DockerWS

Docker Workshop ®

Version 3.0

Sela College

This course material has been prepared by:

Sela Software Labs Ltd.

14-18 Baruch Hirsch St. Bnei Brak 51202 Israel

Tel: 972-3- 6176666 Fax: 972-3- 6176667

The Use of this book

The material in this book is designed to assist the student during the course. It does not include all of the information that will be referred to during the course and should not be regarded as a replacement for reference manuals.

Limits of Responsibility

Sela Software Labs Ltd invests significant effort in updating this book, however, Sela Software Labs Ltd is not responsible for any errors or material which may not meet specific requirements. The user alone is responsible for decisions based on the information contained in this book.

Protected Trademarks

In this book, protected trademarks appear that are under copyright. All rights to the trademarks in this material are reserved to the authors.

SELA wishes you success in the course!

Table of Contents

Module 01 - Introduction

Module 02 - Understanding Docker

Module 03 - Installing Docker

Module 04 - Docker Architecture

Module 07 - Managing Containers

Module 08 - Volumes and Networks

Module 10 - Advanced Docker Overview

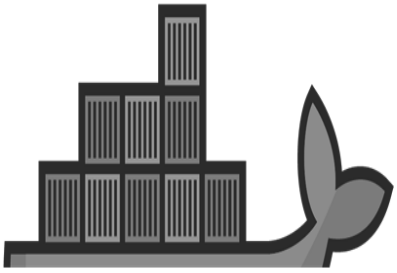
Module 01 - Introduction

Contents:

SELA

Module 01: Introduction

Docker Workshop



Agenda

- Workshop Objectives and Agenda
- Microservices 101
- Why containers ?
- Docker facts
- Resources

Workshop Objectives

- Getting started with Docker containers
- Learn Docker through hands-on labs
- Understand how Docker works behind the scenes
- And most importantly, have fun with Docker!

Workshop Agenda

- Module 01: Introduction
- Module 02: Understanding Docker
- Module 03: Installing Docker
- Module 04: Docker Architecture
- Module 05: Docker Basics
- Module 06: Building Containers
- Module 07: Managing Containers
- Module 08: Volumes & Networks
- Module 09: Working with Registries
- Module 10: Advanced Docker Overview

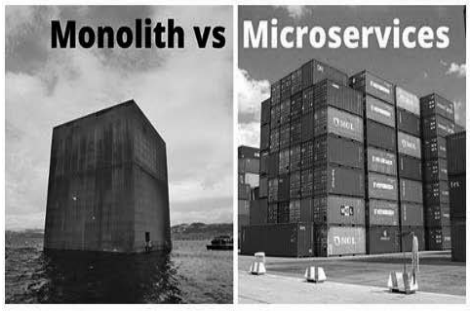
The Good and Bad of Monolithic Application

The Good - when we are just getting started:

Having one code base, one development language, one runtime, one build, one everything is easy.

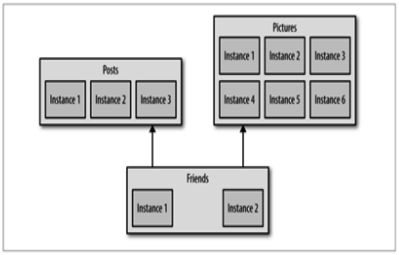
- Easy to develop*
- Easy to build*
- Easy to deploy*
- Easy to find bugs*
- Easy to maintain*
- Easy to scale*

* for small application



The Good and Bad of Monolithic Application

- Development velocity decreasing fast
  - Many developers requires synchronization
- Build takes ages to complete
  - Things are changing all the time that requires full builds, breaking each other code
- Super hard to do a Zero-Down-Time Deployments
  - All this huge thing must be replaced at once
- Maintenance takes more time than ever
  - Fixing a bug can cause instability in other components
  - Coupling is killing us
- Testing gets complicated
  - Every change can require regression testing
- Scaling is inefficient and slow
  - scaling the entire application instead of the module that needed the scale



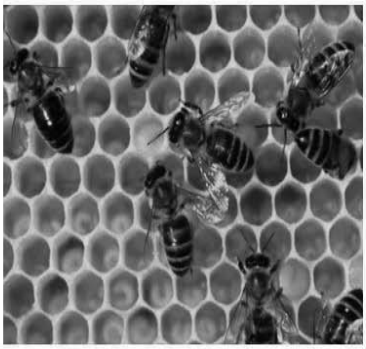
Back to basics

'Micro-services is a software development technique—[...] that structures an application as a **collection of loosely coupled services**. In a microservices architecture, **services are fine-grained** and the **protocols are lightweight**. The benefit of decomposing an application into different smaller services is that it **improves modularity** and makes the application **easier to understand, develop, test**, and **more resilient** to architecture erosion. It also parallelizes development by **enabling small autonomous teams to develop**, deploy and **scale independently**. [...] Microservices-based architectures **enable continuous delivery and deployment**.'

** Wikipedia*

Micro-services 101

- Small – do one thing and do it well
- Simple!
- Has clear domain boundaries and well defined API's
- Autonomous
- Independent development
- Independent deployment
- Build and release is automated
- Testable
- Loosely coupled

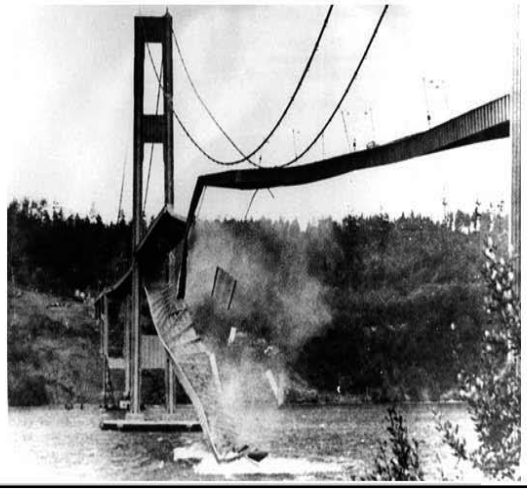


What does that mean practically?

- Small and Simple = Replaceable
  - Create services that 5 devs can re-write them as needed in one month
- Technology is just a tool
  - Use the right technology for the job!
  - Don't write Image processing in NodeJs, don't serve web pages in Java
- Use Rich and Smart Middlewares\ServiceMesh
  - Smart routing, A/B testing, circuit-breaking, tracing, monitoring, etc..
- API is King
  - Create well-defined API and context boundaries
  - Use versioning from the start
- Automate everything - Tests, Builds, Deployments, etc..

Fallacies of Distributed Computing

- The Network is reliable
- Latency is Zero
- Bandwidth is Infinite
- The network is Secure
- Topology doesn't change
- There is one Administrator
- Transport cost is zero
- The network is homogeneous



With simplicity, comes complexity ®

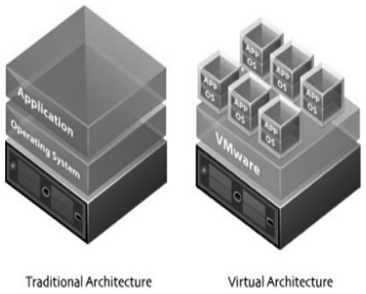
- How to deploy or update services with zero-downtime?
- How to A/B test the application?
- How to handle network failures?
- How to manage security between services?
- How to handle timeouts? Retries?
- How to rate limit? Add quotas? Back-pressure ?
- Telemetry, Logging, Monitoring?
- What about Polyglot, Legacy systems?
- Different Tech Stacks

Docker History

- Docker was created by dotCloud and launch in March 13, 2013 at PyCon Lighting Talk
- dotCloud was a PaaS platform company (now Docker Inc)
- Solomon Hykes is the father of Docker
- Hykes had a cofounder who's now at a partner company (mesosphere)
- Hykes never liked Docker's name

Why Containers ?

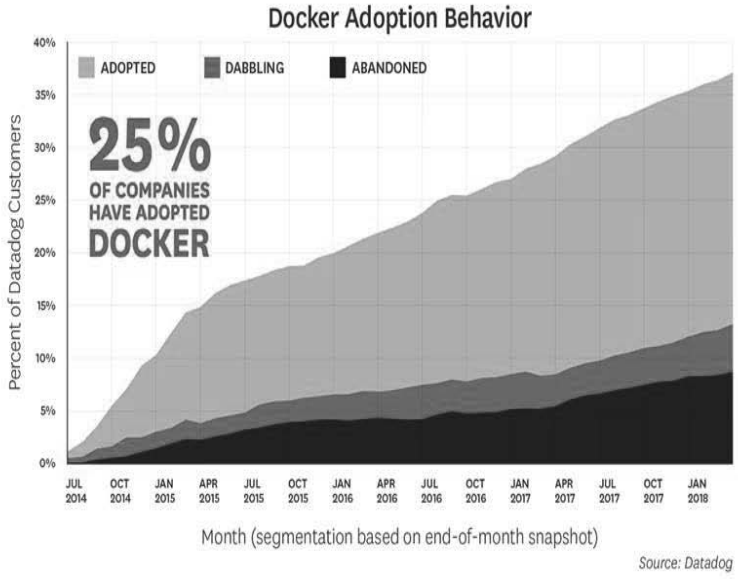
- Typically 5-10 years ago administrators would "spin-up" a virtual machine containing several applications
- Disadvantages of VM's
  - needs management layer (usually enterprise)
  - Takes time to spin up (minutes/hours)
  - High dependencies between components
  - High dependency on the Host OS
  - Updating library for app A is hard since App B relies on the same VM
  - Loading entire OS from the host



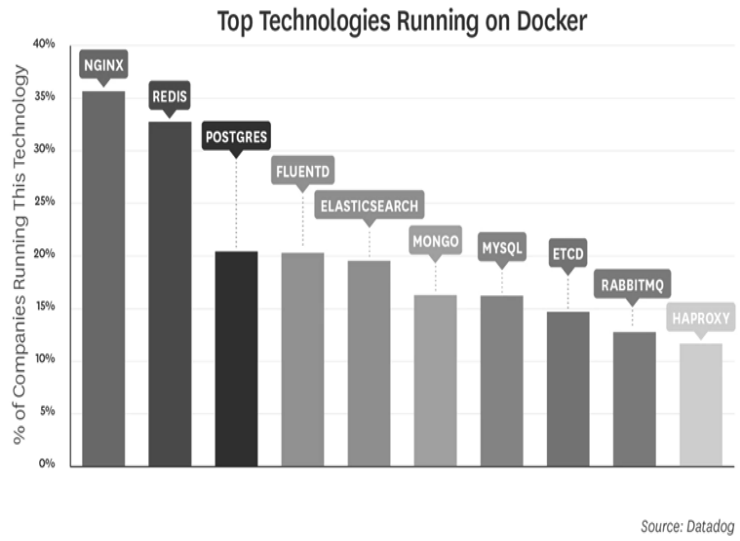
Container Definition

- An "isolated" partition inside a single Operating system
- Very low footprint in comparison to VM's
- Processes inside containers are isolated (networks, storage, secrets)
- Quick deployment
- Multiple environment support : can run it on several guest OS
- Suite for microservices architecture :
  - Isolation
  - Resilience

Docker Adaptation

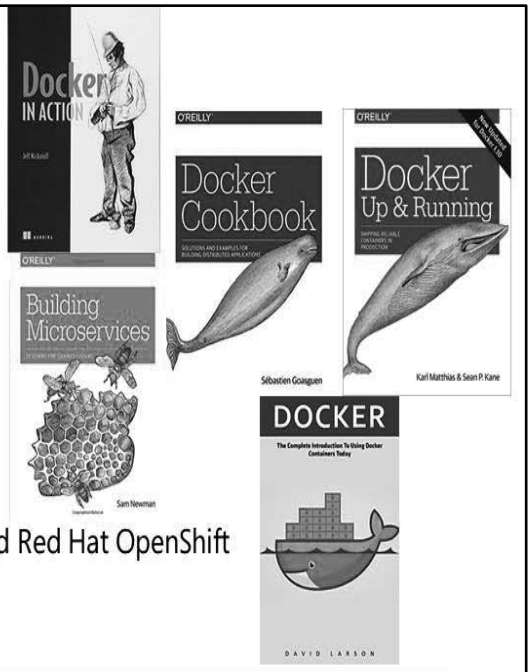


Docker facts



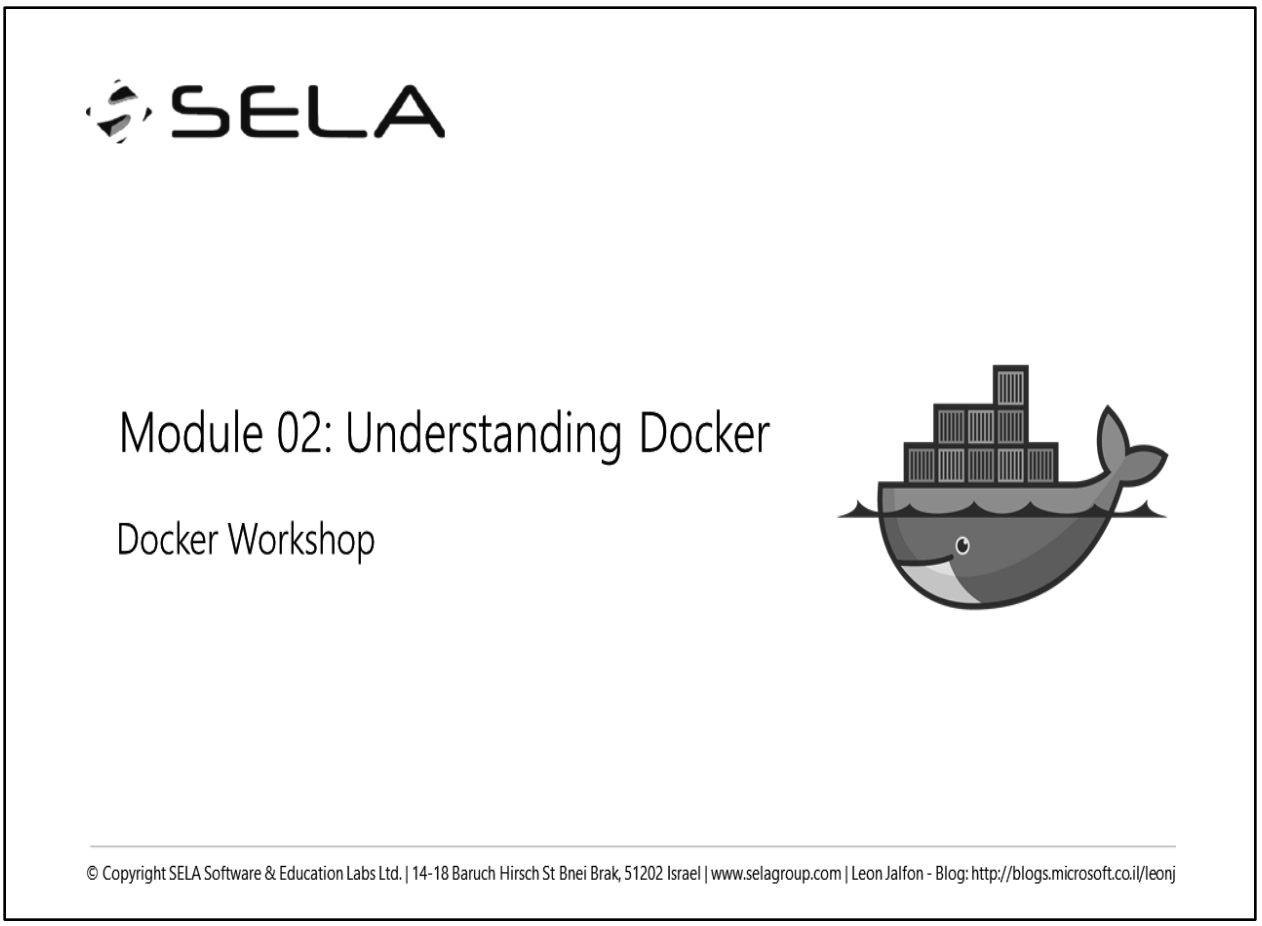
Resources for this course

- Books :
  - Docker In Action
  - Docker Up and Running
  - Docker: The Complete Introduction to Using Docker Containers Today
  - Docker Cookbook
  - Building Microservices
- Online training (free)
  - Edx -Fundamentals of Containers, Kubernetes, and Red Hat OpenShift
  - Cloud Native free training (LFS158)



Module 02 - Understanding Docker

Contents:


SELA
Module 02: Understanding Docker
Docker Workshop
© Copyright SELA Software & Education Labs Ltd. | 14-18 Baruch Hirsch St Bnei Brak, 51202 Israel | www.selagroup.com | Leon Jalfon - Blog: http://blogs.microsoft.co.il/leonj

Agenda

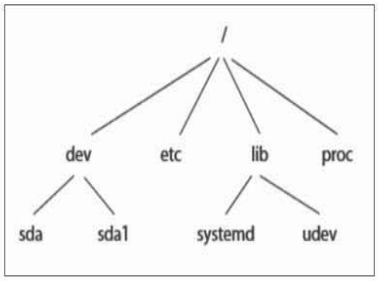
- What is a Linux Container?
- What is this all about?
- A little of history before we started
- What is Docker?
- About Docker
- Demo 01: Our first contact with docker

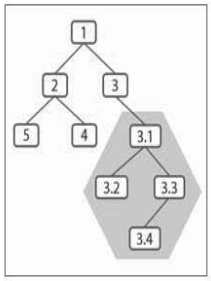
What is a Linux Container (LXC)?

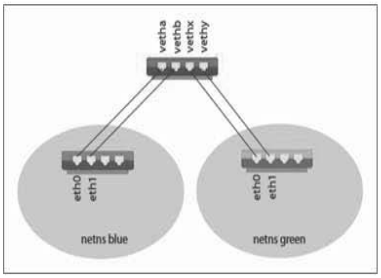
- Is an operating-system-level virtualization method.
- Run multiple isolated Linux systems (containers) on a control host using a single Linux kernel.
- Is the technology on which Docker it's based.

What is a Linux Container (LXC)?

- LXC have Independent:
  - File System
  - Process Tree
  - Networking Stacks
- Linux containers uses:
  - namespaces → Isolation
  - cgroups → Apply limits
  - capabilities → Manage privileges

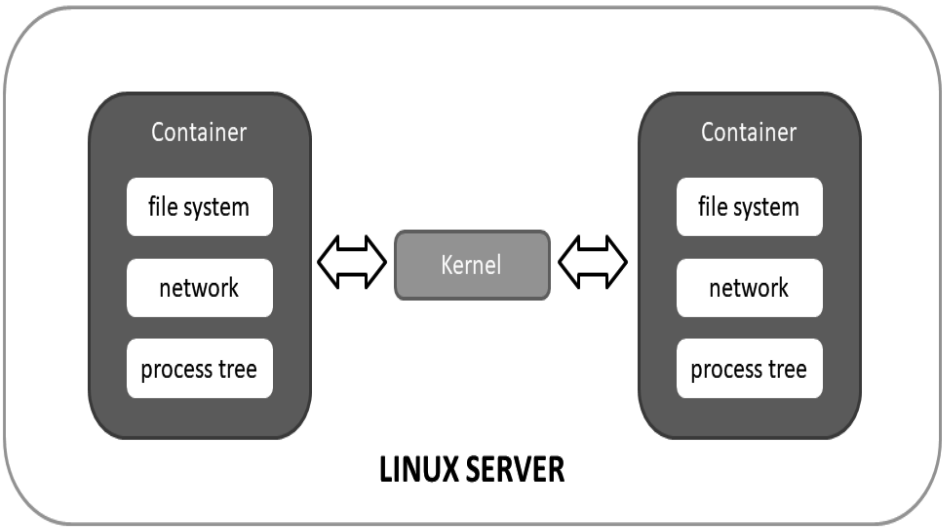






this kind of isolation is accomplished via a feature of the linux kernel called namespaces (pid, net, mnt, user)
cgroups (1:1) let us group together resources and apply limits (CPU, memory, block IO, etc)
capabilities give us fine grain control over what privileges a user or a process gets

What is a Linux Container (LXC)?



But, what is this all about?

Containers are a big IT revolution in our days,

But why?

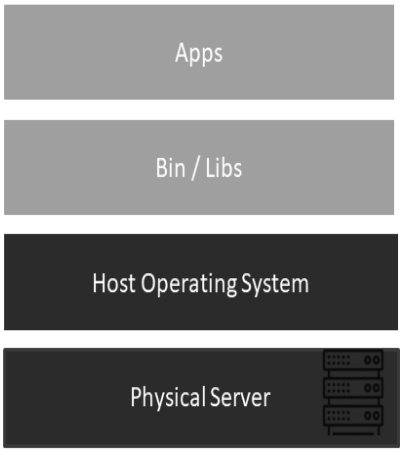
What is this all about?

Actually, it's quite simple,

Is all about the **APPLICATIONS!**

After all, the OS only exist to facilitate the application

Pre-Virtualization World



Problems :

- Huge Cost
- Slow Deployment
- Hard To migrate

A little of history before we started…

- Then, virtual machines comes to the rescue!

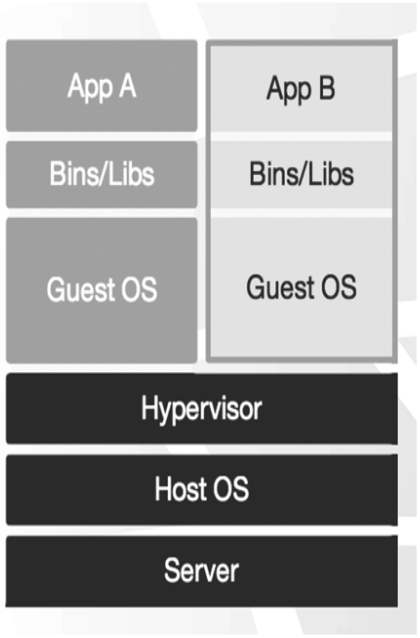
10 Applications = 1 Physical Servers

1 Physical Server = 10 Virtual Machines

10 Virtual Machines = 10 Operation Systems

(The result, a waste of resources)

Hypervisor-based Virtualization



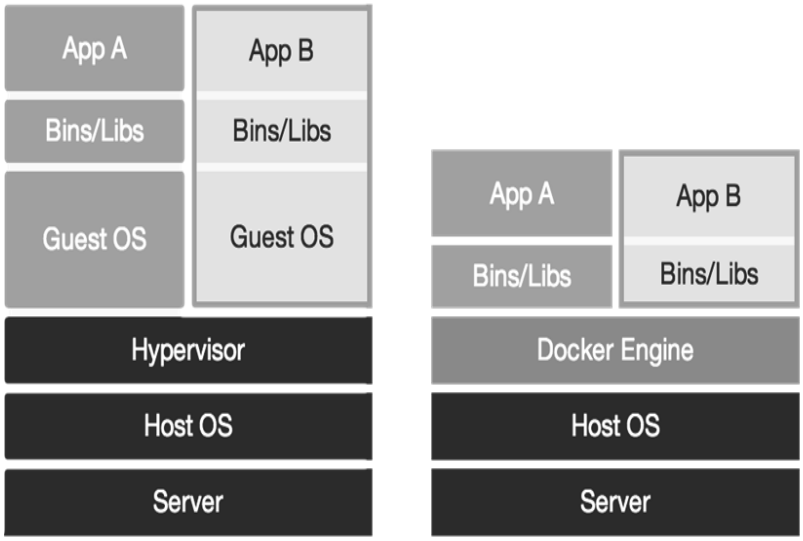
Benefits:

- Cost – Efficient
- Easy To Scale

Limitations:

- Kernel Resource duplication
- Application Portability

Docker Vs VM



Containers Advantages:

- It's then when **containers** come to solve the problem:
  - More lightweight than VM's
  - Containers consumes less CPU, less RAM and less diskspace.
  - Every container shares a single common linux kernel in the host
  - Containers are faster and more portable than VM's
  - Provides a secure isolated runtime environment for each container
  - Stackable : You can stack services vertically

(No more operating system for each application)

What is Docker?

- Docker is the platform for developers and sysadmin to develop, deploy and run applications with containers.
- It brings together the kernel namespaces, cgroups, capabilities and all of that stuff into a product
- Docker provides a very uniform and standard runtime

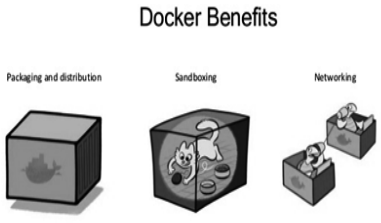
- Docker is growing more than just a container runtime, becoming more of a platform (registry, clustering, orchestration, networking, etc)

About Docker

- Docker is written in Golang and is open source under the apache 2 license
- Windows apps developed on window docker containers will only run on windows hosts running docker and the same goes for linux containers
- Docker it's a basic client server model where the docker client sends commands to the docker deamon and the deamon responds.

Benefits of using Docker – Development POV

- Packaging software in a way that leverages the skills developers already have
- Bundling application software and required OS filesystems together in a single standardized image format
- Using packaged artifacts to test and deliver the exact same artifact to all systems in all environments
- Abstracting software applications from the hardware without sacrificing resources



What docker isn't

- Enterprise virtualization platform (VMware, KVM, etc.)
- Cloud platform (OpenStack, CloudStack, etc.)
- Configuration management (Puppet, Chef, etc.)
- Deployment framework (Capistrano, Fabric, etc.)
- Workload management tool (Mesos, Kubernetes, Swarm, etc.)
- Development environment (Vagrant, etc.)

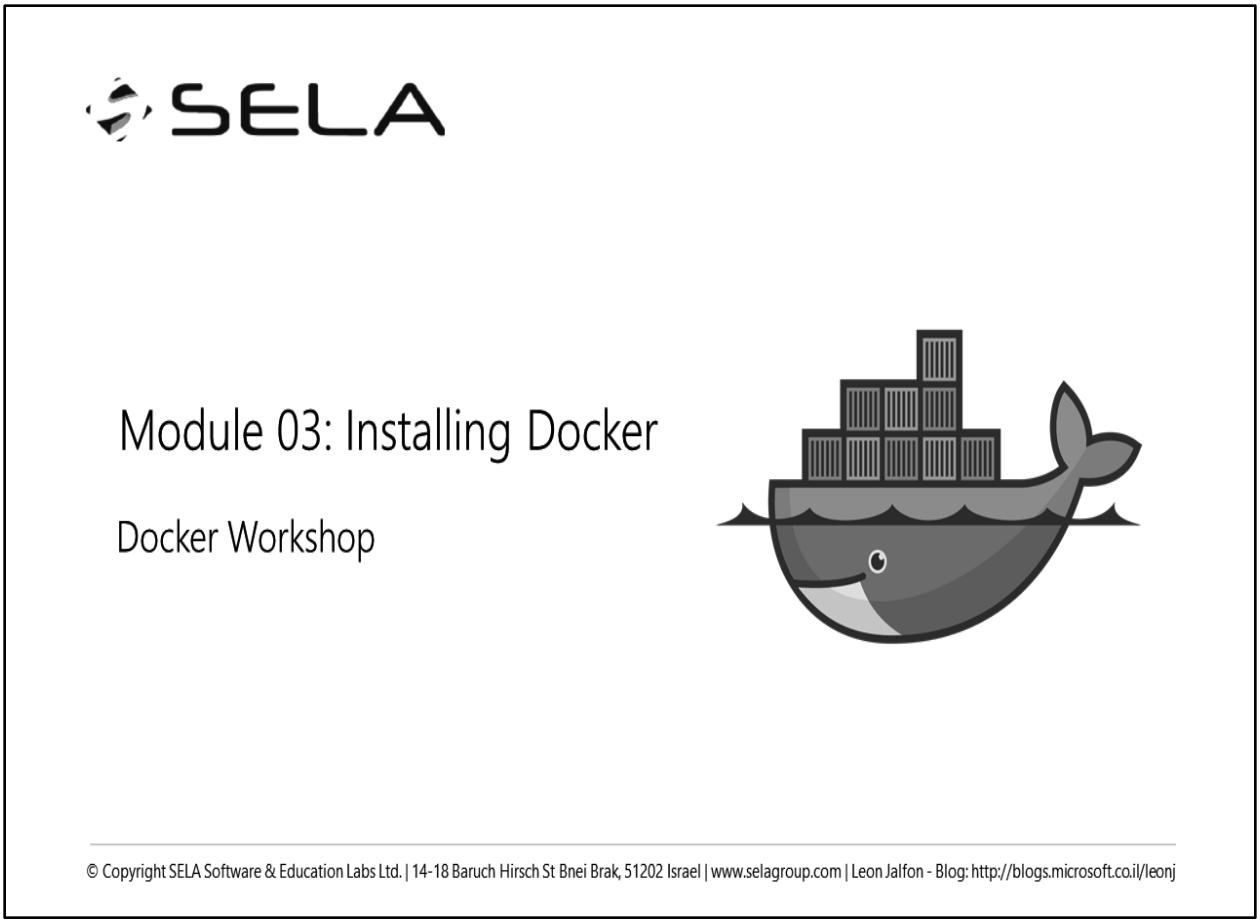
Terminology

- Docker Client – is the *docker* command used to control most of ocker workflows
- Docker images - consist of one or more filesystem layers and some important metadata that represent all the files required to run a Dockerized application.
- Docker container – a Linux container that has been instantiated from a Docker image.

Demo 01: Our first contact with Docker

Module 03 - Installing Docker

Contents:


SELA
Module 03: Installing Docker
Docker Workshop
© Copyright SELA Software & Education Labs Ltd. | 14-18 Baruch Hirsch St Bnei Brak, 51202 Israel | www.selagroup.com | Leon Jalfon - Blog: http://blogs.microsoft.co.il/leonj

Agenda

- Available Editions and Versions
- Supported Platforms
- Docker in the Cloud
- Lab 01: Installing Docker in Ubuntu 18.04

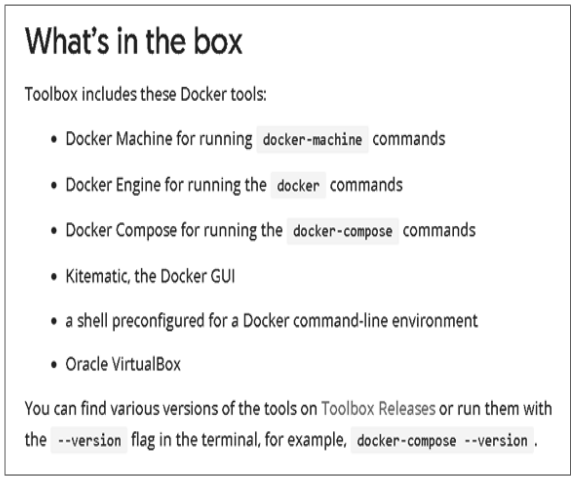
Available Editions and Versions

Capabilities	Community Edition	Enterprise Edition Basic	Enterprise Edition Standard	Enterprise Edition Advanced
Container engine and built in orchestration, networking, security	✓	✓	✓	✓
Certified infrastructure, plugins and ISV containers		✓	✓	✓
Image management			✓	✓
Container app management			✓	✓
Image security scanning				✓

- The **Stable** version gives you reliable updates every quarter
- The **Edge** version gives you new features every month

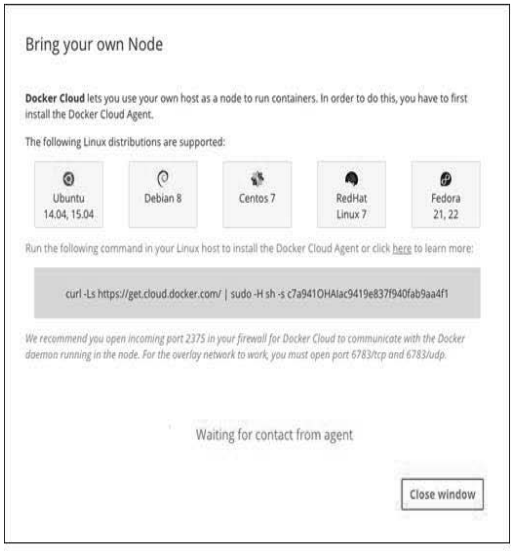
Supported Platforms

- Linux
  - Native Application
- Mac
  - Native Application
  - Docker Toolbox – Legacy
- Windows
  - Windows 10 (native)
  - Windows Server 2016 (native)
  - Other (Docker Toolbox)



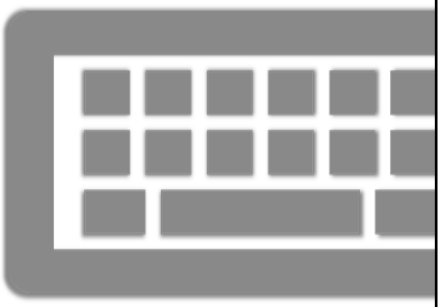
Docker in the Cloud

- Google Cloud Platform
- Amazon Web Services
- Microsoft Azure
- IBM Cloud
- Digital Ocean
- Packet
- Docker Cloud



Lab: Lab 01: Installing Docker in Ubuntu 18.04

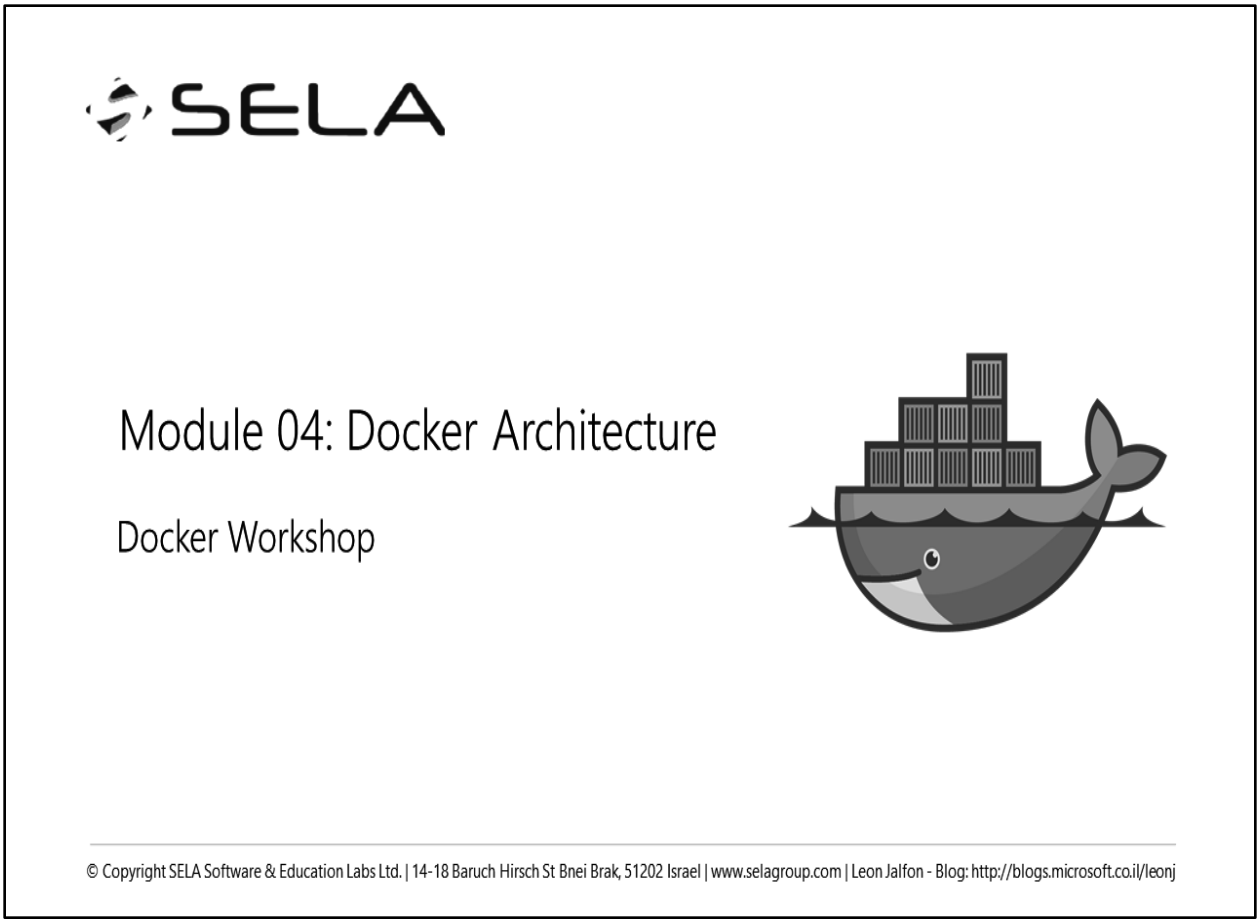
Lab



https://github.com/tshaiman/docker-workshop /lab-01

Module 04 - Docker Architecture

Contents:


SELA
Module 04: Docker Architecture
Docker Workshop
© Copyright SELA Software & Education Labs Ltd. | 14-18 Baruch Hirsch St Bnei Brak, 51202 Israel | www.selagroup.com | Leon Jalfon - Blog: http://blogs.microsoft.co.il/leonj

Agenda

- Docker Components Overview
- The Docker Engine (Deamon)
- Docker Images
- Docker Containers
- Docker Registries
- Demo 02: Understanding the Docker components

Docker Components Overview

(Shipping Yard) (Manifests) (Shipping Containers)

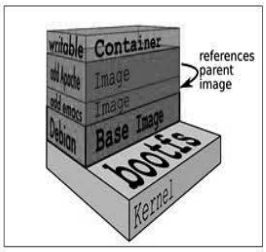
- **Shipping Yard:** The infrastructure that's needed to import and export goods
- **Manifests:** List of the container content plus instructions on how to build it
- **Containers:** A package with all the goods ready to be imported/exported

The Docker Engine (Deamon)

- It's our Docker program that we install on each Docker host to provide us with a Docker environment and access to all Docker services.
- Contains the application infrastructure and runtime dependencies (Standardized).
- Run in the same way in your laptop, datacenter or in the cloud.

Docker Images

- Are read only templates used to create containers
- Images are comprised of multiple layers
- The first image is called the "Base Image"
- Intermediate layers increases reusability, decrease disk usage, and speed up the build
- The higher layers win where there are conflicts



why not just one image with everything inside of it?
really efficient on disk and cache space, but also really easy for us to make config changes
no need to crack open a single large monolithic image and shovel new chnages in
want to make changes? just add them to a new layer

Docker Containers

- We can think of them as running instances of an image.
- Containers are created from images. Inside a container, it has all the binaries and dependencies needed to run the application.
- Under the hood containers are Linux processes running in the Docker host

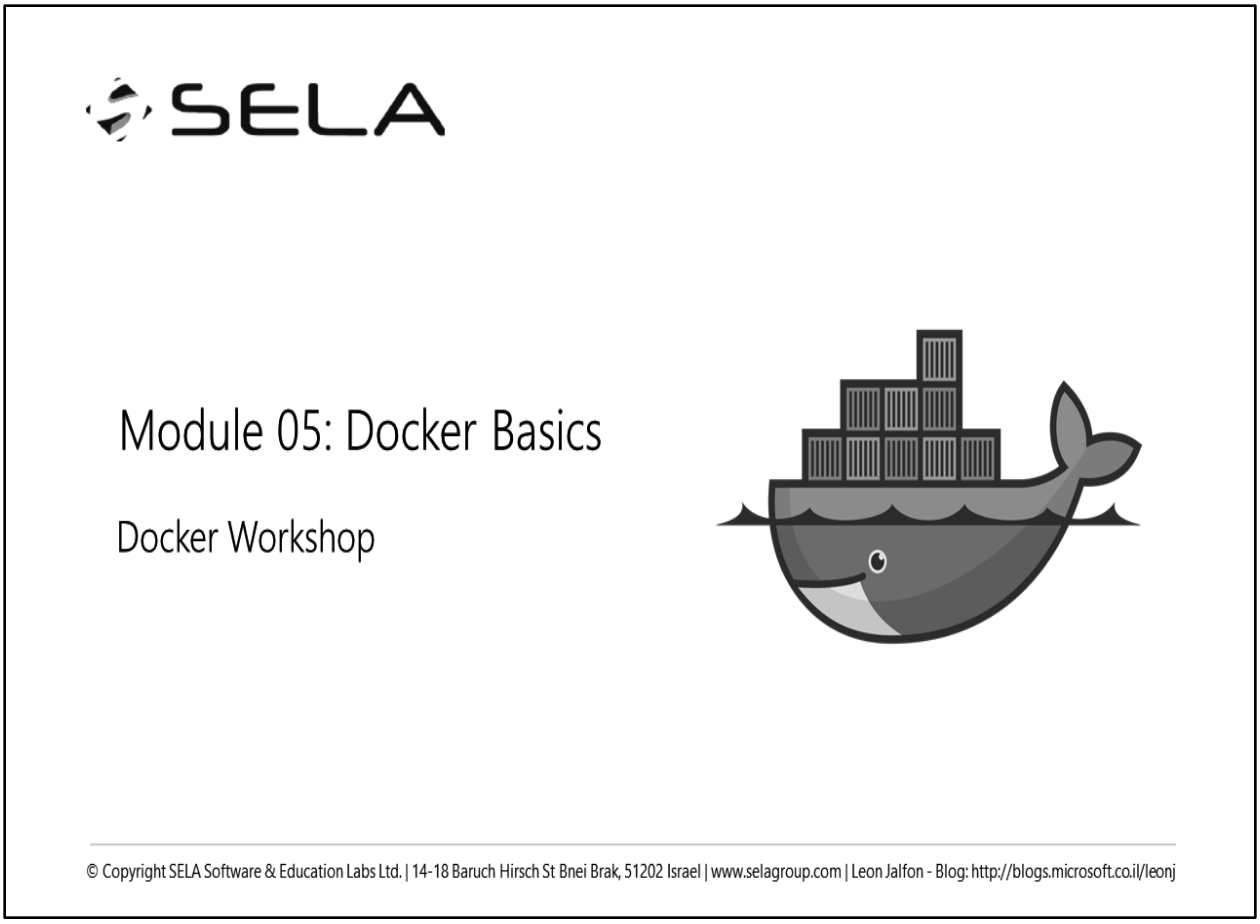
Docker Registries

- Is a stateless, highly scalable server side application that stores and lets you distribute Docker images.
- A docker registry contains multiple repositories (can be private or public)
- You can pull/push images from repositories
- Docker Hub is the default Docker Registry

Demo 02: Understanding the Docker Components

Module 05 - Docker Basics

Contents:


SELA
Module 05: Docker Basics
Docker Workshop
© Copyright SELA Software & Education Labs Ltd. | 14-18 Baruch Hirsch St Bnei Brak, 51202 Israel | www.selagroup.com | Leon Jalfon - Blog: http://blogs.microsoft.co.il/leonj

Agenda

- Docker CLI
- $ docker run
- $ docker ps
- $ docker images
- $ docker attach
- $ docker exec
- $ docker rm
- $ docker rmi
- $ docker commit
- $ docker save
- $ docker load

The Docker CLI

```
$ docker
Usage: docker [OPTIONS] COMMAND [ARG...]
       docker [ --help | -v | --version ]

A self-sufficient runtime for containers.

Options:
      --config string      Location of client config files (default "/root/.docker")
  -D, --debug              Enable debug mode
      --help               Print usage
  -H, --host value         Daemon socket(s) to connect to (default [])
  -l, --log-level string   Set the logging level ("debug"|"info"|"warn"|"error"|"fatal") (de
      --tls                Use TLS; implied by --tlsverify
      --tlscacert string   Trust certs signed only by this CA (default "/root/.docker/ca.pem
      --tlscert string     Path to TLS certificate file (default "/root/.docker/cert.pem")
      --tlskey string      Path to TLS key file (default "/root/.docker/key.pem")
      --tlsverify          Use TLS and verify the remote
  -v, --version            Print version information and quit
```

- Docker commands reference : https://docs.docker.com/engine/reference/commandline/cli/

$ docker run

```
$ docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

- Create a new container based in an specific image
- If the image is not found locally, it's pulled from the Docker Hub
- Each container have it's own Id
- The container exits once the command running inside of it exits
- Detached vs Foreground

$ docker ps

```
$ docker ps
```

- Will list the running container in your host

```
$ docker ps
CONTAINER ID   IMAGE        COMMAND                  CREATED      STATUS      PORTS                    NAMES
5f35bb815832   registry:2   "/bin/registry /etc/d"   8 months ago  Up 3 hours   0.0.0.0:5000->5000/tcp   registry
```

- "docker ps -a" command show all containers that have run in the past, but are not necessarily running now.

$ docker images

```
$ docker images
```

- Show all top level images, their repository and tags, and their size.
- Intermediate layers are not shown by default.
- To see the intermediate layer as well use the flag "-a"

$ docker rm

```
$ docker rm <container-id>
```

- Delete a container
- The container must be stopped in order to be removed
- The flag "-f" can be used to remove running containers
- You can remove all the containers at once using the command:
  - $ docker rm $(docker ps -a -q)

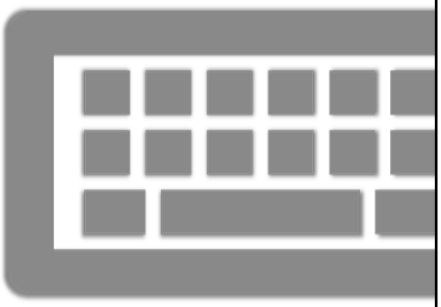
$ docker rmi

```
$ docker rmi <image-id>
```

- Delete the specified image
- You can delete multiple images in the same commands passing them as arguments, for example:
  $ docker rmi 18wj2 as83j a92k4
- You can remove all the images at once using the command:
  $ docker rmi $(docker images -a -q)

Lab: Lab 02: Basic commands

Lab



https://github.com/tshaiman/docker-workshop/lab-02

$ docker attach

```
$ docker attach <container-id>
```

- Attaches to PID1 inside the container
- To detach from the container use "Ctrl + P + Q"
- Using "Ctrl + C" will stop the process in the container (and therefore stop the container itself)

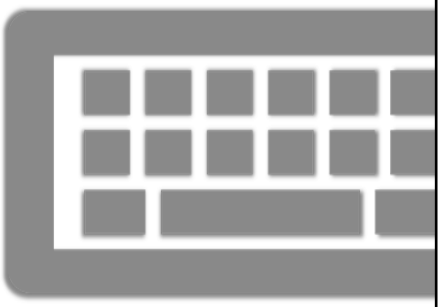
$ docker exec

```
$ docker exec <container-id> <tool>
```

- Runs a new command (process) in a running container
- Useful when the PID1 is not a shell
- You can use the flag -it to run the command interactively

Lab: Lab 03: Running commands inside the container

Lab



https://github.com/tshaiman/docker-workshop/lab-03

$ docker commit

```
$ docker commit <container-id> <new-image-name>
```

- Save container status creating a new image
- By default, the container being committed and its processes will be paused while the image is committed
- Use the flag "-p=false" to avoid this behavior

$ docker save

```
$ docker save -o <path/to/file.tar> <image-id>
```

- Save a container image in a file
- Useful to share containers without a container registry

$ docker load

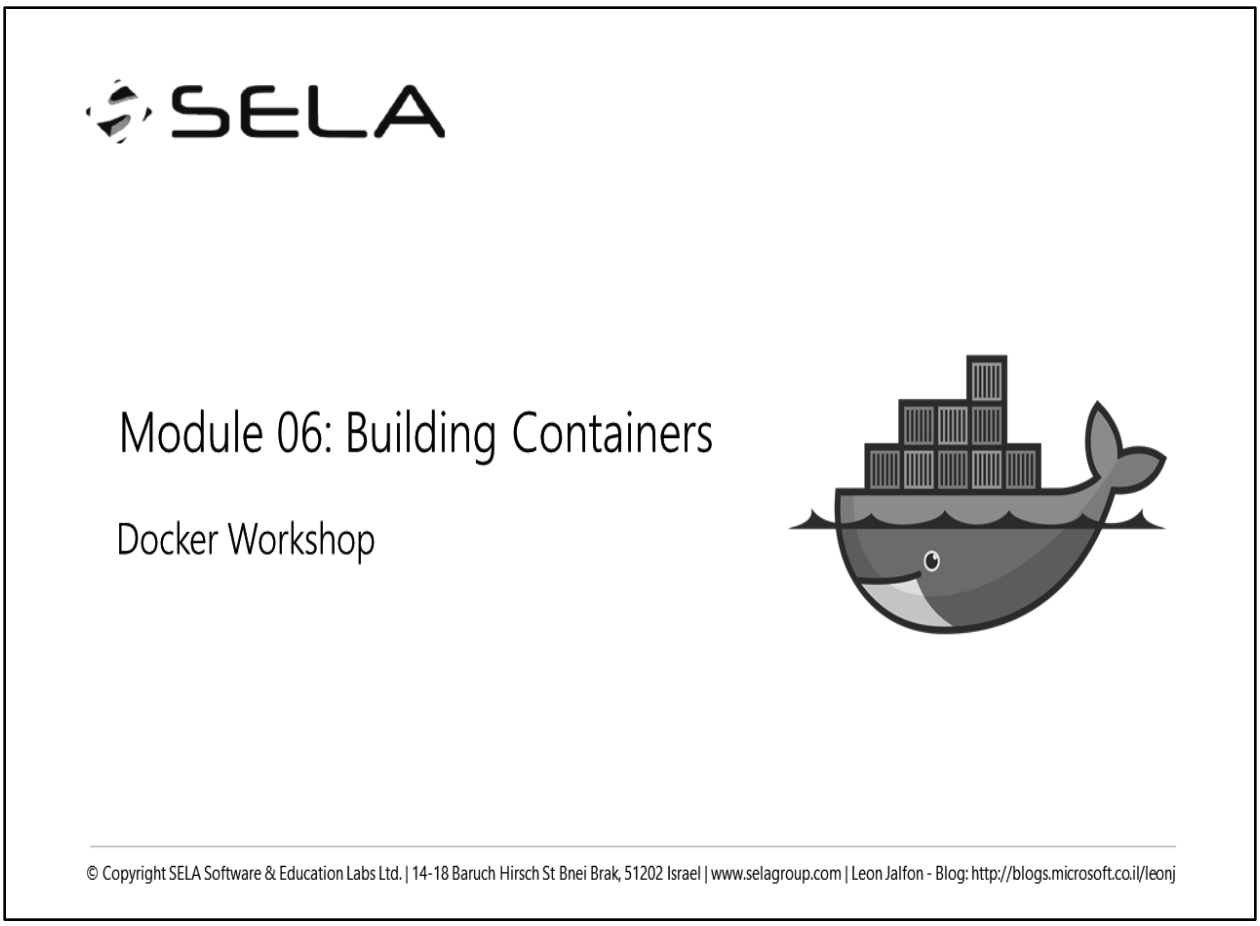
```
$ docker load -i <path/to/file.tar>
```

- Load a container image from a file
- Useful to share containers without a container registry

Lab: Lab 04: Updating and Sharing Containers

Module 06 - Building Containers

Contents:


SELA
Module 06: Building Containers
Docker Workshop
© Copyright SELA Software & Education Labs Ltd. | 14-18 Baruch Hirsch St Bnei Brak, 51202 Israel | www.selagroup.com | Leon Jalfon - Blog: http://blogs.microsoft.co.il/leonj

Agenda

- The Dockerfile
- $ docker build
- Dockerfile Instructions
- Lab 05: Building your first image
- Lab 06: Build more complex images
- Best Practices & Common Mistakes

The Dockerfile

- Dockerfile (with capital D)
- It's comprised of instructions that define how to build an image
- These instructions get read one at time, from top to bottom

```
FROM ubuntu:14.04
RUN \
  apt-get update && \
  apt-get -y install apache2

VOLUME /myvol
ADD index.html /var/www/html/index.html

EXPOSE 80

CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

- When we build images from Dockerfiles, any other files and directories in the same directory as the Dockerfile were going to get included in the build (build context).

$ docker build

```
$ docker build -t name:tag  .
```

```
$ docker build github.com/creack/docker-firefox
```

- Create a new Docker image following the Dockerfile instructions
- You can specify a Dockerfile in the filesystem or build an image from a Dockerfile stored in GitHub
- Tags are used to manage image versions

Dockerfile – Instructions

FROM

- Sets the Base Image for subsequent instructions
- The image can be any valid image (usually a container start by pulling an image from the Docker Hub)

```
FROM ubuntu:15.04
```

Dockerfile – Instructions

LABEL

- Add metadata to the docker image
- Used to indicate things like the image maintainer, version, description, etc

```
FROM ubuntu:15.04
LABEL maintainer="leonj@sela.co.il"
LABEL version="1.0"
LABEL description="my image description"
```

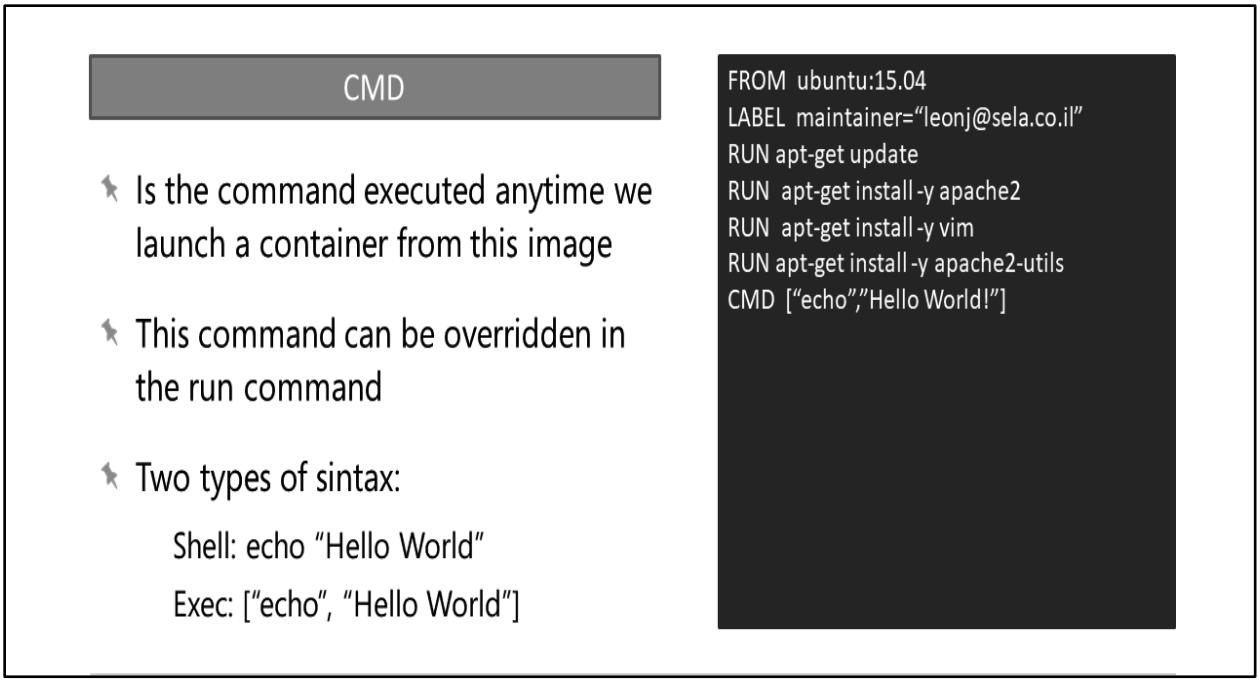
Dockerfile – Instructions

RUN

- Used to run commands against our images that we're building
- Every run instruction adds a layer to our image
- Run commands are the image "build steps"

```
FROM ubuntu:15.04
LABEL maintainer="leonj@sela.co.il"
RUN apt-get update
RUN apt-get install -y apache2
RUN apt-get install -y vim
RUN apt-get install -y apache2-utils
```

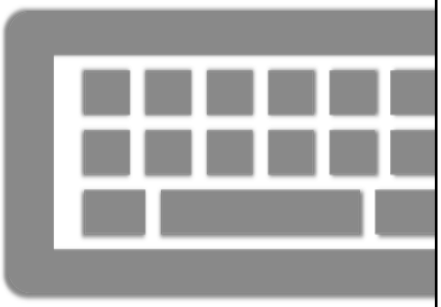
Dockerfile – Instructions



Shell form: [command "arg"]
Exec form: ["command","arg"] (recommended)
-> Allow commands to be executed inside of containers that don't have a shell
-> Avoid any potential string munging by the shell
-> No goodness of the shell (no variable expansion, no special characters)

Lab: Lab 05: Building your first image

Lab



https://github.com/tshaiman/docker-workshop/lab-05

Build Cache

- When we build a new image the docker deamon iterates through our Dockerfile executing each instruction.
- As each instruction gets executed, the deamon checks to see whether or not it's got an image for that instruction already in its build cache
- The build cache store each instruction + linked image
- (Change the docker file invalidates the build cache)

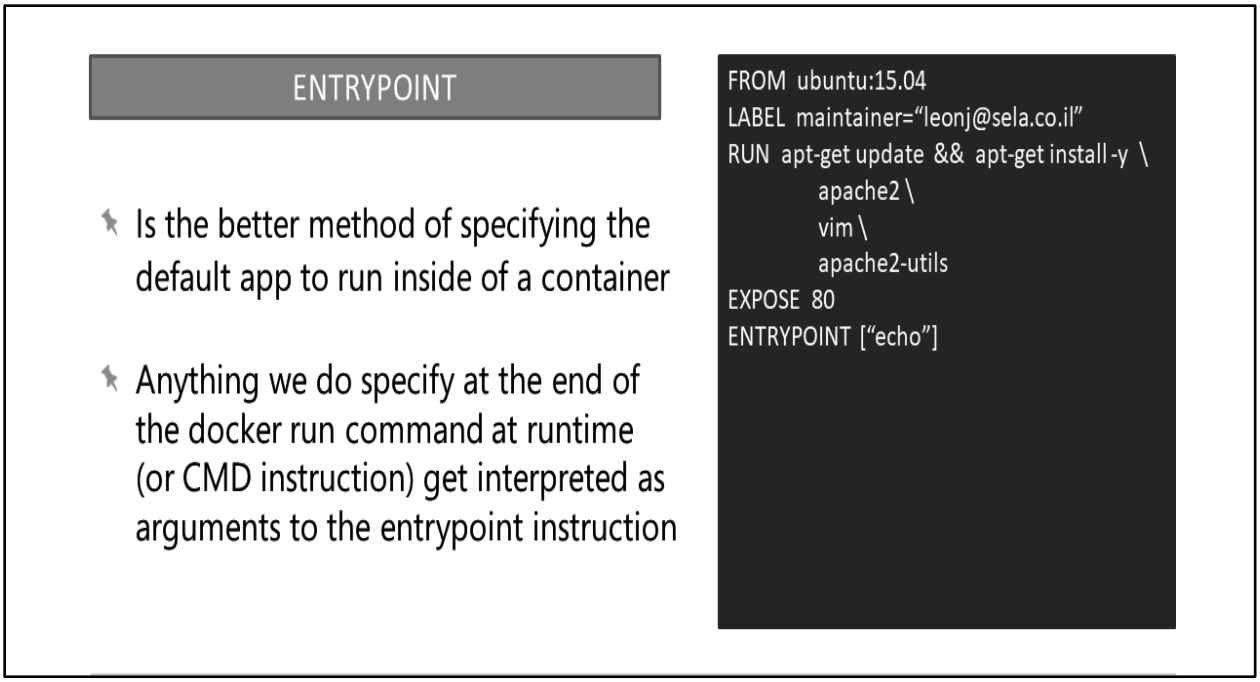
Dockerfile – Instructions

EXPOSE

- The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime.
- You can expose one port number and publish it externally under another number.

```
FROM ubuntu:15.04
LABEL maintainer="leonj@sela.co.il"
RUN apt-get update && apt-get install -y \
        apache2 \
        vim \
        apache2-utils
EXPOSE 80
CMD ["apache2ctl","-D","FOREGROUND"]
```

Dockerfile – Instructions



Shell form: [command "arg"]
Exec form: ["command","arg"] (recommended)
-> Allow commands to be executed inside of containers that don't have a shell
-> Avoid any potential string munging by the shell
-> No goodness of the shell (no variable expansion, no special characters)

Dockerfile – Instructions

ENV

- Used to assign environment variables
- Environment variables can be overridden in the docker run command using **-e "var=value"**

```
FROM ubuntu:15.04
LABEL maintainer="leonj@sela.co.il"
RUN apt-get update && apt-get install -y \
        apache2 \
        vim \
        apache2-utils
ENV var1=val1
EXPOSE 80
ENTRYPOINT echo $var1
```

Dockerfile – Instructions

ARG

- The ARG instruction defines variables used at build-time
- Arguments can be passed in the docker build command using the flag **--build-arg <varname>=<value>**

```
FROM ubuntu:15.04
LABEL maintainer="leonj@sela.co.il"
RUN apt-get update && apt-get install -y \
        apache2 \
        vim \
        apache2-utils
ENV var1=val1
ARG var2=val2
RUN echo $var2
EXPOSE 80
ENTRYPOINT echo $var1
```

Dockerfile – Instructions

COPY

- Copies new files or directories from <src> and adds them to the filesystem of the container at the path <dest>
- Each <src> may contain wildcards

```
FROM ubuntu:15.04
LABEL maintainer="leonj@sela.co.il"
RUN apt-get update && apt-get install -y \
        apache2 \
        vim \
        apache2-utils
COPY hom* /mydir
ENV var1=val1
ARG var2=val2
RUN echo $var2
EXPOSE 80
ENTRYPOINT ["echo"]
```

Dockerfile – Instructions

ADD

- Similar than Copy but:
  - ADD allows <src> to be an URL
  - If the <src> parameter of ADD is an archive in a recognized compression format or a URL, it will be unpacked

```
FROM ubuntu:15.04
LABEL maintainer="leonj@sela.co.il"
RUN apt-get update && apt-get install -y \
        apache2 \
        vim \
        apache2-utils
ADD test.tar.gz /mydir
ENV var1=val1
ARG var2=val2
RUN echo $var2
EXPOSE 80
ENTRYPOINT ["echo"]
```

Shell form: [command "arg"]
Exec form: ["command","arg"] (recommended)
-> Allow commands to be executed inside of containers that don't have a shell
-> Avoid any potential string munging by the shell
-> No goodness of the shell (no variable expansion, no special characters)

Dockerfile – Instructions

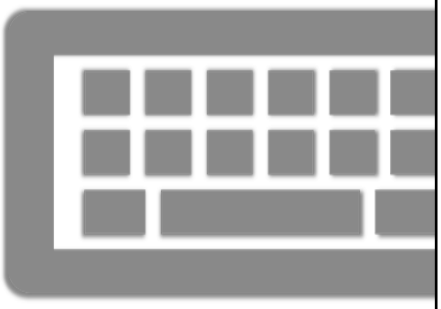
WORKDIR

- Used to set the working directory inside the container

```
FROM ubuntu:15.04
LABEL maintainer="leonj@sela.co.il"
RUN apt-get update && apt-get install -y \
        apache2 \
        vim \
        apache2-utils
ADD test.tar.gz /mydir
ENV var1=val1 var2=val2
ARG var2=val2
RUN echo $var2
EXPOSE 80
WORKDIR /mydir
ENTRYPOINT ["echo"]
```

Lab: Lab 06: Building more complex images

Lab



https://github.com/tshaiman/docker-workshop/lab-06

Dockerfile Common Mistakes

- Using "latest" tag in base images
- Using external services during the build
- Adding EXPOSE and ENV at the top of your Dockerfile
- Add the app directory at the beginning of the Dockerfile
- Wrong multiple FROM statements
- Multiple services running in the same container

Dockerfile Best Practices

- Use a .dockerignore file
- Containers should be immutable & ephemeral (no data inside)
- Minimize the number of layers / Consolidate instructions
- Avoid installing unnecessary packages
- Sort multi-line arguments
- Use Build cache
- Understand CMD and ENTRYPOINT

The alpine base image

- Alpine is a vert tiny linux image which "weights" only 3.6 MB !

DISTRIBUTION	VERSION	SIZE
Debian	Jessie	123MB
CentOS	7	193MB
Fedora	25	231MB
Ubuntu	16.04	118MB
Alpine	3.6	3.98MB

- Suppose your organization has millions pulls from Docker-hub each year, this pull size difference can save to your company approx. 400K$ a year!

The alpine base image

Debian

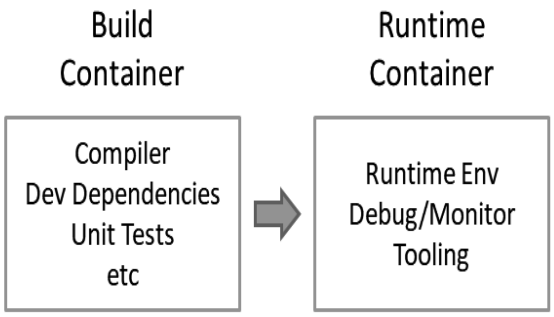
```
time docker run --rm debian sh -c "apt-get update && apt-get install curl"

real    0m27.928s
user    0m0.019s
sys     0m0.077s
```

Alpine

```
time docker run --rm alpine sh -c "apk update && apk add curl"

real    0m5.698s
user    0m0.008s
sys     0m0.037s
```

Docker Builder Pattern

- Pattern used to create small images



- Benefits:
  - Size
  - Performance
  - Security

Docker Builder Pattern

700 MB
392 Vulnerabilities

```
FROM go:onbuild
WORKDIR /app
ADD . /app
RUN cd /app && go build -o goapp
EXPOSE 8080
ENTRYPOINT ./goapp
```

VS

12 MB
3 Vulnerabilities

```
FROM golang:alpine AS build-env
WORKDIR /app
ADD . /app
RUN cd /app && go build -o goapp

FROM alpine
RUN apk update && apk add ca-
certificates && rm -rf /var/cache/apk/*
WORKDIR /app
COPY --from=build-env /app/goapp /app

EXPOSE 8080
ENTRYPOINT ./goapp
```

What is a better approach?

(A)

```
FROM python:3.6
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
ENTRYPOINT ["python"]
CMD ["ap.py"]
```

(B)

```
FROM python:3.6
WORKDIR /app
COPY requirements.txt /app/requirements.txt
RUN pip install -r requirements.txt
COPY . /app
ENTRYPOINT ["python"]
CMD ["ap.py"]
```

- **B)** The COPY . /app command will invalidate the cache as soon as any file in the current directory is updated.

Only requirements.txt should invalidate the build cache in order to install the updated packages

What is a better approach?

(A)

```
FROM centos:7

RUN yum update -y
RUN yum install -y sudo
RUN yum install -y git
RUN yum clean all
```

470 MB

(B)

```
FROM centos:7

RUN yum update -y \
 && yum install -y \
 sudo \
 git \
 && yum clean all
```

265 MB

- **B)** In "A" many layers were added for nothing. Also the "yum clean all" command is meant to reduce the size of the image but it actually does the opposite by adding a new layer

Only requirements.txt should invalidate the build cache in order to install the updated packages

Building Node.js and Java applications

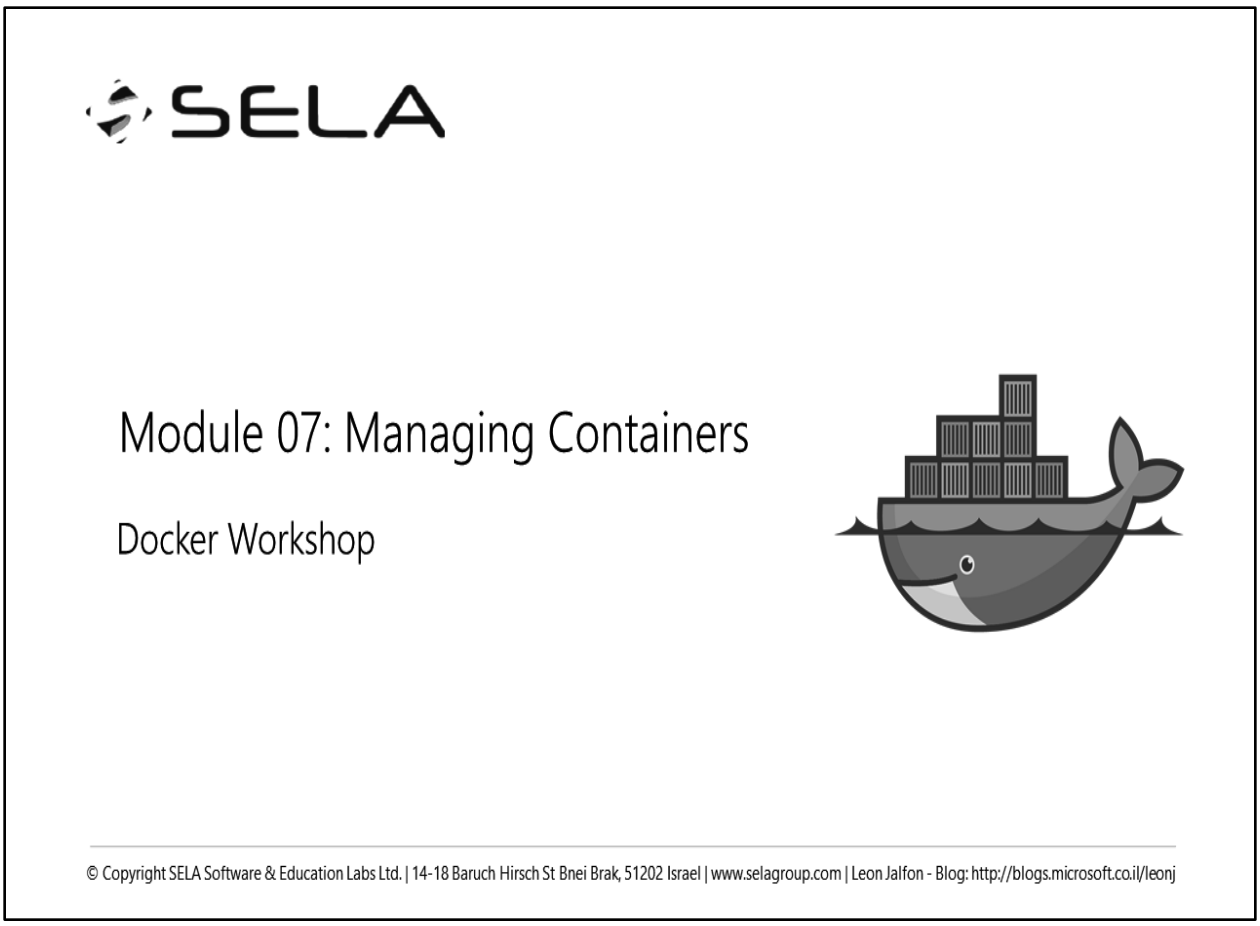
Demo

https://github.com/tshaiman/docker-workshop/Demos/hello-java
https://github.com/tshaiman/docker-workshop/Demos/hello-node

Questions

Module 07 - Managing Containers

Contents:


SELA
Module 07: Managing Containers
Docker Workshop
© Copyright SELA Software & Education Labs Ltd. | 14-18 Baruch Hirsch St Bnei Brak, 51202 Israel | www.selagroup.com | Leon Jalfon - Blog: http://blogs.microsoft.co.il/leonj

Agenda

- $ docker stop
- $ docker kill
- $ docker start
- $ docker restart
- $ docker info
- $ docker top
- $ docker history
- $docker inspect
- $ docker logs

$ docker stop

```
$ docker stop <container-id>
```

- Terminate the container PID1 process and make sure that all other processes on the system are cleaned up as gracefully as possible

$ docker kill

```
$ docker kill <container-id>
```

- Kill the container PID1 process

$ docker start

```
$ docker start <container-id>
```

- Start a container

$ docker restart

```
$ docker restart <container-id>
```

- Restart a container

$ docker info

```
$ docker info
```

- Display system-wide information
- The number of images shown is the number of unique images. The same image tagged under different names is counted only once

$ docker top

```
$ docker top <container-id>
```

- Display the running processes of a container
- Note it shows the view of processes and PIDs from outside the container (from our docker host, within the wider namespace of the docker host process tree)
- If we want to see the PIDs inside the container we can jump into the container and check (docker attach + ps -ef)

$ docker history

```
$ docker history <id>
```

- Show the history of an image
- When using the --format option, the history command will either output the data exactly as the template declares or, when using the table directive, will include column headers as well.

$ docker inspect

```
$ docker inspect <container-id>
```

- Return low-level information on Docker objects
- By default, docker inspect will render results in a JSON array.

$ docker logs

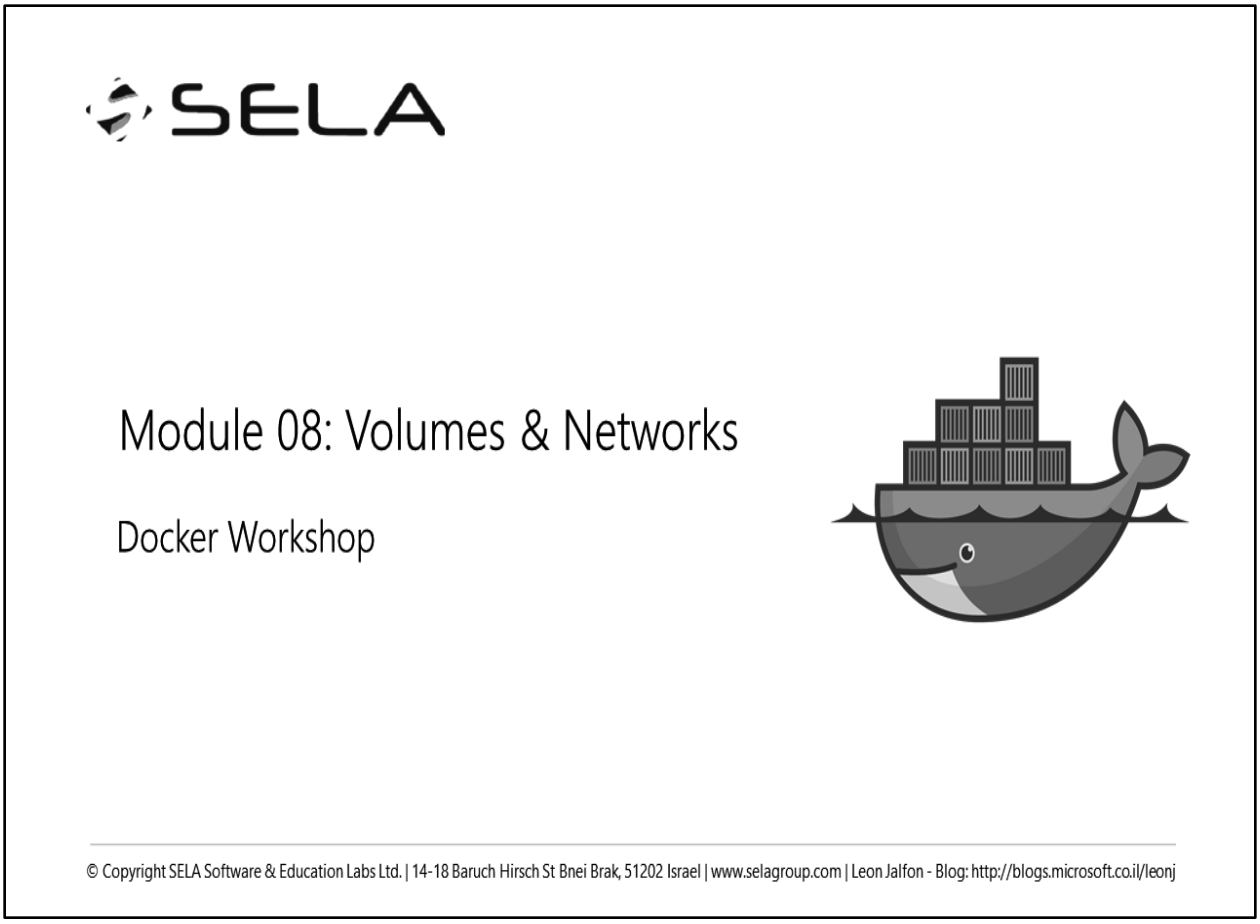
```
$ docker logs <container-id>
```

- Fetch the logs of a container (output)
- The docker logs --follow command will continue streaming the new output from the container's STDOUT and STDERR
- The docker logs --details command will add on extra attributes, such as environment variables and labels, provided to --log-opt when creating the container.

Lab: Lab 07: Managing Containers

Module 08 - Volumes and Networks

Contents:

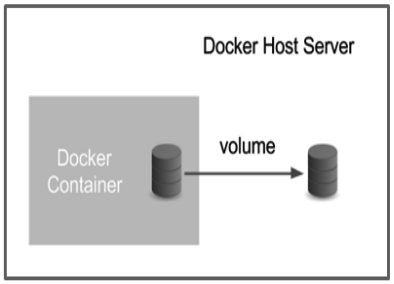

SELA
Module 08: Volumes & Networks
Docker Workshop
© Copyright SELA Software & Education Labs Ltd. | 14-18 Baruch Hirsch St Bnei Brak, 51202 Israel | www.selagroup.com | Leon Jalfon - Blog: http://blogs.microsoft.co.il/leonj

Agenda

- Docker Volumes
- Lab 08: Using Volumes
- Docker Networking
- Lab 09: Docker Networks

Docker Volumes

- When a container dies all the data it has created (logs, database records, etc) dies with it (and remember containers are ephemeral).
- Volumes are external storage areas used to store data produced by a Docker container.
- Volumes can be located on the Docker host or even on remote machines



Docker Volumes

- By default volumes are not deleted when the container is stopped.
- Data volumes can be shared across containers.
- Volumes could be mounted in read-only mode.
- **--volume:** Create a file or directory if it doesn't exist on the Docker host
- **--mount:** Does not automatically create it for you, but generates an error

Docker Volumes - Syntax

```
$ docker run [OPTIONS] -v "volume_name:/container/path" [IMAGE]
```

```
$ docker run [OPT] --mount "type=bind,source=host/path,target=contain/path" [IMAGE]
```

- Optionally you can create a volume with a default name:

```
$ docker run [OPTIONS] -v "container/path" [IMAGE]
```

Docker Volumes – Dockerfile

VOLUME

- Create a new volume with any data that exists at the specified location within the base image.
- Anything after the VOLUME instruction will not be able to make changes to that volume.

```
FROM microsoft/iis
RUN powershell -NoProfile -Command
Remove-Item -Recurse
C:\inetpub\wwwroot\*
WORKDIR /inetpub/wwwroot
COPY . .
VOLUME c:/inetpub/wwwroot
EXPOSE 80
```

Docker Volumes – Managing Volumes

- Create a volume:

```
$ docker volume create volume-name
```

```
$ docker volume create demo-volume
demo-volume
```

- List volumes:

```
$ docker volume ls
```

```
$docker volume ls
DRIVER          VOLUME NAME
local           demo-volume
local           ed702f0a2c8b6ceb56...
local           my_volume
```

Docker Volumes – Managing Volumes

- Inspect a volume:

```
$ docker volume inspect volume-name
```

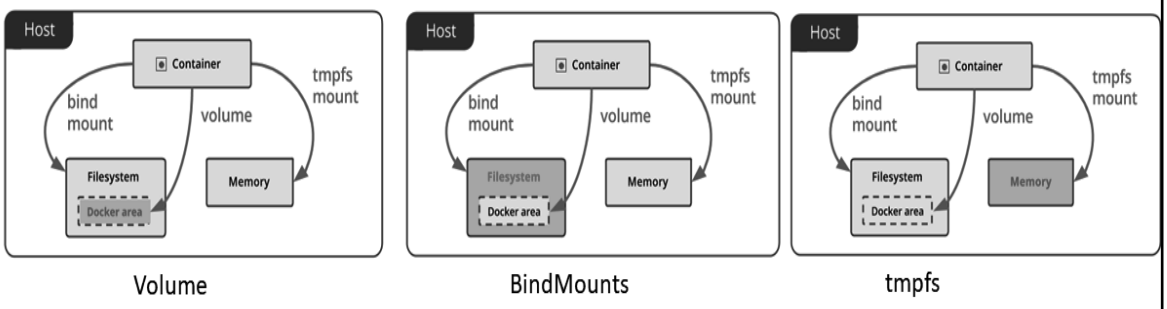
```
$ docker volume inspect demo-volume
[
  {
    "CreatedAt": "2018-06-07T23:39:20+03:00",
    "Driver": "local",
    "Labels": {},
    "Mountpoint":
"/var/lib/docker/volumes/demo-volume/_data",
    "Name": "demo-volume",
    "Options": {},
    "Scope": "local"
  }
]
```

- Remove a volume:

```
$ docker volume rm volume-name
```

```
$ docker volume rm demo-volume
demo-volume
```

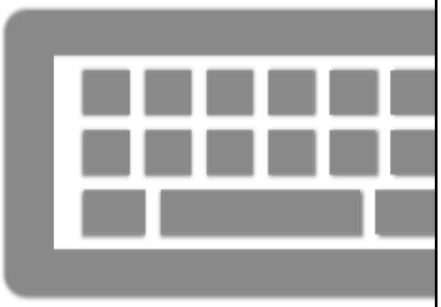
Volume vs. BindMounts vs. tmpfs



- Volumes are the preferred way to persisting data since they do not depend on the host filesystem structure
- BindMounts :mounts a file/directory on the host system into the container.
- tmpfs only works on linux and cannot be shared between containers

Lab: Lab 08: Using Volumes

Lab



https://github.com/tshaiman/docker-workshop/lab-08

Docker Networking

- Docker includes support for networking containers through the use of network drivers.
- By default, the container is assigned an IP address for every Docker network it connects to (the Docker daemon acts as a DHCP server).
- By default, a container inherits the DNS settings of the Docker daemon (can be overridden on a per-container basis).

Network Drivers

- **bridge:** Allows containers connected to the same bridge network to communicate, while providing isolation from containers which are not connected to that bridge network.
- **host:** For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly.
- **overlay:** Creates a distributed network among multiple Docker hosts. (available only using swarm mode)

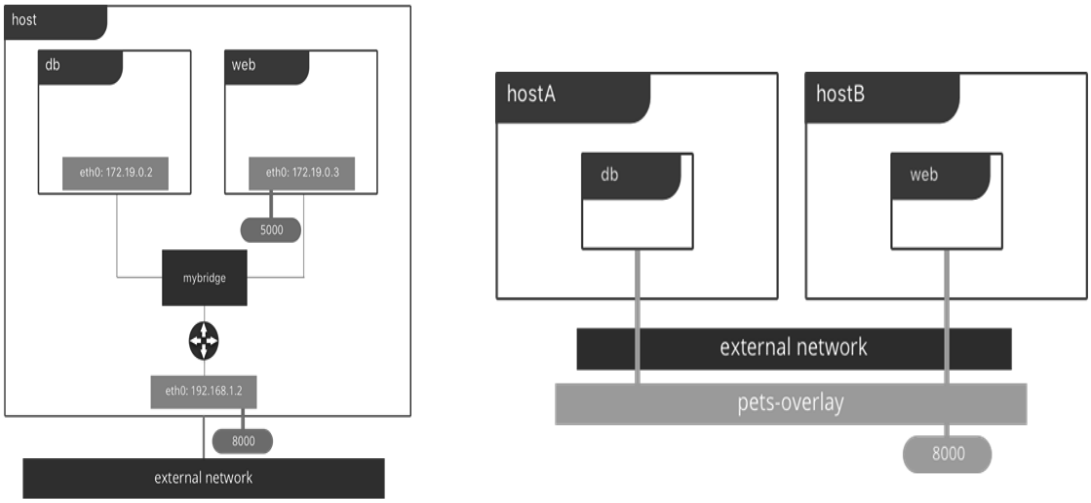
- The host networking driver only works on Linux hosts, and is not supported on Docker for Mac, Docker for Windows, or Docker EE for Windows Server.

Network Drivers

- **macvlan:** Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network.
- **none:** disable all networking for the container.
- **Network Plugins:** You can install and use third-party network plugins with Docker. (Develop or download from the Docker Store)

- The host networking driver only works on Linux hosts, and is not supported on Docker for Mac, Docker for Windows, or Docker EE for Windows Server.
  - To use the overlay driver you must be in swarm mode.
  - You also can install/develop network drivers

Bridge vs Overlay Networks



Default Networks

- Every installation of the Docker Engine automatically includes three default networks:

```
$ docker network ls

NETWORK ID          NAME                DRIVER
18a2866682b8        none                null
c288470c46f6        host                host
7b369448dccb        bridge              bridge
```

- Unless you tell it otherwise, Docker always launches your containers in the "bridge" network

Docker Networks – Managing Networks

- Create a network:

```
$ docker network create -d bridge network-name
```

- Delete a network:

```
$ docker network rm network-name
```

Docker Networks – Managing Networks

- Run a container adding it to an specific network:

```
$ docker run [OPTIONS] --network=network-name [IMAGE]
```

- Add running container to a network:

```
$ docker network connect network-name [CONTAINER]
```

Docker Networks – Managing Networks

- Disconnect container from a network:

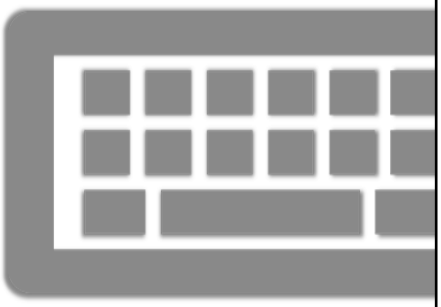
```
$ docker network disconnect network-name [CONTAINER]
```

- Inspect networks:

```
$ docker network inspect network-name
```

Lab: Lab 09: Docker Networks

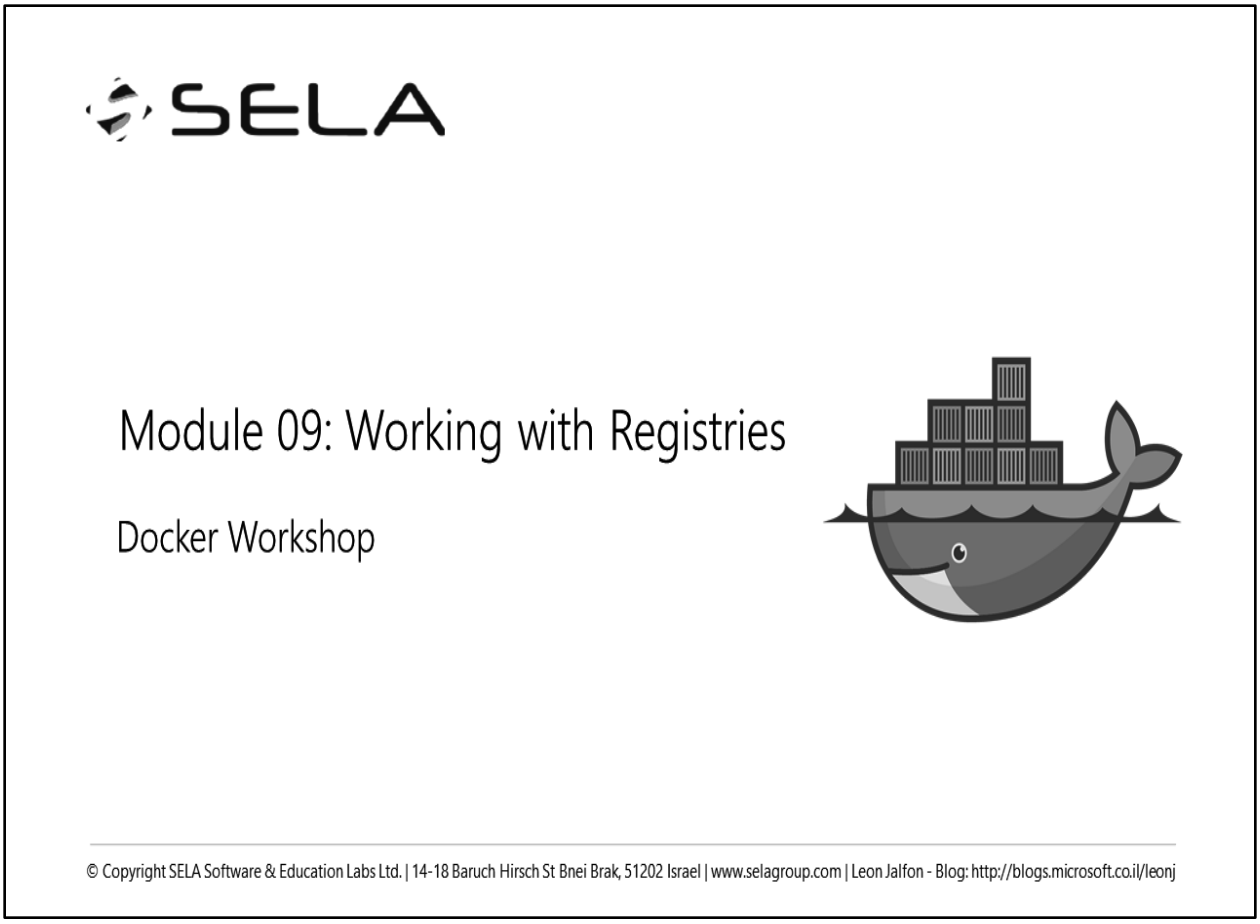
Lab



https://github.com/tshaiman/docker-workshop/lab-09

Module 09 - Working with Registries

Contents:


SELA
Module 09: Working with Registries
Docker Workshop
© Copyright SELA Software & Education Labs Ltd. | 14-18 Baruch Hirsch St Bnei Brak, 51202 Israel | www.selagroup.com | Leon Jalfon - Blog: http://blogs.microsoft.co.il/leonj

Agenda

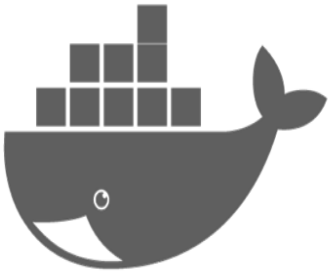
- Docker Registries
- Lab 10: Pushing images to Docker Hub (Registry)

Docker Registries

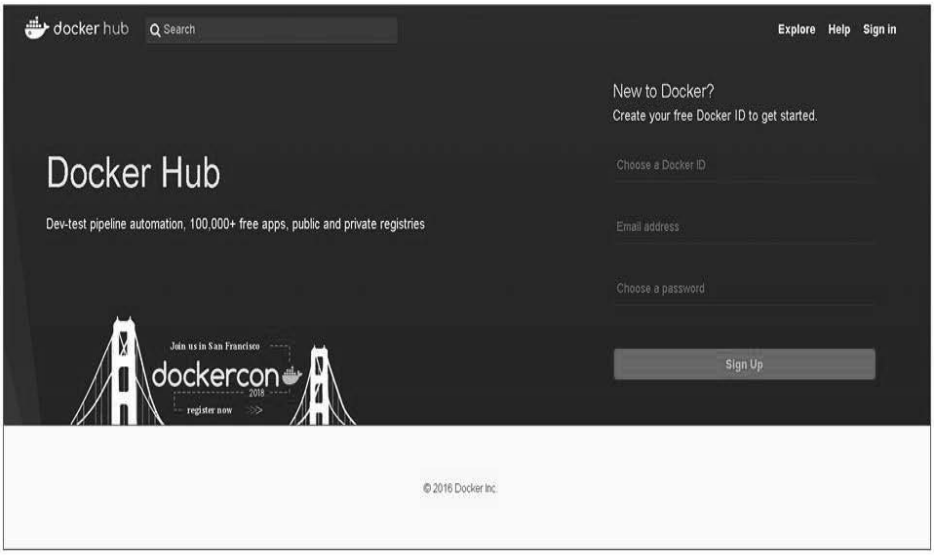
- Used to share and manage Docker images
- Docker Hub is the default registry
- There are also private registries (Artifactory, GCP, Azure, AWS, etc)

Pushing images to the Docker Hub

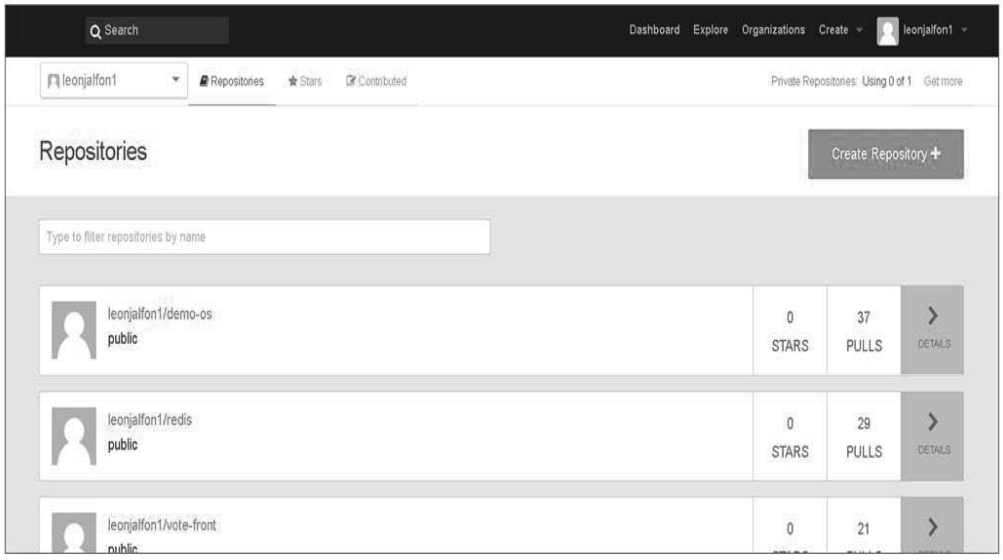
1. Create an account (free)
2. Create a new repository
3. Tag the image following the convention
4. Login to Docker Hub
5. Push the image



Push to Docker Hub – Create an account



Push to Docker Hub – Create a repository



Push to Docker Hub – Tag the Image

- To create a new tag for the image use:

```
$ docker tag <image-id> <username>/<repo-name>:<tag>
```

- The name of the image must follow the format:

  <username>/<repository>:<tag>

Push to Docker Hub – Login

- Login using the Docker Hub credentials

```
$ docker login -u <username> -p <password>
```

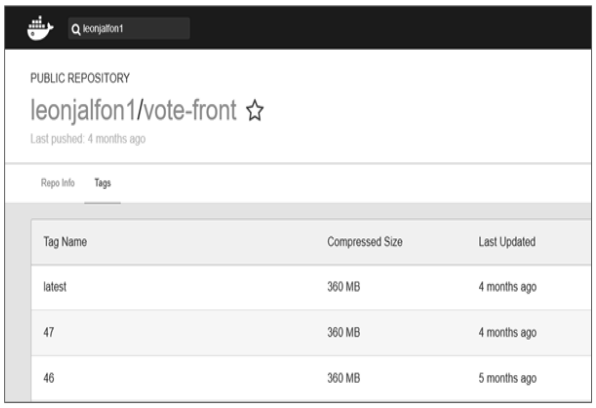
Push to Docker Hub – Push the Image

```
$ docker push <username>/<repo-name>:<tag>
```

- Then the image will be available in the docker hub
- You can see all the version pushed to the repository in the repository page

Pulling images from the Docker Hub

1. Login (for private repos)
2. Pull the image



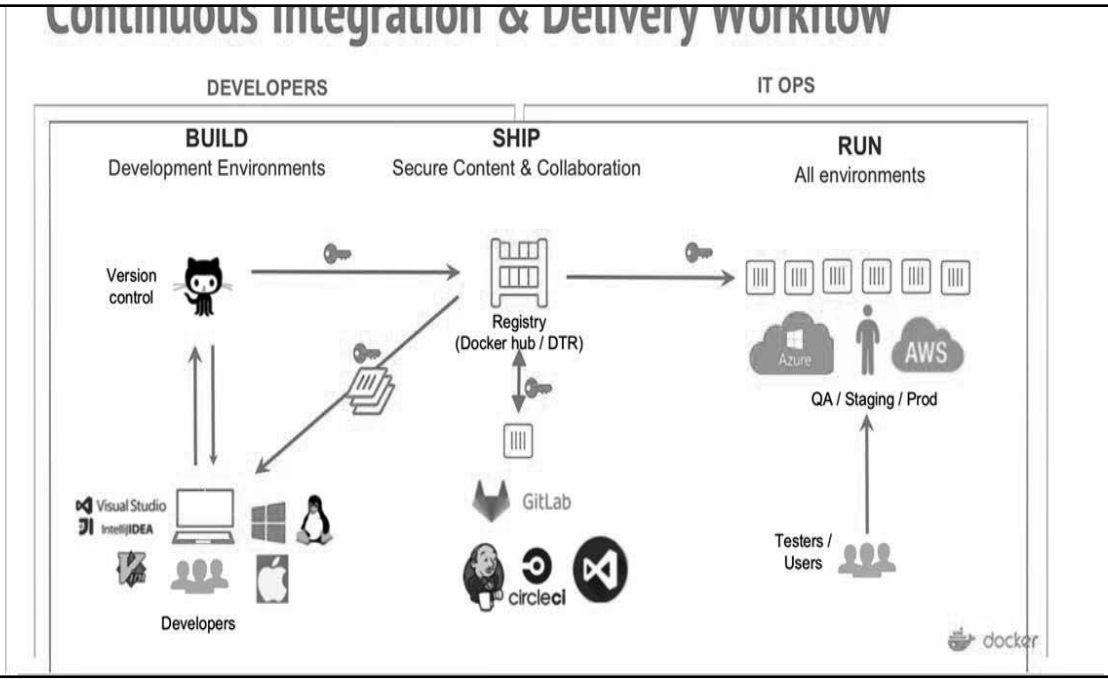

```
$ docker pull <username>/<repo-name>:<tag>
```

What can we do with all this information

- by now you know:
  - To write your app (da)
  - To "dockerize" it (docker build)
  - To run the container , control it and stop it
  - To enhance it with volumes and network
  - To control which app to run according to the label system.
  - Where does it come handy in a developer /product workflow ?

CI / CD

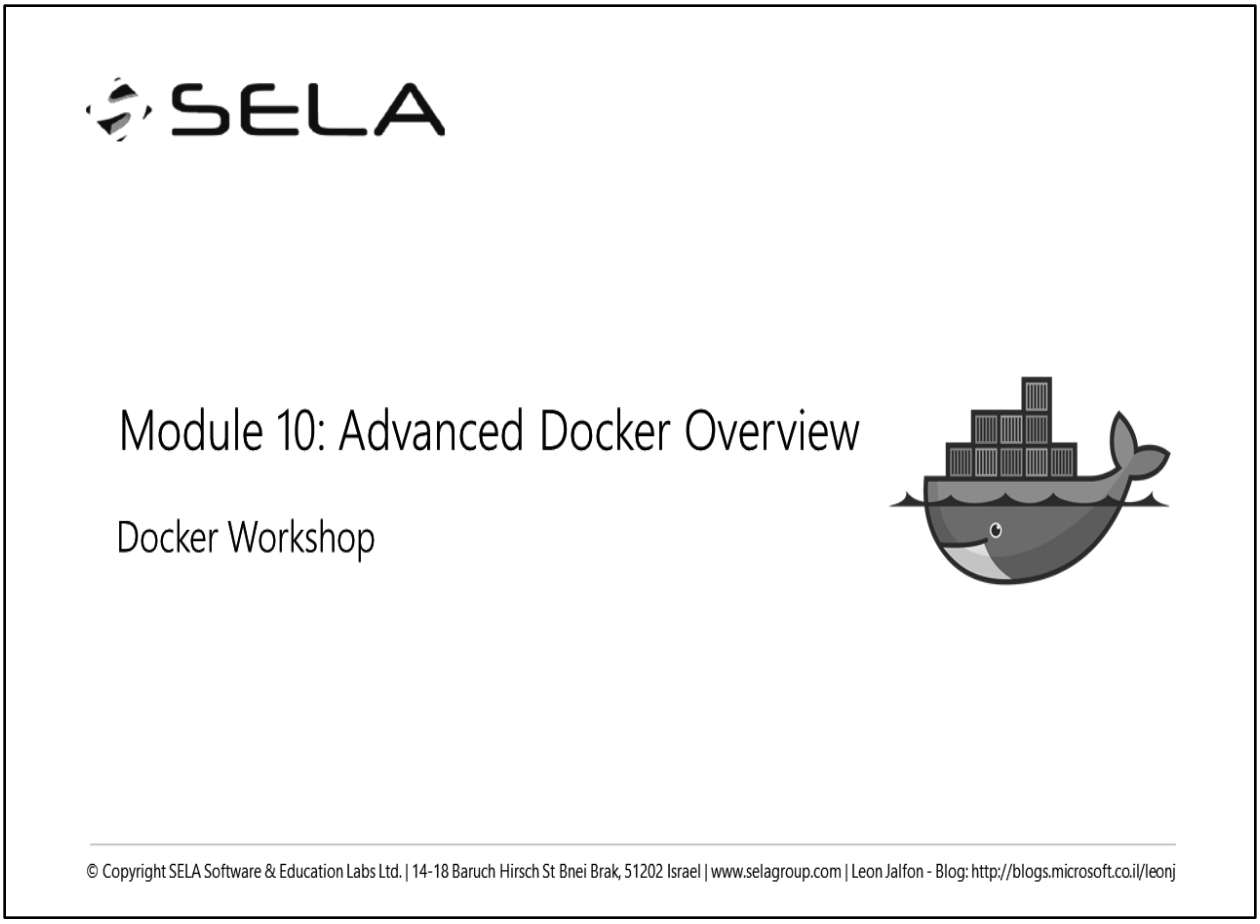
- Continuous Integration (CI):
  - a practice where you merge the work of all your developers several times a day. Each integration is verified by your automated tests and built (when it needs to be, for compiled languages mostly).
  - Why is it hard to perform CI ?
    - several branches / merge conflicts
    - Dependencies
    - Versions
    - Different tools / languages / platforms and libraries
    - Unless done effectively can slow down the dev cycle and hurt the release timeline.
- Continuous Delivery (CD) :
  - A practice where the team produce the software (release) at any time.
  - Aims for building /testing and releasing the software with greater speed and frequency



Lab: Lab 10: Working with Docker Hub (Registry)

Module 10 - Advanced Docker Overview

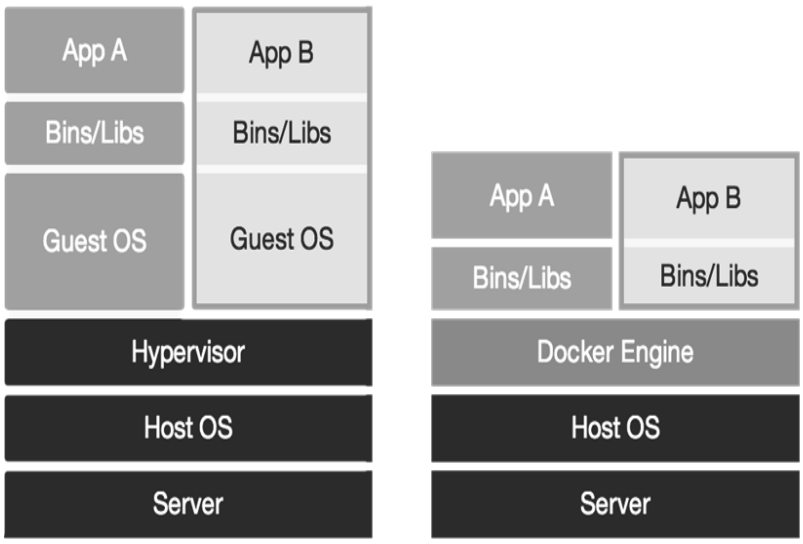
Contents:


SELA
Module 10: Advanced Docker Overview
Docker Workshop
© Copyright SELA Software & Education Labs Ltd. | 14-18 Baruch Hirsch St Bnei Brak, 51202 Israel | www.selagroup.com | Leon Jalfon - Blog: http://blogs.microsoft.co.il/leonj

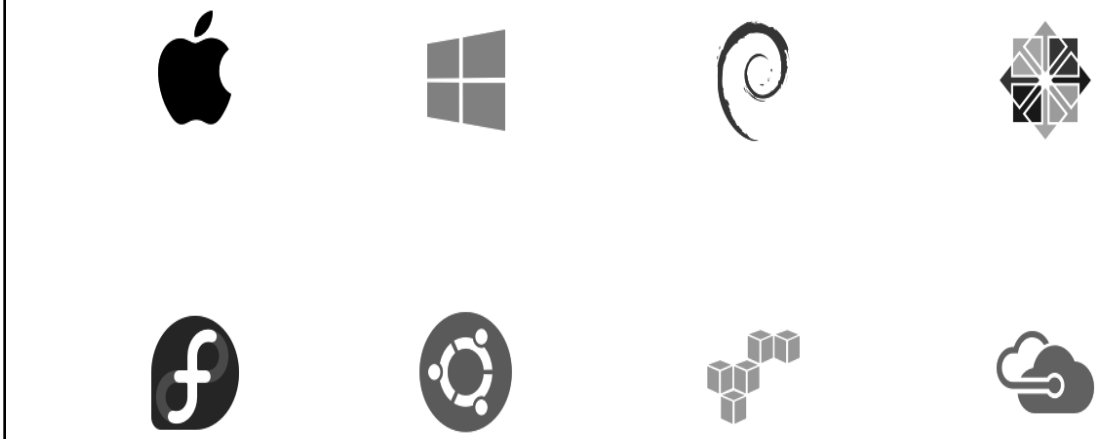
Agenda

- Basic Docker Overview
- Docker compose Introduction
- Container Orchestration Introduction
- Architecture Considerations

Why Docker?



Docker Everywhere



Docker Architecture

- **Daemon:** The Docker environment to access the Docker services.
- **Images:** Read only templates used to create containers.
- **Containers:** Running instances of an image.

Building Containers

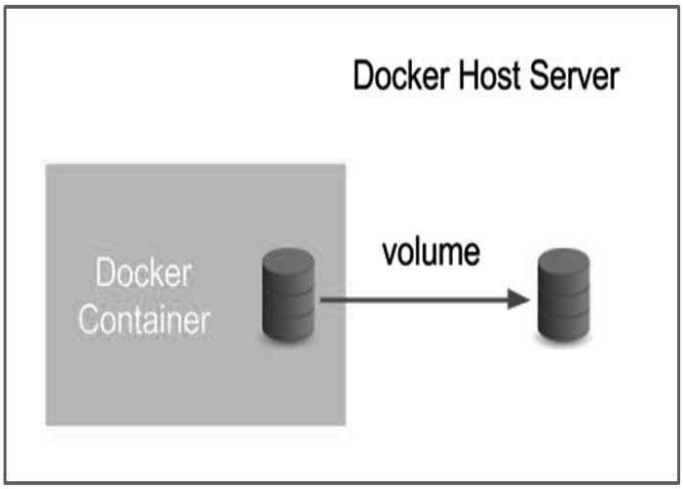
- FROM
- MANTAINER
- RUN
- CMD
- EXPOSE
- ENTRYPOINT
- ENV
- COPY
- ADD
- WORKDIR
- VOLUME

```
FROM ubuntu:14.04
RUN \
 apt-get update && \
 apt-get -y install apache2

VOLUME /myvol
ADD index.html /var/www/html/index.html

EXPOSE 80

CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

Managing Containers

- $ docker run
- $ docker ps
- $ docker images
- $ docker rm
- $ docker rmi
- $ docker attach
- $ docker exec
- $ docker save/load
- $ docker commit
- $ docker stop
- $docker kill
- $ docker start
- $ docker restart
- $ docker info
- $ docker top
- $ docker history
- $docker inspect
- $ docker logs

Docker Volumes



Docker Networks

- bridge
- host
- overlay
- macvlan
- none
- Network Plugins

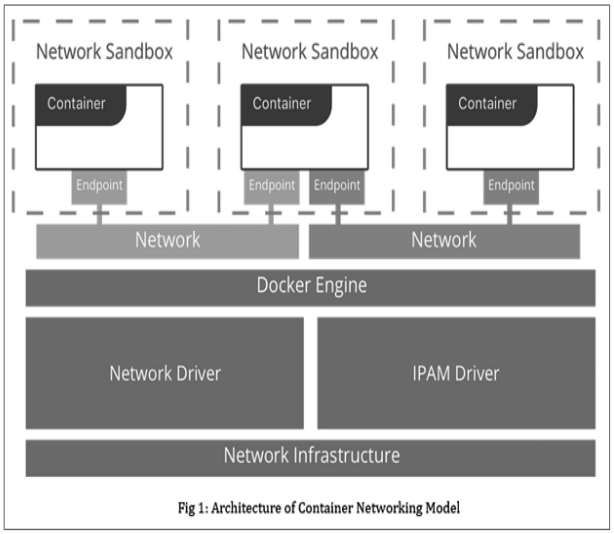


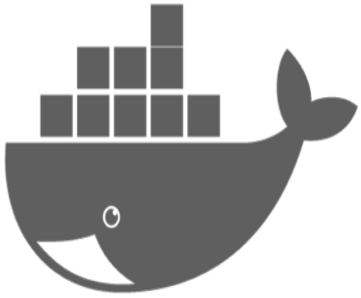
Fig 1: Architecture of Container Networking Model

Docker Registries

```
$ docker login -u <username> -p <password>
```

```
$ docker push <username>/<repo-name>:<tag>
```

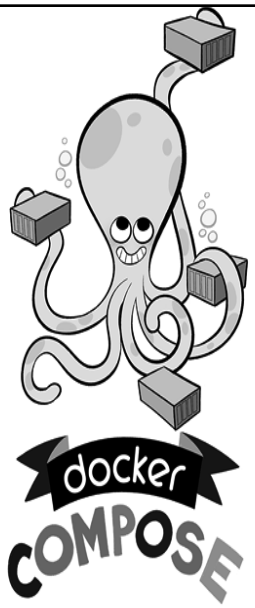
```
$ docker pull <username>/<repo-name>:<tag>
```



Containers Need Management

What is docker compose?

- Compose is a tool for defining and running complex applications with Docker.
- Define a multi-container application in a single file.
- Spin your application up in a single command.
- Docker-compose uses a YML file called "docker-compose.yml"



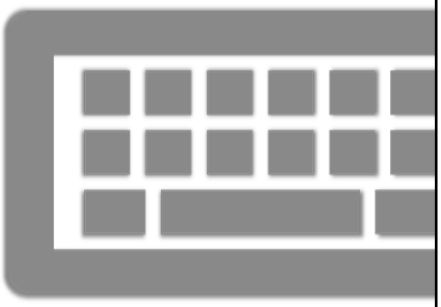
docker-compose.yml

```
version: '3'
services:
  db:
    image: postgres:9.4
    labels:
      io.skopos.singleton: '1'
      io.skopos.visual.position: 900,380
  nginx:
    depends_on:
    - vote
    deploy:
      replicas: 1
    image: datagridsys/sample-lb:1.0
    labels:
      io.skopos.dt.vote: _
      io.skopos.lb.name: vote-in
      io.skopos.lb.position: 20,80
      io.skopos.singleton: '1'
      io.skopos.visual.position: 300,80
    ports:
    - 8880:80
```

```
[root@terry dockerapp]# docker-compose build
redis uses an image, skipping
Building dockerapp
Step 1/6 : FROM python:3.5
 ---> b0d7fc8a7ace
Step 2/6 : RUN pip install Flask==0.11.1 redis==2.10.5
 ---> Using cache
 ---> b463f1061f34
Step 3/6 : RUN useradd -ms /bin/bash asiye
 ---> Running in 80c6015b840a
 ---> 3bbd504db65b
Removing intermediate container 80c6015b840a
Step 4/6 : USER asiye
 ---> Running in 4e03cf70f3b2
 ---> edc903a2502a
Removing intermediate container 4e03cf70f3b2
Step 5/6 : WORKDIR /root/dockerapp/app
 ---> cc080b462c85
Removing intermediate container 9f6441f3906c
Step 6/6 : CMD python app.py
 ---> Running in 8ab5f65de135
 ---> 1df04b7fd0d1
Removing intermediate container 8ab5f65de135
Successfully built 1df04b7fd0d1
Successfully tagged dockerapp_dockerapp:latest
[root@terry dockerapp]#
```

Lab: Lab 11: Docker Compose

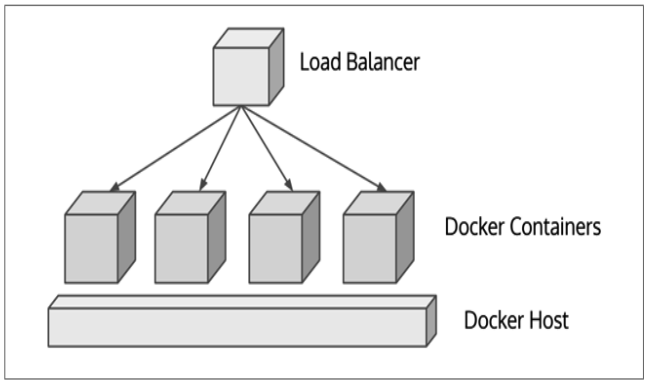
Lab



https://github.com/tshaiman/docker-workshop/lab-11

Problems with Standalone Docker

- Running a server cluster on a set of Docker containers, on a single Docker host is vulnerable to single point of failure!



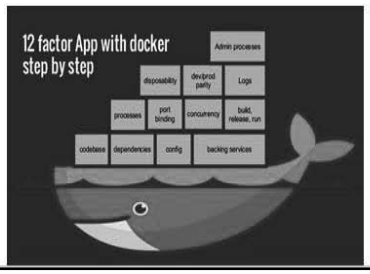
Docker Orchestrators

- Scalability
- Service Discovery
- Networking
- Volume Management
- Monitoring and Logging
- High Availability
- Load Balancing
- Health Checks
- Rolling Upgrades

Service discovery allow containers to discover other containers and establish connections to them.

Architecture Considerations

- In 2011 , Adam Wiggins , the founder of Heroku, published an article called "the twelve factor app" https://12factor.net/
- The "12-factor-app" describes 12 practices to follow when designing and architecting Software-As-A-Service (SaaS) environment.
- Those guidelines are ideal for the docker workflow

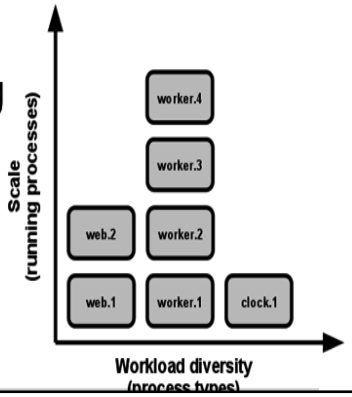


12-Factor-App

- Codebase – a single codebase tracked in revision control (each and every docker image should be build from a single source code repository)
- Dependencies – Explicitly declare and isolate dependencies
- Config – Store config in Evn. Variables not in files checked into the code base (e.g –e App_ENV=production). Keeping the configuration out of the source code helps deploy the same container to multiple environments
- Backing Services- treat backing services as attached resource

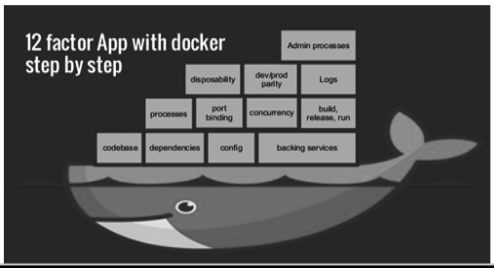
12-Factor-App

- Build, release, run –Strictly separate build and run stages.
- Processes – Execute the app as one or more stateless processes It is preferred to write an application that do not need to keep state longer than the time it takes to process and response to single request.
- Port Binding – Export services via port binding
- Concurrency - Scale out via the process model
- Disposability -Maximize robustness with fast startup and graceful shutdown



12-Factor-App

- Dev/Prod Parity – Keep Development, Staging and Production as similar as possible
- Logs – Treat Logs as event Streams . Services should not concern themselves with routing or storing logs. Events should streamed to STDOUT.
- Admin Processes - Run admin/management tasks as one-off processes.



Where To Go Next ?

- Experiment with docker with small scale on your laptop
- Gain a stronger understanding on how the pieces fit together
- Move to the cloud (AWS / Azure / Google all have free tiers)
- Learn how Kubernetes and Orchestration tool can assist you in more complex scenarios such as Service Discovery, Deployment sets, Load balancing etc.
- Spread the word.....

Happy Development !