:6.2.2
```

Docker compose

Elasticsearch | Logstash | Kibana

- Kibana dockerfile need to use the following configuration in his root folder:

```
FROM docker.elastic.co/kibana/kibana-oss:6.2.2
```

- Elastic dockerfile need to use the following configuration in his root folder:

```
FROM docker.elastic.co/elasticsearch/elasticsearch-oss:6.2.2
```

Docker compose

Elasticsearch | Logstash | Kibana

- Docker-compose -
- Fix the file base on the next slide configuration

```
Located: seminars/docker/dockercompose/elk/docker-compose.yml
```

Docker compose

Elasticsearch | Logstash | Kibana

- Add:
  - Elasticsearch:
    - Fix Volume issue if on windows (note the :ro at the end… what is it ?)
    - PORTS: 9200 and 9300
    - Networks: elk
  - LogStash
    - PORTS: 5000
    - Networks : elk
    - Depends_on: elasticsearch
  - Kibana
    - PORTS: 5601
    - Networks: elk
    - Depends_on: elasticsearch

Docker compose

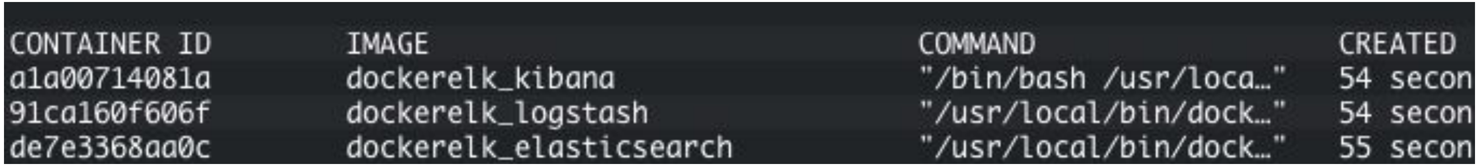
START YOUR ENGINES!

Docker compose

Try Building and running the app with compose

Run “ docker-compose up “ and start verifying the installation:

- Start with “docker ps “



```
CONTAINER ID        IMAGE                          COMMAND                    CREATED
a1a00714081a        dockerelk_kibana               "/bin/bash /usr/loca…"     54 secon
91ca160f606f        dockerelk_logstash             "/usr/local/bin/dock…"     54 secon
de7e3368aa0c        dockerelk_elasticsearch        "/usr/local/bin/dock…"     55 secon
```

- You’ll notice that ports on my localhost have been mapped to the default ports used by Elasticsearch (9200/9300), Kibana (5601) and Logstash (5000/5044).

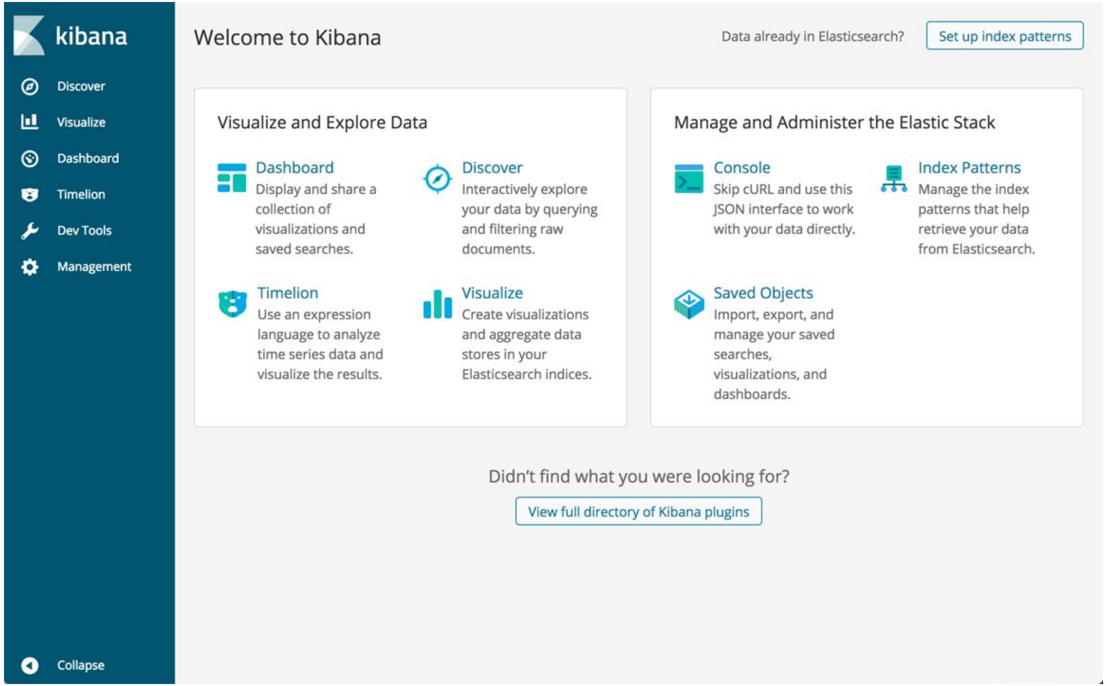
Docker compose

Build the first index

```
curl -XPOST -D-
'http://localhost:5601/api/saved_objects/index-pattern'
\
    -H 'Content-Type: application/json' \
    -H 'kbn-version: 6.2.2' \
    -d
'{"attributes":{"title":"logstash-*","timeFieldName":"@t
imestamp"}}'
```

Docker compose

Browse Kibana:
http://localhost:5601



Welcome to Kibana

Data already in Elasticsearch?

Visualize and Explore Data

Dashboard
Display and share a collection of visualizations and saved searches.

Discover
Interactively explore your data by querying and filtering raw documents.

Timelion
Use an expression language to analyze time series data and visualize the results.

Visualize
Create visualizations and aggregate data stores in your Elasticsearch indices.

Manage and Administer the Elastic Stack

Console
Skip cURL and use this JSON interface to work with your data directly.

Index Patterns
Manage the index patterns that help retrieve your data from Elasticsearch.

Saved Objects
Import, export, and manage your saved searches, visualizations, and dashboards.

Didn't find what you were looking for?

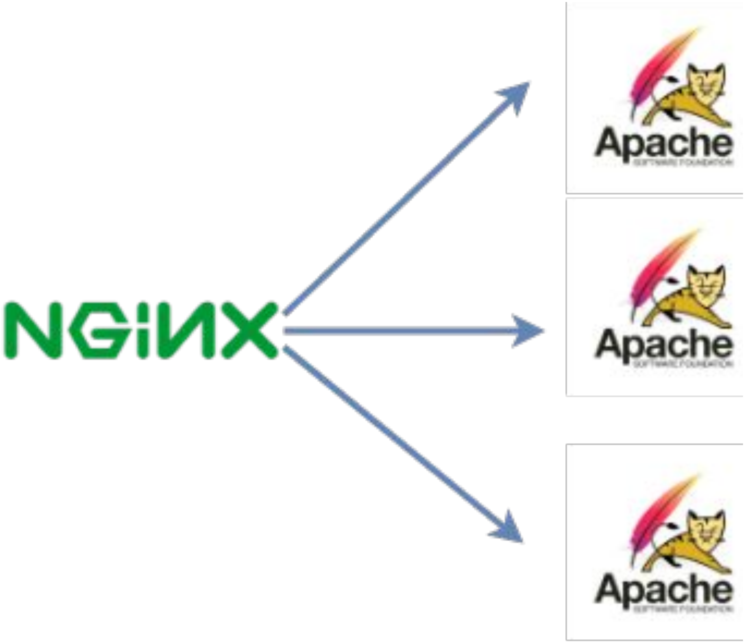
Docker compose

PART 2

BUILDING OUR APP STACK (NGINX, TOMCAT, MONGODB)

Docker compose

LB NGINX AND DOCKER - UPSTREAMS



Docker compose

LB NGINX AND DOCKER

To configure our NGINX to LB our containers we first need to configure it with multiple upstreams

```
# Based on http://nginx.org/en/docs/http/load_balancing.html
upstream backend {
 #ip_hash;
 server music_app_1:8080;
 server music_app_2:8080;
 server music_app_3:8080;
}
```

Docker compose

BUILDING OUR APP STACK (NGINX, TOMCAT, MONGODB)

CODE REVIEW IN :

seminars/docker/playground/labs/music/

Docker compose

Building our stack

```
docker-compose build

docker-compose up

# What happened when we ran it?

Try:

docker logs proxy
```

Docker compose

Scale up WEB before nginx start

```
docker-compose up --scale app=3

# LB Works!
```

Docker compose

Check logs in ELK

http://localhost:5601/

Jenkins and Docker

Jenkins and Docker

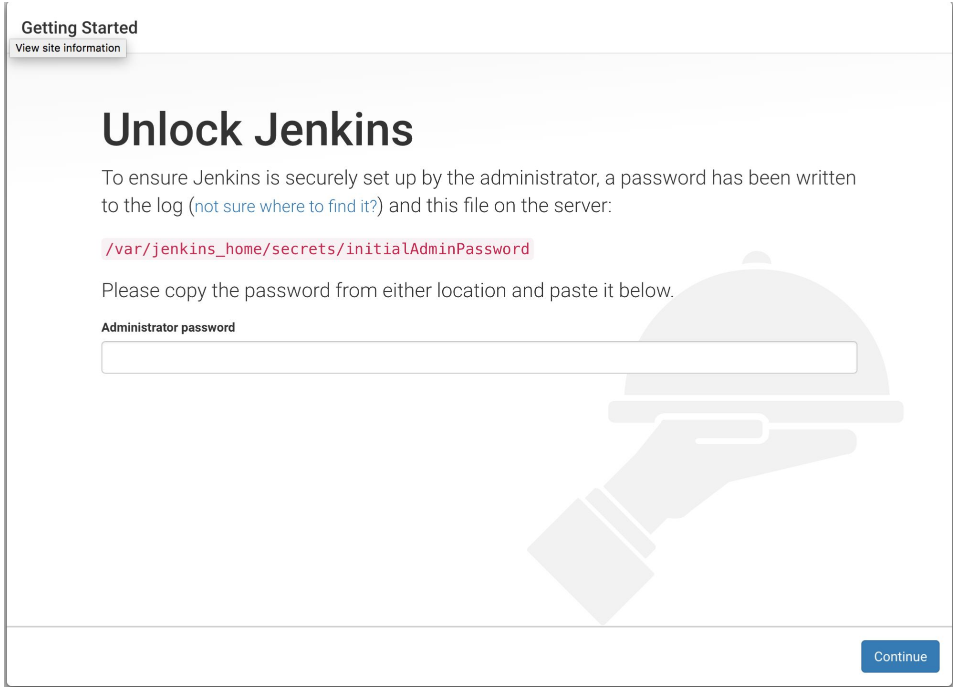
Installing Jenkins

```
docker run \ -u root \ --rm \ -d \ -p 8080:8080 \ -p
50000:50000 \ -v jenkins-data:/var/jenkins_home \ -v
/var/run/docker.sock:/var/run/docker.sock \
jenkinsci/blueocean
```

Jenkins and Docker

Browse: http://localhost:8080

```
For password run:

Docker logs on your container
```



Getting Started

View site information

Unlock Jenkins

To ensure Jenkins is securely set up by the administrator, a password has been written to the log (not sure where to find it?) and this file on the server:

`/var/jenkins_home/secrets/initialAdminPassword`

Please copy the password from either location and paste it below.

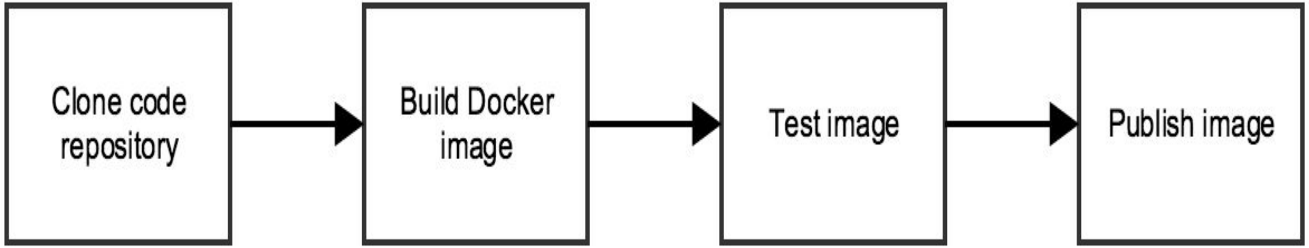
Administrator password

Jenkins and Docker

Install plugin: CloudBees Docker Build and Publish plugin

Jenkins and Docker

Build pipeline



Jenkins and Docker

Writing jenkinsfile

```
File Location on repo:

https://github.com/yanivomc/jenkinsdocker.git
```

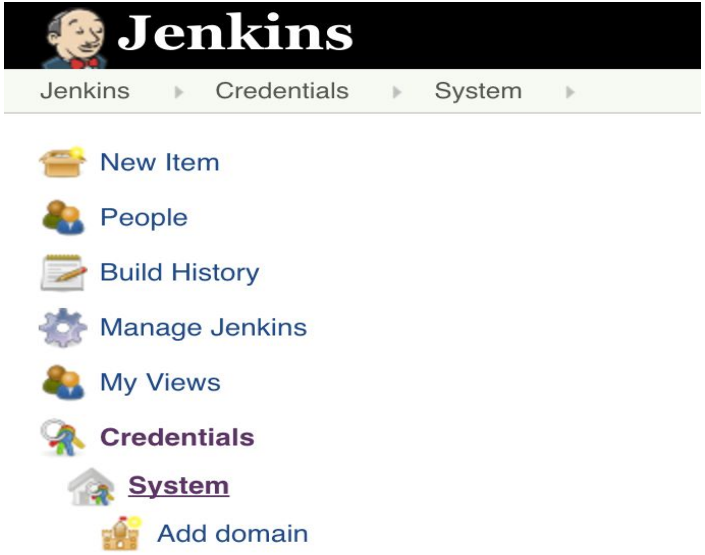
CONFIGURING DOCKER HUB WITH JENKINS

To store the Docker image resulting from our build, we'll be using Docker Hub.

We'll need to give Jenkins access to push the image to Docker Hub. For this, we'll create *Credentials* in Jenkins, and refer to them in the Jenkinsfile.

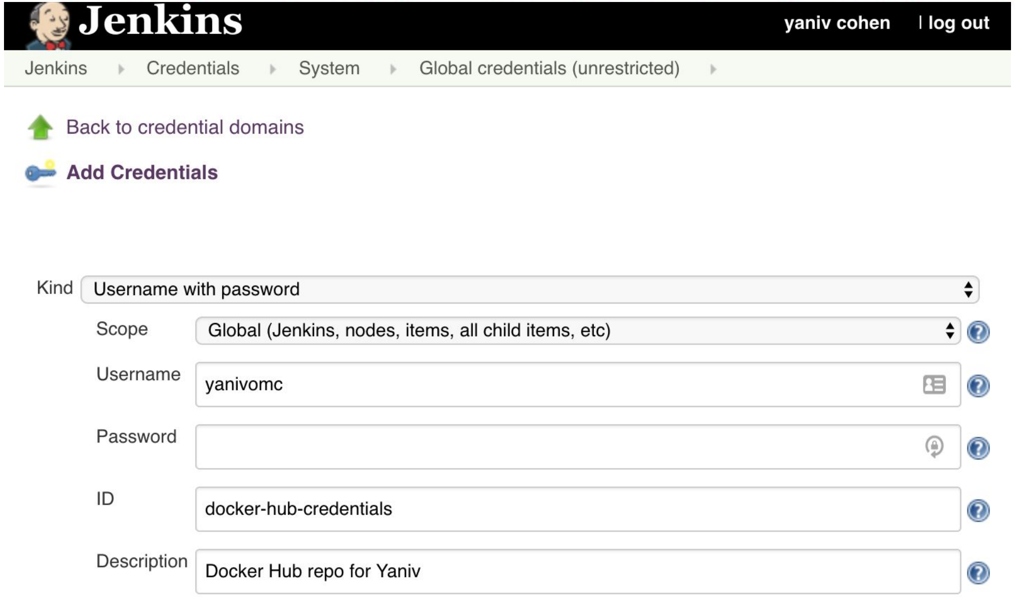
As you might have noticed in the above Jenkinsfile, we're using **docker.withRegistry** to wrap the **app.push** commands - this instructs Jenkins to log in to a specified registry with the specified credential id (docker-hub-credentials).

CONFIGURING DOCKER HUB WITH JENKINS



On the Jenkins front page, click on **Credentials** -> **System** -> **Global credentials -> Add Credentials**

CONFIGURING DOCKER HUB WITH JENKINS



Kind: Username with password

Scope: Global (Jenkins, nodes, items, all child items, etc)

Username: yanivomc

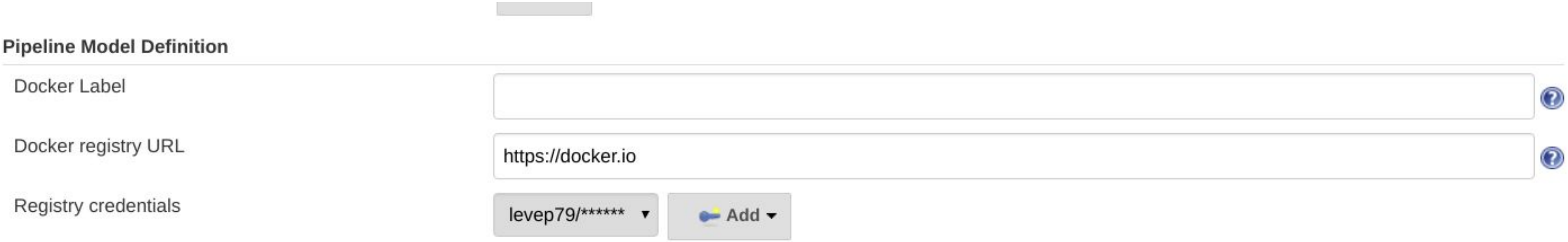
Password:

ID: docker-hub-credentials

Description: Docker Hub repo for Yaniv

Add your Docker Hub credentials as the type **Username with password**, with the ID **docker-hub-credentials that we used in JenkinsFile**

CONFIGURING DOCKER HUB WITH JENKINS



Pipeline Model Definition

Docker Label

Docker registry URL: https://docker.io

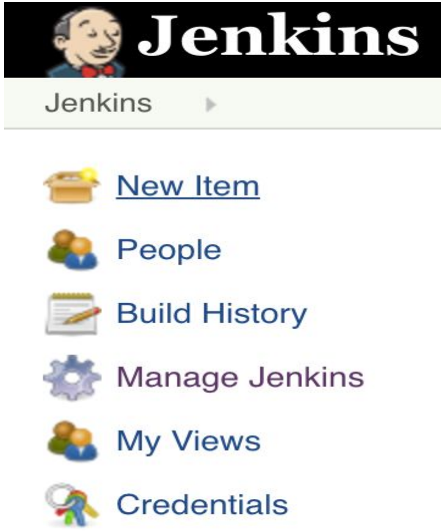
Registry credentials: levep79/****** ▾

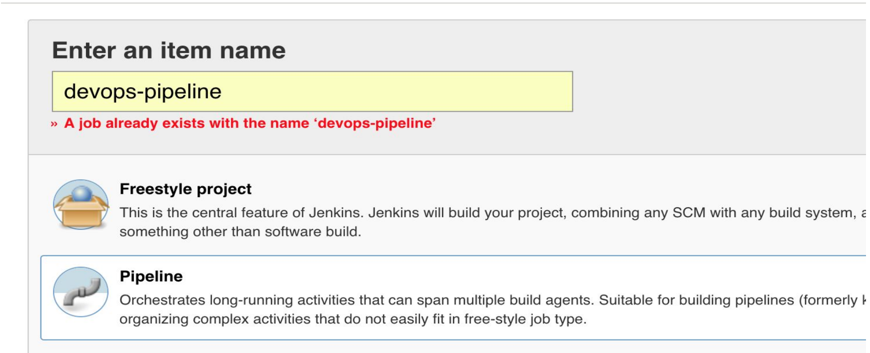
Manage Jenkins -> Configure System

Add Docker Registry and Registry credentials

CREATING A JOB IN JENKINS

The final thing we need to tell Jenkins is how to find our repository. We'll create a Pipeline job, and point Jenkins to use a Jenkinsfile in our repository.





Enter an item name

devops-pipeline

» A job already exists with the name 'devops-pipeline'

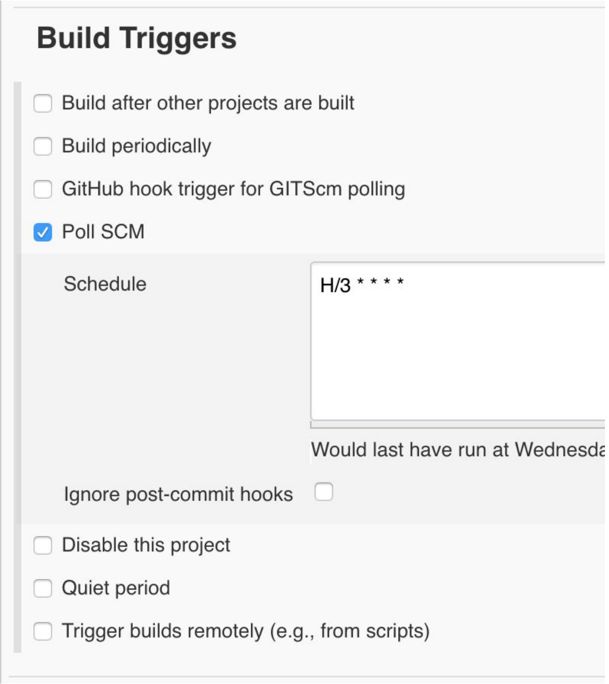
Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, a
something other than software build.

Pipeline
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly k
organizing complex activities that do not easily fit in free-style job type.

Click on **New Item** on the Jenkins front page.

CREATING A JOB IN JENKINS

Configure the Build trigger (Poll or Hook)



Build Triggers

□ Build after other projects are built

□ Build periodically

□ GitHub hook trigger for GITScm polling

☑ Poll SCM

Schedule: H/3 * * * *

Would last have run at Wednesda

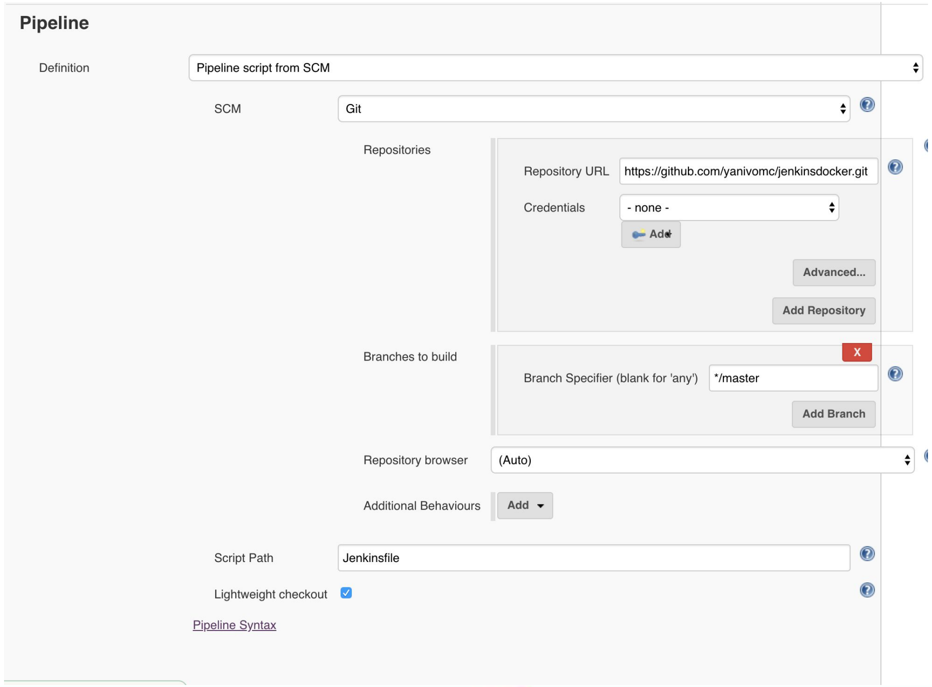
Ignore post-commit hooks □

□ Disable this project

□ Quiet period

□ Trigger builds remotely (e.g., from scripts)

Configure the pipeline to point our Repo and JenkinsFile location



Pipeline

Definition: Pipeline script from SCM

SCM: Git

Repositories

Repository URL: https://github.com/yanivomc/jenkinsdocker.git

Credentials: - none -

Branches to build

Branch Specifier (blank for 'any'): */master

Repository browser: (Auto)

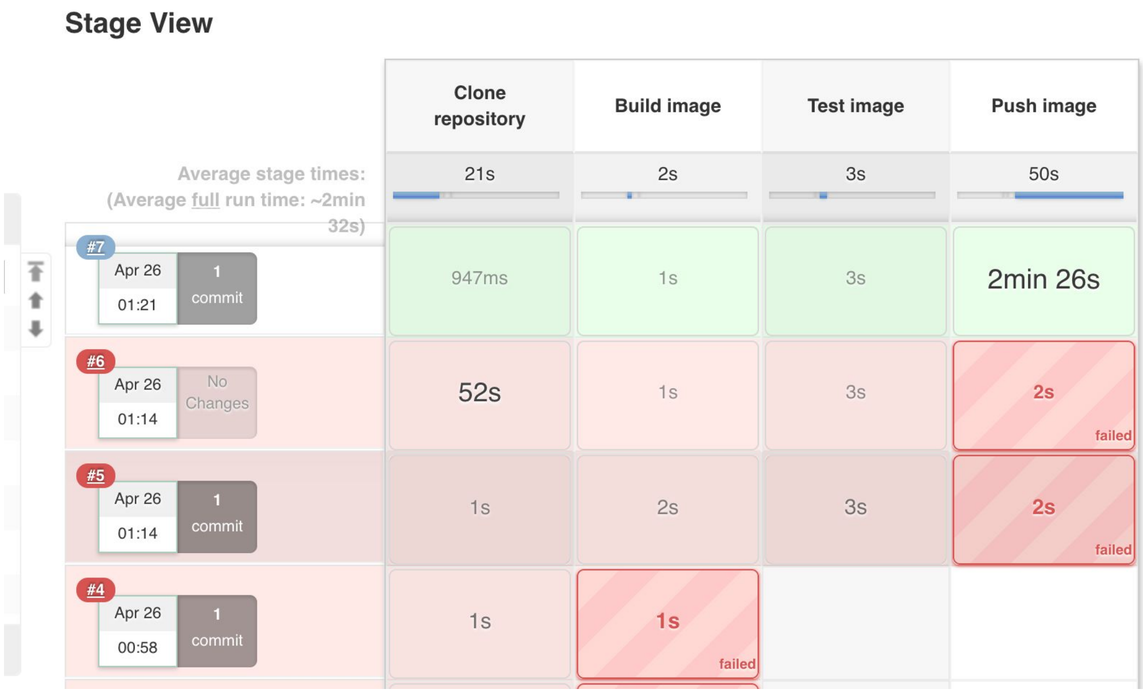
Additional Behaviours: Add

Script Path: Jenkinsfile

Lightweight checkout ☑

Running the job

Running our pipeline - Clone > Build > Test > Push image with tag latest and build number

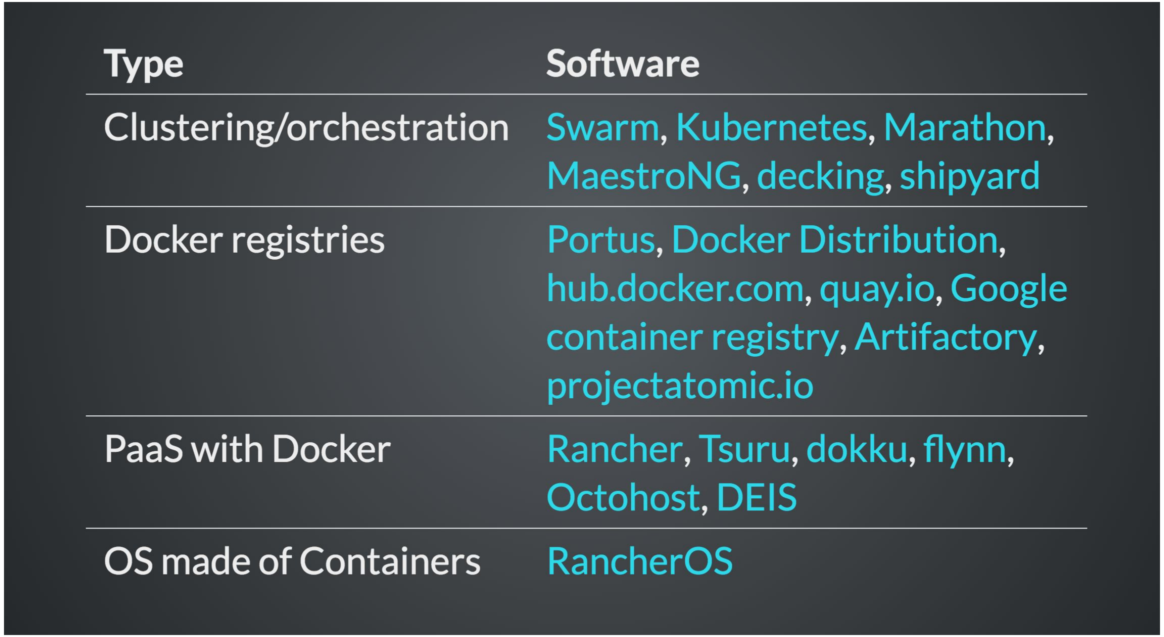


Stage View

	Clone repository	Build image	Test image	Push image
Average stage times: (Average full run time: ~2min 32s)	21s	2s	3s	50s
#7 Apr 26 01:21 1 commit	947ms	1s	3s	2min 26s
#6 Apr 26 01:14 No Changes	52s	1s	3s	2s failed
#5 Apr 26 01:14 1 commit	1s	2s	3s	2s failed
#4 Apr 26 00:58 1 commit	1s	1s failed		

Where do we go next?

Where to go next ?



Type	Software
Clustering/orchestration	Swarm, Kubernetes, Marathon, MaestroNG, decking, shipyard
Docker registries	Portus, Docker Distribution, hub.docker.com, quay.io, Google container registry, Artifactory, projectatomic.io
PaaS with Docker	Rancher, Tsuru, dokku, flynn, Octohost, DEIS
OS made of Containers	RancherOS

QUESTIONS?

END

DevOps Course