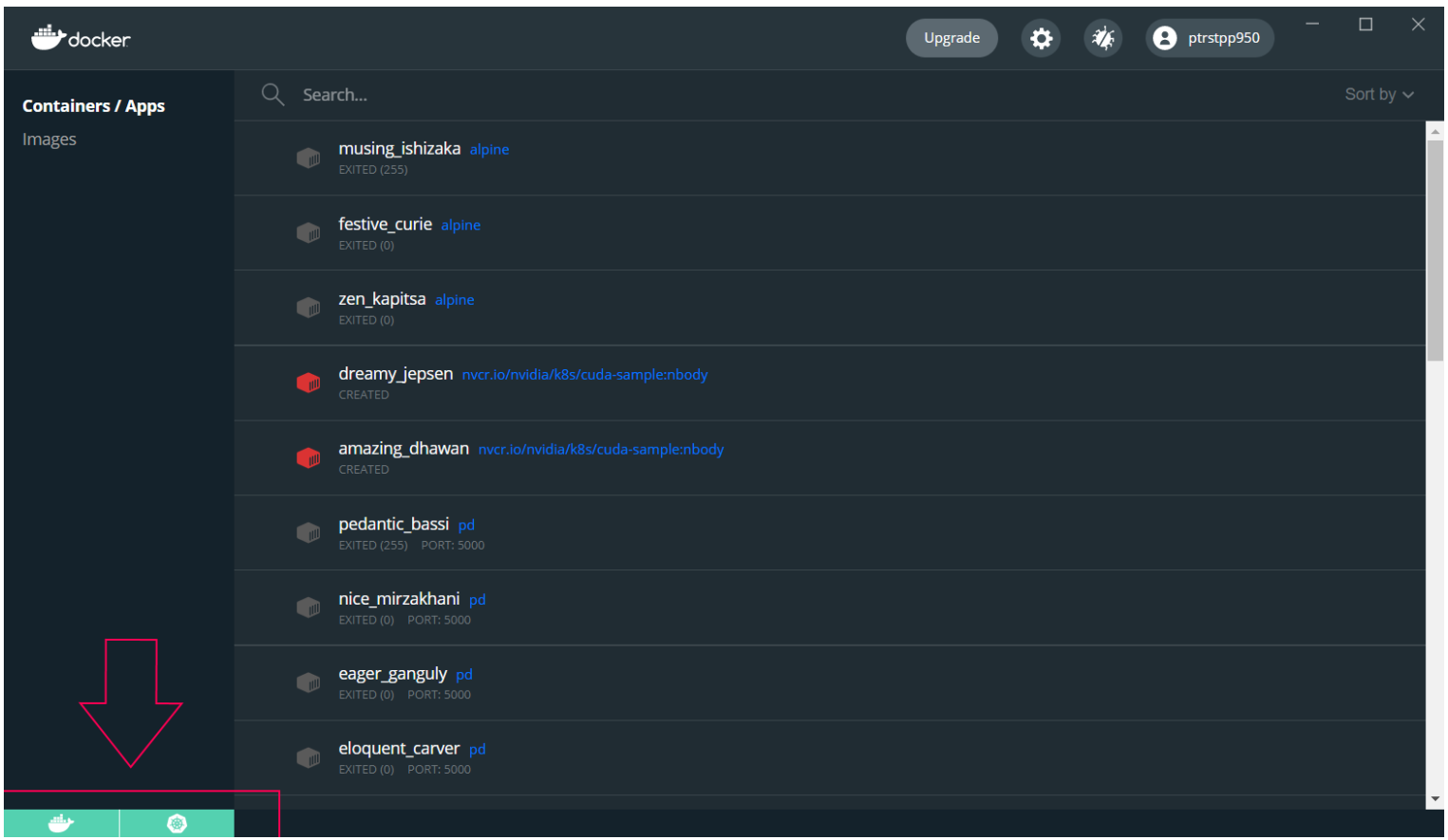
Docker++

LUXMED 2021


Before we begin
Organizational stuff

Is you PC ready?

(we will need this later)



Contract

1. Ask about anything!
2. Make notes!
3. Do your exercises, and we can guarantee you will gain understanding of how things works
4. If you feel you need a break, tell us/write us!

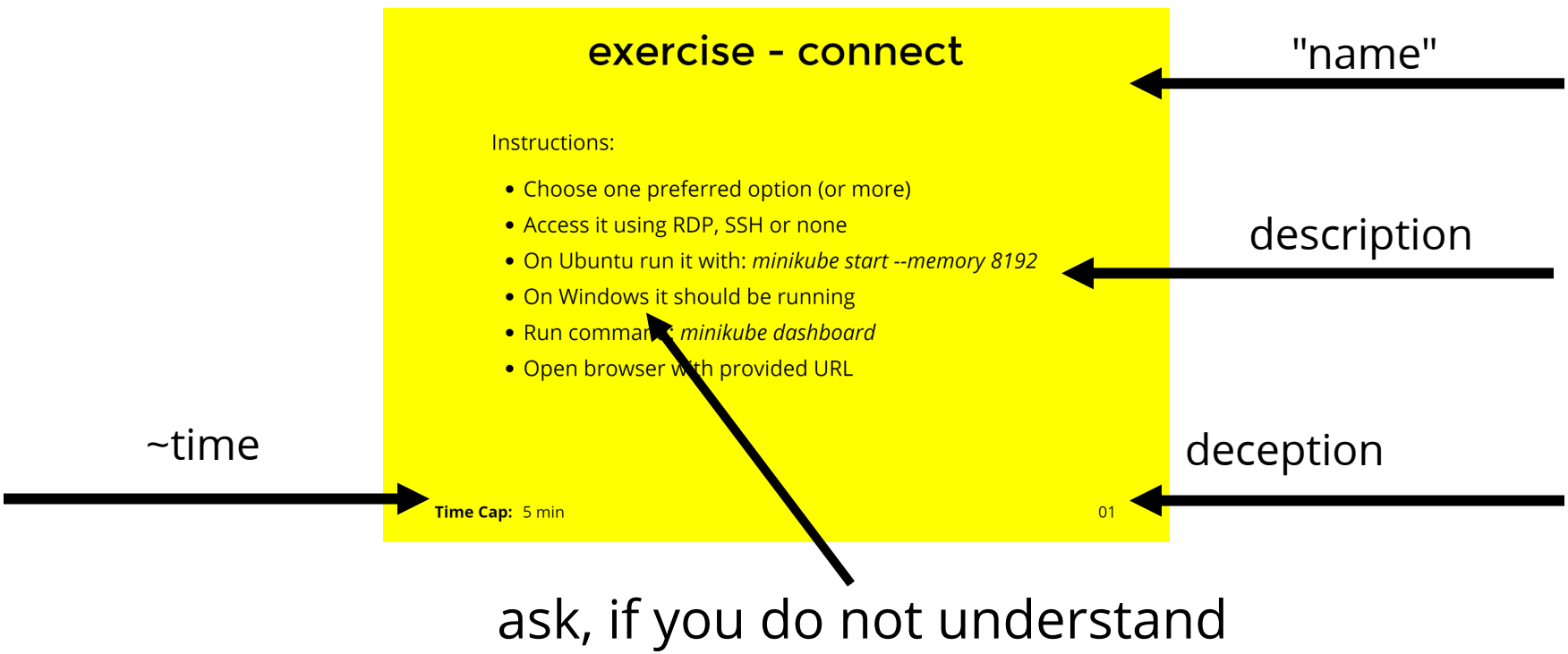
Introduction

Goal of the workshop


Follow us
https://bit.ly/docker-luxmed-2021-live
All slides
https://bit.ly/docker-luxmed-2021-all

Exercises

https://github.com/ptrstpp950/workshop-docker-101



WWW - Containers history

WWW - What? Why? When?

1979

chroot

chroot - historia

- Wprowadzone w Seventh Edition Unix
- Są “podejrzenia” że wprowadził to 1982 roku Bill Joy do kompilowania BSD 4.2 w BSD 4.1 (patrz Wikipedia)
- chroot(2) odnosi się do “system call”
- chroot(8) to “wrapper program”

chroot

```
chroot()
{
  if (suser())
    chdirec(&u.u_rdir);
}
```

- Wymaga **superuser**
- Zmienia katalog główny programu, czyli root
- Stosuje się do wszystkich procesów potomnych
- Tworzy “wirtualną” kopię systemu zwaną **chroot jail**
- Z “wewnątrz” nie ma dostępu na “zewnątrz”

"Jail"

An early use of the term "jail" as applied to chroot comes from Bill Cheswick creating a honeypot to monitor a cracker in 1991.

demo/exercise preparation WSL2/Ubuntu

```
# 1. enter home dir
cd ~

# 2. download fish shell
wget http://bit.ly/poznajdocker-fish-tar -O fish.tar

# 3. create "root" dir
mkdir container-root

# 4. enter it
cd container-root

# 5. untar fish shell into it
tar -xf ../fish.tar
```

If you use ARM run: *cat /proc/cpuinfo* | *grep model* and replace link with:

- for ARM v7: http://bit.ly/poznajdocker-fish-tar-arm
- for ARM v6: http://bit.ly/poznajdocker-fish-tar-armv6

demo/exercise preparation part 2 - Run chroot

```
# 1. enter created dir
cd ~/container-root

# 2. Create file
touch I-WAS-HERE

# 3. Check if it exists
ls

# 4. Go to home dir
cd ..

# 5. run chroot
sudo chroot container-root /usr/bin/fish

# 6. You should be in fish terminal :)
```

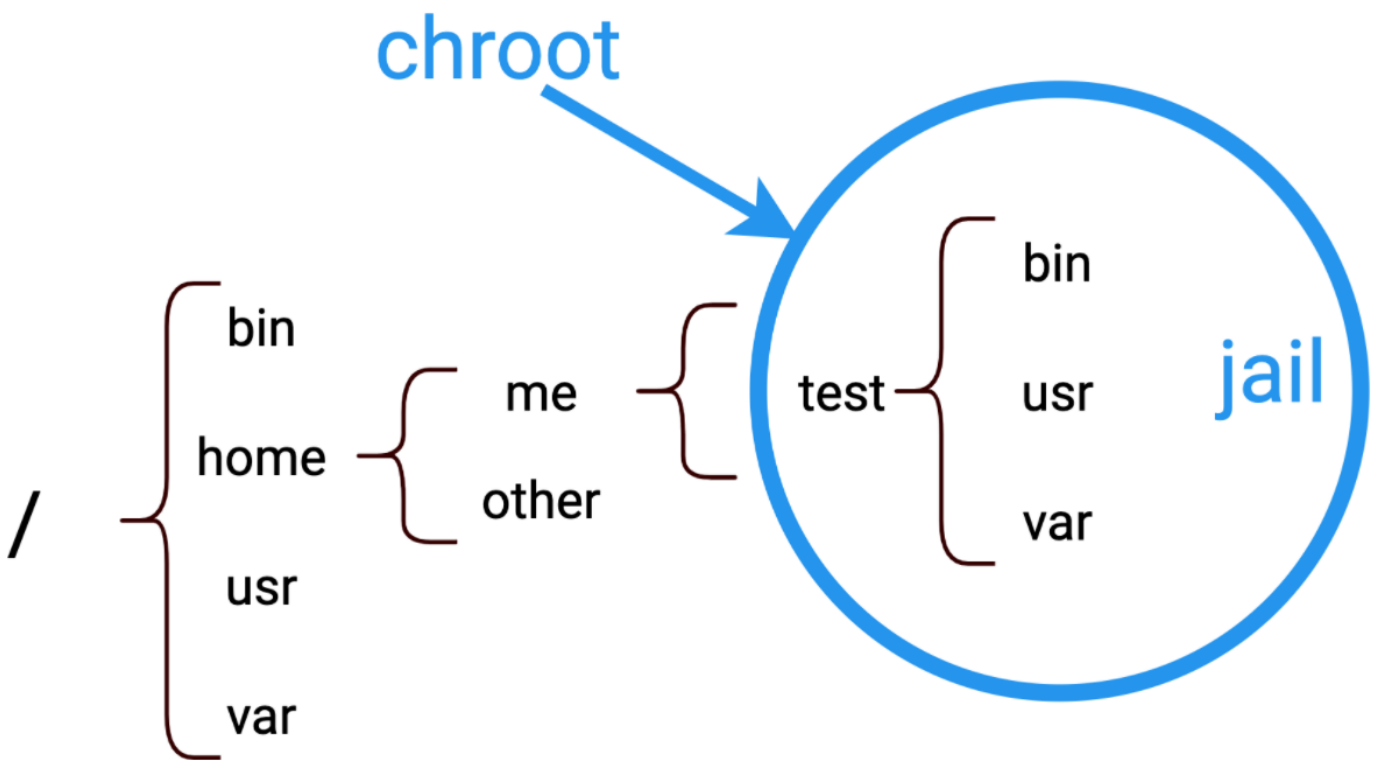
demo/exercise - kill

Instructions:

- Open second tab or in WSL you can use **SHIFT+ALT+=** or **SHIFT+ALT+|**
- Run **top** in it (on host system)
- You can exit top with **q**
- Try to kill a **top** process from host from fish shell (using **kill <PID>**)

Time Cap: 10 min

Filesystem



demo/exercise - mount

Instructions:

- In fish shell mount pseudo file system proc
- Use: **mount -t proc proc /proc**
- Use **ps aux** and compare it with **top** output from host

Time Cap: 5 min

Chroot

- Quite old 😅
- Processes are limited to and only to the part of the filesystem
- Host and chroot process share:
  - System resources
  - Running processes
  - Network
  - Users

2000

FreeBSD jails

FreeBSD jails - 4 elements

- A directory subtree: the starting point from which a jail is entered.
- A hostname: which will be used by the jail.
- An IP address: which is assigned to the jail. The IP address of a jail is often an alias address for an existing network interface.
- A command: the path name of an executable to run inside the jail. The path is relative to the root directory of the jail environment.

Improved security

- No process access outside jail
- No kernel modifications and module loading from jail (most of *sysctls* and *securelevel* is forbidden)
- Network configuration (including routing table) if forbidden
- There is a separate VNET for jail
- *Mount* & *unmount* for filesystems is forbidden
- Jails cannot create devices nodes

2001

Linux-VServer

Linux-VServer

- New term: Virtual Private Server (VPS)
- System core modification for creating operating system-level virtualization
- Similar to FreeBSD jails
- Linux VServer != Linux Virtual Server
- Last stable from 2008
- Last preview from 2019

2002

namespaces

Namespaces

Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources.

Namespaces

- First namespace - **mount** in 2002
- Others are:
  - Proces IP (pid)
  - Network (net)
  - Interprocess Communication (ipc)
  - UTS (UNIX Time-Sharing)
  - User
  - Control group (cgroup)
  - Time Namespace

demo / exercise - unshare

Instructions:

- We will use **unshare** - command creates new namespaces and then executes the specified program
- Run **hostname** and remember it (put it in Notepad)
- Run **sudo hostname <your new name>**
- Check hostname in second shell
- Revert change
- Repeat it in new shell created with unshare command: **sudo unshare --uts /bin/bash**
- Does it affected the host?

Time Cap: 5 min

Demo unshare

```
sudo unshare --uts --pid --fork --mount-proc \
chroot "container-root" \
   /bin/sh -c \
     "/bin/mount -t proc none /proc && \
      hostname not-yet-a-docker-demo && \
      /usr/bin/fish"
```

- **--uts / -u** - hostname separation
- **--pid / -p** - PID separation
- **--fork / -f** - run it as fork instead of full process (in our case chroot)
- **--mount-proc / -m** - run "mount" for proc filesystem with empty dir before run

demo / exercise - CPU bomb

Instructions:

- Run command from previous demo
- Use **yes > /dev/null &** command to run "heavy" process for one core
- Verify CPU usage with **htop** or Windows Task Manager
- Clean (kill) processes with **killall yes**

```
sudo unshare --uts --pid --fork --mount-proc \
chroot "container-root" \
   /bin/sh -c \
     "/bin/mount -t proc none /proc && \
      hostname not-yet-a-docker-demo && \
      /usr/bin/fish"
```

Time Cap: 5 min

2004

Solaris Containers

Solaris Containers/Zones

- Name changed a few times - currently it is "zones"
- Another solution for operating system virtualization
- Main new features:
  - states
  - snapshot + cloning

States in Solaris Zones

- Configured
- Incomplete
- Installed
- Ready
- Running
- Shutting down
- Down

Snapshot + cloning

- Requires ZFS - Zettabyte file system
- Only differences in image are saved
- Allows to create new zones in seconds

2005

Open Virtuozzo / OpenVZ

Open Virtuozzo

- Created by Parallels (to be precise SWSoft acquisition was in 2004)
- Based on Linux Kernel 2.6.32 form Red Hat Enterprise Linux 6.
- Last stable from 2015

New features

- Advanced resource management:
  - CPU scheduler using fair-share scheduling strategy
  - I/O scheduler
  - User Beancounters - per-container counters, limits and guaranteesBased on Linux Kernel 2.6.32 form Red Hat Enterprise Linux 6
- Checkpointing
- Live migration

2006 & 2007

cgroups

cgroups

- "Process containers" (first name) created by Paul Menage and Rohit Seth from Google
- Full name is control groups from 2007
- Included in 2.6.25 kernel - released in January 2008
- Still in active development
- Version 2 was included in 4.5 kernel in 2016

New features

- Limits and isolation for CPU, memory, I/O, network and more
- Single process can use only dedicated part of resources
- V2 introduces
  - Single process hierarchy
  - Separate threads and processes

demo / exercise - cgroups

Preparations:

- Install tools with: **sudo apt-get install cgroup-tools**
- Run following commands

```
# 0. Install tools
sudo apt-get update
sudo apt-get install cgroup-tools

# 1. Create unique identifier
uuid="cgroup_demo"

# 2. Create cgroup with cgcreate
sudo cgcreate -g "cpu,cpuacct,memory:$uuid"

# 3. Create CPU piority
sudo cgset -r cpu.shares=512 "$uuid"

# 4. Add memory limits
sudo cgset -r memory.limit_in_bytes=10000 "$uuid"
```

Time Cap: 1 min

demo / exercise - cgroups

Preparations part 2:

Run process in cgroup using

```
sudo cgexec -g "cpu,cpuacct,memory:$uuid" \
    unshare -fmuipn --mount-proc \
    chroot container-root \
     /bin/sh -c \
     "/bin/mount -t proc none /proc && \
      hostname cgroups-hostname &&  \
      /usr/bin/fish"
```

Time Cap: 1 min

home exercise - cgroups

Instructions:

- Using **sudo cgset -r memory.limit_in_bytes=X "$uuid"**
- Try to find minimal value for X
- Tip: **sudo cgget $uuid | grep <something ;)>**

```
# 1. Set memoty limit
sudo cgset -r memory.limit_in_bytes=X "$uuid"

# 2. Run
sudo cgexec -g "cpu,cpuacct,memory:$uuid" \
    unshare -fmuipn --mount-proc \
    chroot container-root \
     /bin/sh -c \
     "/bin/mount -t proc none /proc && \
      hostname cgroup-hostname &&  \
      /usr/bin/fish"
```

Time Cap: 5 min

exercise (optional) - cgroups + CPU

Instructions:

- Using **cpu.cfs_period_us** and **cpu.cfs_quota_us** try to limit CPU
- **us** stands for µs - microseconds

```
uuid="cgroup_limit_cpu"
sudo cgcreate -g "cpu,cpuacct,memory:$uuid"
sudo cgset -r cpu.cfs_period_us=200000 "$uuid"
sudo cgset -r cpu.cfs_quota_us=1000000 "$uuid"
sudo cgexec -g "cpu,cpuacct,memory:$uuid" \
    unshare -fmuipn --mount-proc \
    chroot container-root \
     /bin/sh -c \
     "/bin/mount -t proc none /proc && \
      hostname poznaj-docker-demo && \
      /usr/bin/fish"

# inside use
yes > /dev/null &
# outside use to monior usage
htop
```

Time Cap: 5 min

2007

AIX Workload partitions (WPARs)

WPARs

- Be careful what you type. It is not a WARP 😅
- IBM shows that big ones also wants to optimize load 🙈🙉🙈
- Works from IBM AIX 6.1
- It has 2 parts:
  - WPAR system - full AIX installation
  - WPAR application - lightweight environment with one working process.

New features

- Versioning (from AIX 7.1)
- Mobility:
  - Move load form one psychical machine to an other
  - It works for both - system and application parts
- Why "mobility"?
  - Planning upgrades or downtimes
  - Too heavy usage

2008

LXC (Linux Containers)

LXC

- Nothing new 🤷. Just bunch of tools
- It uses:
  - cgroups
  - namespaces
- Low-level tool
- Docker used LXC in first versions

2013

Docker

Docker

- Nothing new, but 🤯
- All tools in one API/CLI
- In first versions only LXC overlay:
  - Optional from 0.9
  - Dropped from 1.10

What's inside?

Docker engine

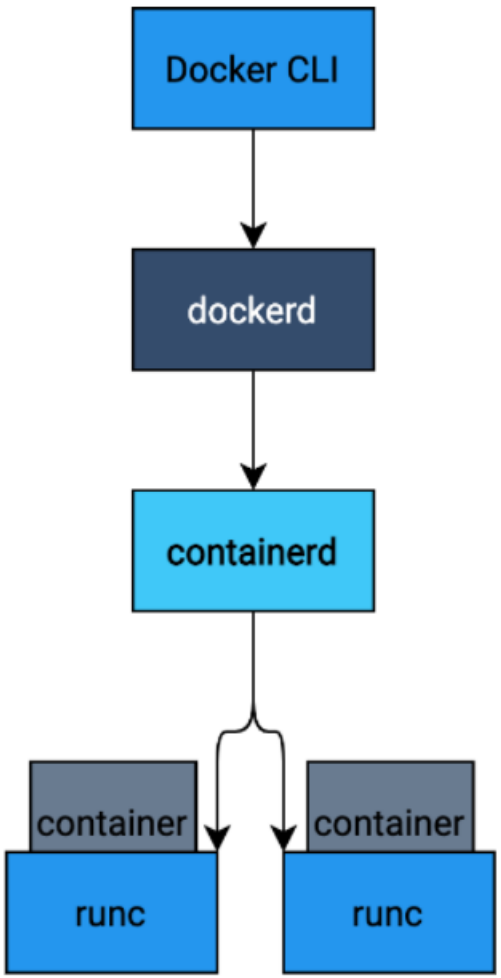
Docker CLI

- "Simple" CLI
- Interaction with:
  - dockerd
  - cloud
  - docker swarm

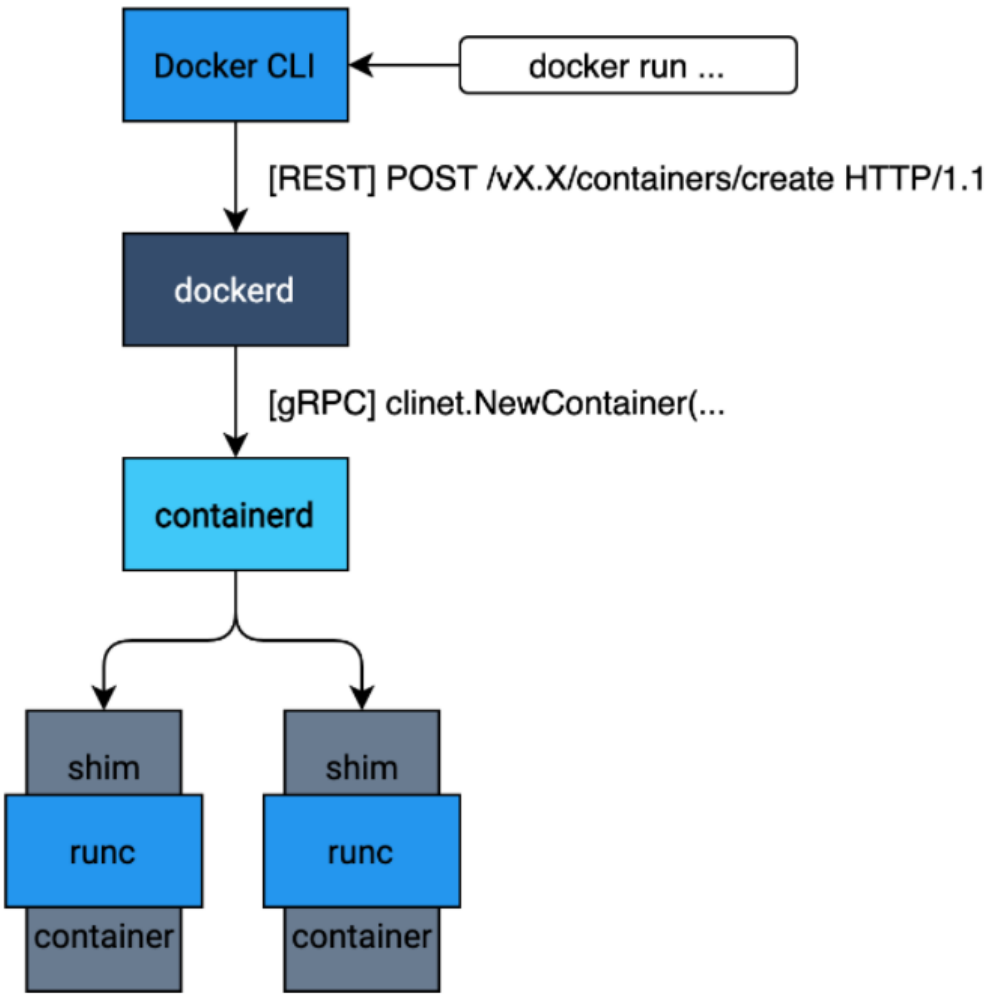
dockerd

- dockerd == docker deamon
- All functionality

containerd & runc



containerd & runc



demo / exercise - runc

Instructions:

- Install runc with: **sudo apt-get install runc**
- Open home dir: **cd ~**
- Create config.json with **runc spec --rootless**
- Replace in line 10: "sh" -> "fish"
- In line 54: "path": "rootfs" => "path": "container-root"
- Run container with: **runc --root /tmp/runc run my-first-container**
- Try to create dir
- Tip: search in config.json for readonly 😎

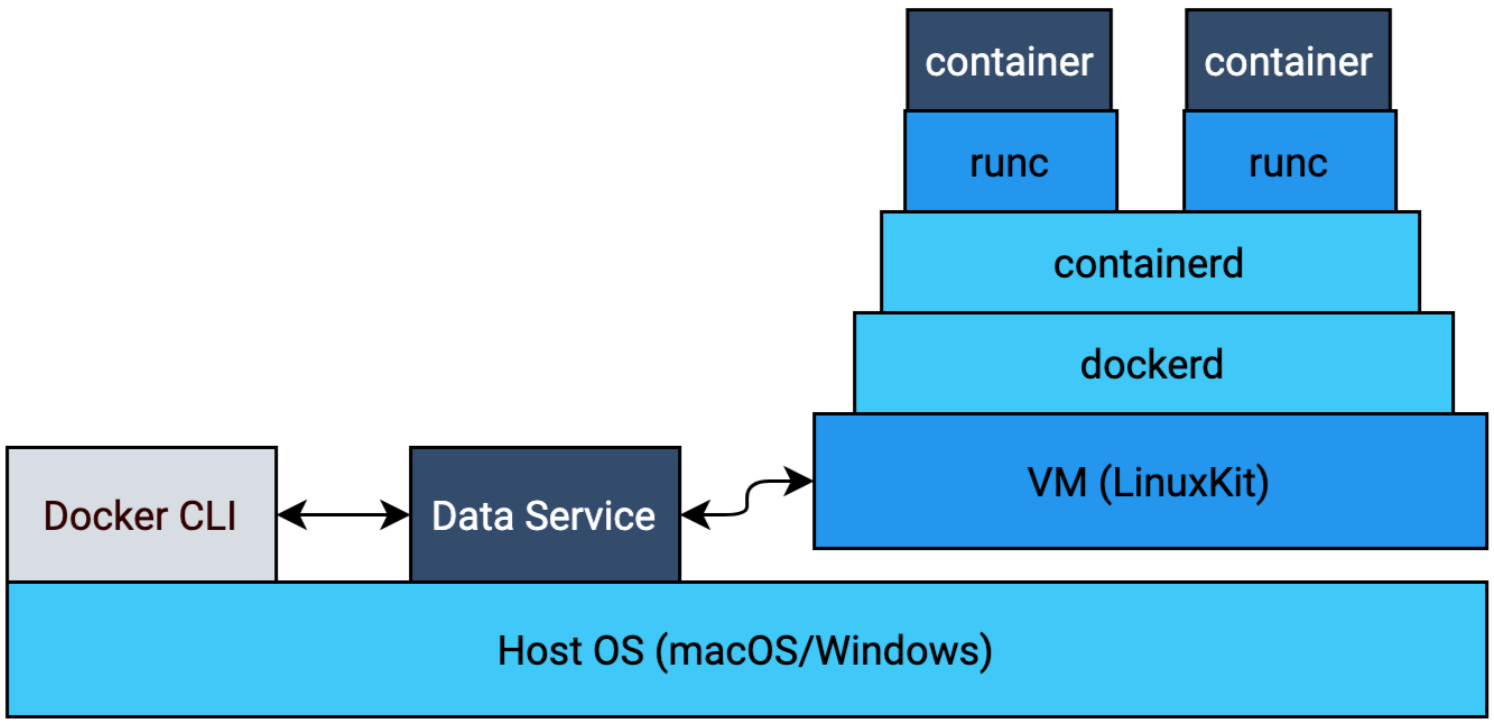
Time Cap: 10 min

Docker Desktop

Docker for Desktop

- For macOS and Windows
- (if we talk about Linux conatiners)
- Virtual machine or WSL

Docker for Desktop



Moby project

The Moby project

- Moby is an open framework (an open source kind project)
- It is a "lego set" of tools to create nonstandard solutions based on containers
- Experiments and ideas exchange
- Most know example: BuildKit (included form Docker 20.10)

exercise - first Docker run

Instructions:

- Run fish shell using Docker😎
- Just simple run cmd: **docker run -it frapsoft/fish fish**

Time Cap: 5 min

Discussion time!

What is Docker?

Running containers

docker help

- Shows all command options
- Better than Google 😅

```
docker [OPTIONS] CMD

OPTIONS:
[...]
```

docker ps

- List containers
- By default only running

```
docker ps [OPTIONS]

OPTIONS:
-a, --all
-f, --filter filter
    --format string
-n, --last int
-l, --latest
    --no-trunc
-q, --quiet
-s, --size
```

docker kill

- Kill 😎
- One or more running containers
- Can send signal to the container

```
docker kill [OPTIONS] \
  CONTAINER [CONTAINER]

OPTIONS:
 -s, --signal string
```

exercise - kill it

Instructions:

- Run (or use already running) fish shell
- Simple run cmd: **docker run -it frapsoft/fish fish**
- How can you kill?
- Tip: at least 3 ways 😅

Time Cap: 5 min

docker run

- Run a container from image
- Allows to run specific command
- It has multiple options

```
docker run [OPTIONS] \
  IMAGE \
  [COMMAND][ARG...]


OPTIONS:
 Next slides :)
```

demo - run

Instructions:

Simple run cmd: **docker run docker/getting-started**

docker run -d

- Run in background
- Waits for "kill"
- Creates a demon 😅

```
docker run \
  --detach IMAGE
```

```
docker run -d IMAGE
```

demo - run + detach

Instructions:

Simple run cmd: **docker run -d docker/getting-started**

How to "enter"?

- interactive - force open STDIN
- tty - pseudo TeleTYpewriter allocation => STDOUT to terminal

```
docker run \
--interactive \
--tty \
IMAGE COMMAND
```

```
docker run -it \
IMAGE COMMAND
```

demo - run + detach

Instructions:

- Simple run cmd: **docker run -it docker/getting-started**
- Better: **docker run -it docker/getting-started /bin/sh**

Logs

- Logs from STDOUT
- We can access running and killed

```
docker logs \
  [OPTIONS] ID

OPTIONS:
      --details
-f, --follow
      --since string
      --tail string
-t, --timestamps
      --until string
```

exercise - find logs

Instructions:

- Run a container (if you didn't already)
- Use **docker ps** find id
- Use **docker logs** read logs
- Tip: do you remember about **-a/--all** option?

Time Cap: 5 min

RM

- RM == remove
- We can remove a running container

```
docker rm [OPTIONS] \
    CONTAINER \
    [CONTAINER...]

OPTIONS:
  -f, --force
  -l, --link
  -v, --volumes
```

exercise - clean up time!

Instructions:

- Clean all old containers
- Use **docker ps** with options :)
- Use **docker rm** to clean
- Bonus: can you create one-liner?
- Bonus tip: try **-q/--quiet** option in docker ps

Time Cap: 5 min

run + rm

- It is just docker run + docker rm in one :)
- It isn't default because logs can be useful ;)

```
docker run --rm IMAGE
```

demo - run rm

Instructions:

- Run: **docker run --rm -it docker/getting-started /bin/sh**

Where are we?

- We run first container
- We can run containers and do clean up
- We can access with interactive mode
- We can get logs
- Still we don't know how to access web application inside using curl 😅

Exposing ports

Exposing ports

- We need it to access web applications
- We can do port mapping (a port from host to a port in container)
- Most images don't need detach option, but it can be useful
- We have following options:
  - None
  - Expose port in docker network only
  - Publish single port to host
  - Publish specific ports to host
  - Publish all ports to host

docker inspect

- Helper
- Allow to view container properties
- Examples:
  - IP
  - Status
  - Exit code
  - Arguments

```
docker inspect \
  [OPTIONS] \
  CONTAINER

OPTIONS
  -f, --format string
  -s, --size
      --type string
```

demo - inspect

Instructions:

- Run: **docker run -d docker/getting-started**
- Run: **docker inspect [ID]**
- Run: **docker inspect [ID] -f '{{.State.Status}}'**

expose

- Expose a port, but only in internal network
- We can access it from other running container

```
docker run \
  --expose PORT \
  IMAGE
```

demo - expose

Instructions:

- Run: **docker run -d --expose 80 docker/getting-started**
- Run: **docker inspect [ID]** to find IP
- Run: **docker run -it docker/getting-started /bin/sh**
- Query IP on port 80 from new container :)

publish

- Expose a port and publish it to the host using mapping
- **HOST** part can be a port or a IP:PORT pair
- TYPE:
  - tcp - default
  - udp
  - sctp

```
docker run \
  --publish \
    HOST:PORT/TYPE \
  IMAGE

docker run \
  -p HOST:PORT/TYPE \
  IMAGE
```

demo - publish

Instructions:

- Run: **docker run -d -p 8080:80 docker/getting-started**

- Open browser on 127.0.0.1:8080

docker port

- Show all mappings

```
docker port CONTAINER
```

publish part 2

- Publish supports also a port range

```
docker run \
  -p HOST_RANGE:RANGE \
  -p HOST_PORT:PORT \
  IMAGE
```

demo - publish

Instructions:

- Run: **docker run -d -p 8050-8060:70-80 docker/getting-started**

- Run **docker ps** to view all mapping

- Run **docker port [ID]** to view clear mappings :)

publish-all

- Publish all declared ports
- Maps to random ports

```
docker run \
  --publish-all \
  IMAGE
```

```
docker run -P IMAGE
```

Expose + Publish

- We have following options:
-
  - None
  - Expose port in docker network only using **--expose**
  - Publish single port to host using **-p/--port**
  - Publish specific ports to host using **-p/--port**
  - Publish all ports to host **-P/--publish-all**

env

- Pass environment variables

```
docker run \
  --env VAR1=VALUE1 \
  --env VAR2=VALUE2 \
  IMAGE

docker run \
  -e VAR1=VALUE1 \
  -e VAR2=VALUE2 \
  IMAGE
```

env from host

- Pass environment variables existing on host

```
export VAR1=VALUE1
export VAR2=VALUE2

docker run \
  --env VAR1 \
  --env VAR2 \
  IMAGE
```

env from file

- Pass environment variables from env file

```
docker run \
  --env-file FILE.ENV
  IMAGE
```

```
cat FILE.ENV
# comment
VAR=VALUE
VAR_FROM_EXPORT
```

Ephemeral

Containers are ephemeral:

- Short-lived
- Temporary
- Transient

Theory: 12 factor -> 6 - Processes

docker exec

- Enter working container
- Or run a command on it
- Almost same options as run

```
docker exec [OPTIONS] \
  CONTAINER \
  COMMAND [ARG...]

OPTIONS:
 -d, --detach
     --detach-keys string
 -e, --env list
 -i, --interactive
     --privileged
 -t, --tty
 -u, --user string
 -w, --workdir string
```

docker stop

- Stops containers gracefully
- Biggest difference versus kill - grace period
- Default time is 10 seconds

```
docker stop [OPTIONS] \
  CONTAINER \
  [CONTAINER ..]

OPTIONS:
  -t, --time int
```

docker start

- Starts stopped containers

```
docker start [OPTIONS] \
  CONTAINER \
  [CONTAINER ..]

OPTIONS:
 -a, --attach
     --detach-keys string
 -i, --interactive
```

demo - exec

Instructions:

- Run: **docker run -d docker/getting-started**
- Run **docker exec -it [ID] /bin/sh** to run shell on container

exercise - a riddle with ports

Part 1

Instructions:

- Run: **docker run -d poznajdocker/random-port**
- It will expose simple web app on random port
- Try to access web application root
- Tip: use docker log to find port
- Tip: use docker exec access app

Time Cap: 10 min

exercise - a riddle with ports

Part 2

Instructions:

- Using image: **poznajdocker/random-port**

- Try to set specific port

- Variable to set is **ASPNETCORE_URLS="http://*:0"**- ZERO is responsible for random

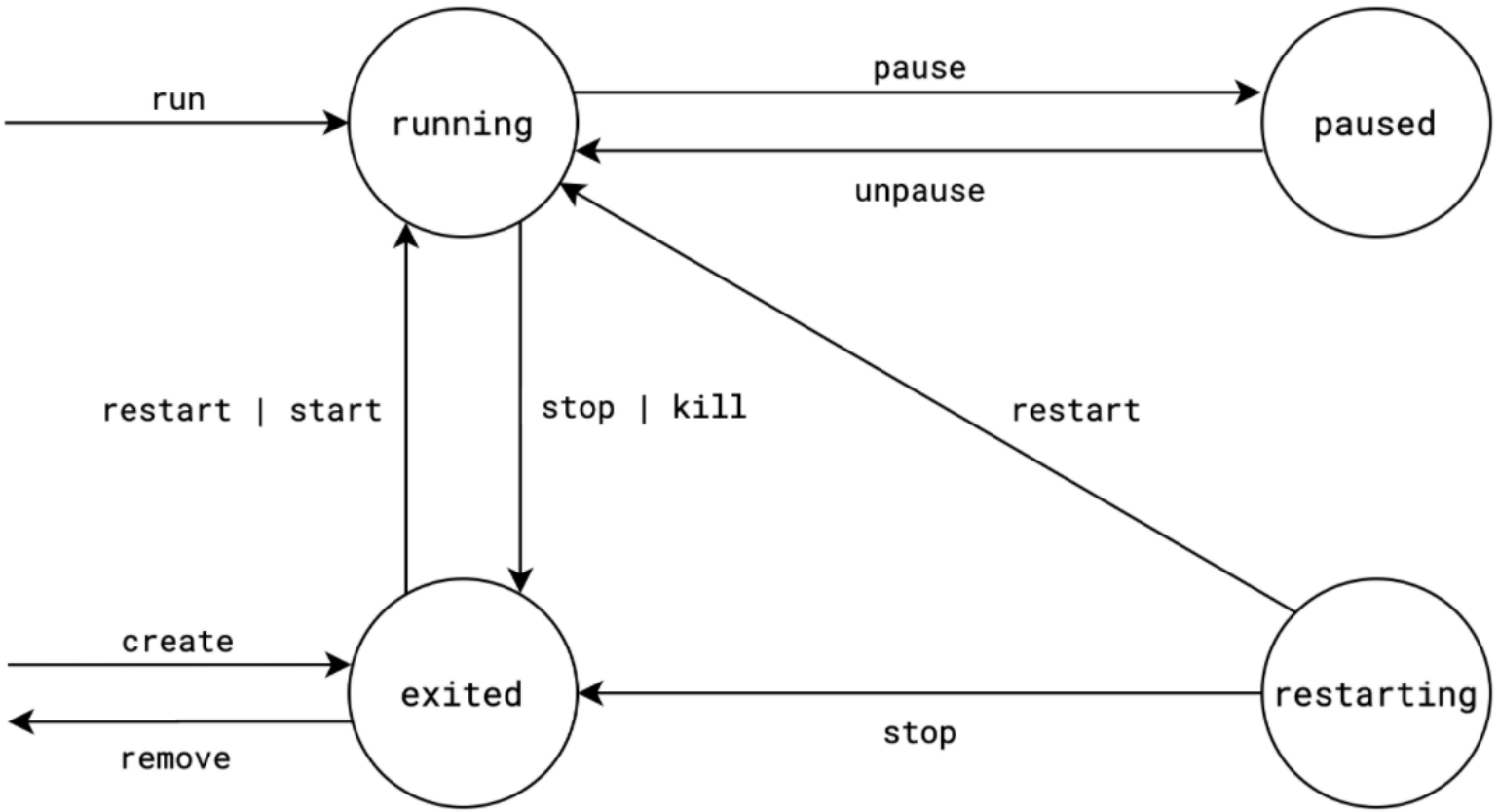
- Tip: use env option

- Bonus: try to create env file

Time Cap: 10 min

Running containers summary

State machine



Summary

- We know how to run containers
- We know how to expose/publish ports
- We know how to pass environment variables
- We know docker container states

Resource limits

Managing resources

- Default is unlimited
- Docker allows to configure:
  - memory limits
  - CPU limits
- Some options depends on host

docker info

- Shows system info
- Mark as WARNING not available options

```
docker info [OPTIONS]

OPTIONS:
  -f, --format string
```

docker stats

- Shows current usage
- Two versions available:
  - Static
  - Live

```
docker stats [OPTIONS]

OPTIONS:
  -a, --all
      --format string
      --no-stream
      --no-trunc
```

demo - stats

Instructions:

- Run multiple containers for example with:

docker run -d docker/getting-started

- Run **docker stats** to see usage

Memory limits

- Docker can use:
  - hard limits
  - soft limits
- There are a lot of options

Memory limits - rules

- You shouldn't share all host memory
- Be careful with OOM (Out of memory exception)
- When Liniux detects OOM it stars to kill processes including docker to save core system functionality
- Docker tries to reduce risk setting priority to Out of Memory Managment

Attention!!!

Attention with safeguards to Out of Memory Managment:

- Be careful with --oom-score-adj option and do not assign big negative number to it
- Same with --om-kill-disable on container

How to reduce risk?

- Use host with enough resources
- Use memory limits 😎

memory

- Basic options
- Limits memory
- Kills a container when exceeded
- Options:
  - b, k, m, g
  - minimum: 4m

```
docker run \
  --memory 4m \
  IMAGE

docker run -m 4m IMAGE
```

exercise - memory

Part 1

Instructions:

- Run:

docker run --rm -d -p 8090:8080 poznajkubernetes/pkad

- Run **docker stats**

- Check system usage with: **top**, **htop** or Task Manager

- Open browser at http://127.0.0.1:8090/-/mem

- Use "Allocate 500 MiB" and observe results

Time Cap: 5 min

exercise - memory

Part 2

Instructions:

- Add **-m XXX** option to previous exercise
- Try to find minimum possible value to run application
- What happens when memory exceeded limit?

Time Cap: 5 min

CPU

- By default all host CPU can be used
- Most users uses default CFS scheduler.
- We can also use realtime scheduler.

CPU

- By default all host CPU can be used
- Most users uses default CFS scheduler.
- We can also use realtime scheduler.

CFS

- CFS is Completely Fair Scheduler
- Implementation is based on Red Black Tree with timeline

Options

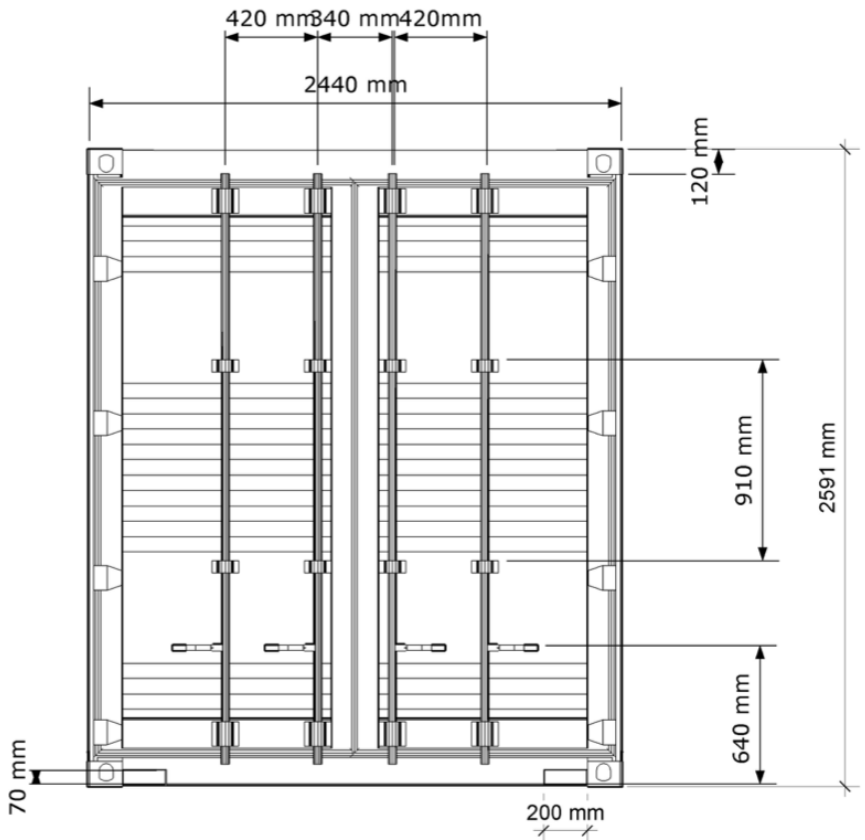
- --cpus=<value> - if you set --cpus=2.5 on 4 cores machine, at most 2.5 core will be assigned to container
- --cpu-period - sets CFS scheduler period
- --cpu-quota - sets CFS quota in container
- --cpuset-cpus - assigned a processor instance to container
- --cpu-shares - soft limit

GPU

- Docker can access GPU from container
- Currently NVIDIA GPU is supported

Images

Image (and layers)



Container


602020 8
22G2
MAX. GROSS
24,000 KG
52,910 LB
TARE
3,200 KG
7,055 LB
NET
28,800 KG
45,855 LB
CU. CAP.
31.0 CU.M.
1,093 CU.FT.

image vs container

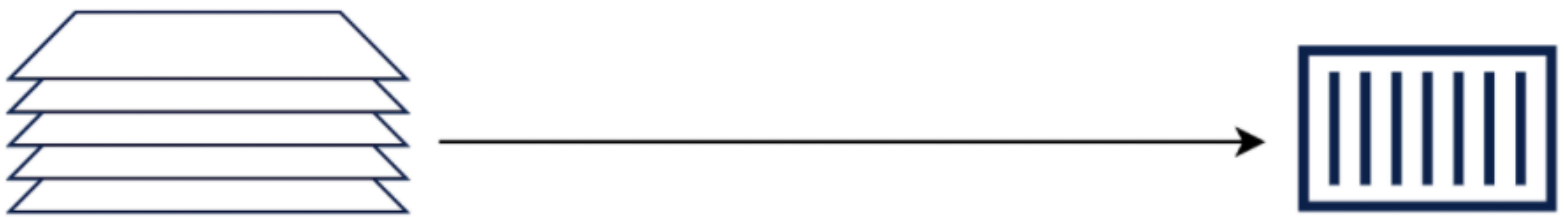
```
0 references
public class DockerImage
{
    0 references
    public string Name {get;set;}
    0 references
    public string Type {get;set;}
    0 references
    public DateTimeOffset CreatedOn {get;}
    0 references
    public int Version {get;set;}
}
```

```
var instance = new DockerImage();
```

exe

running exe

image vs container



build time construct

runtime construct

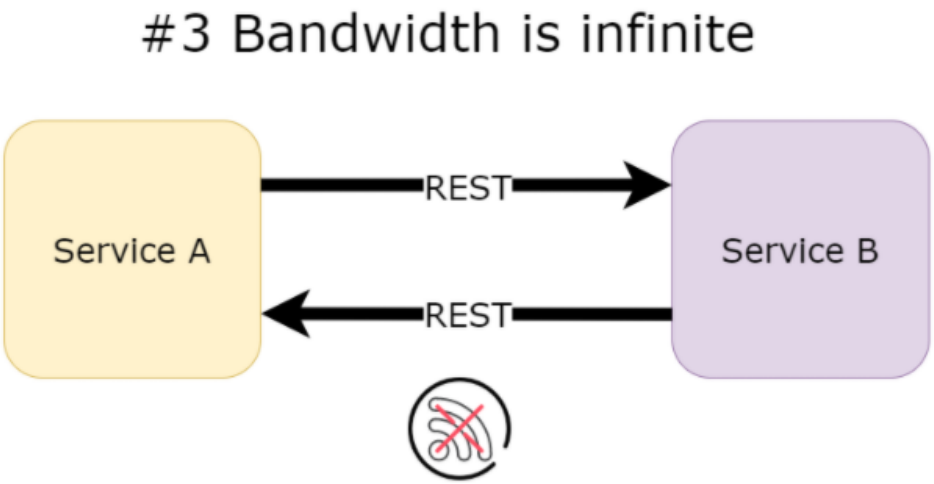
what image should contain?

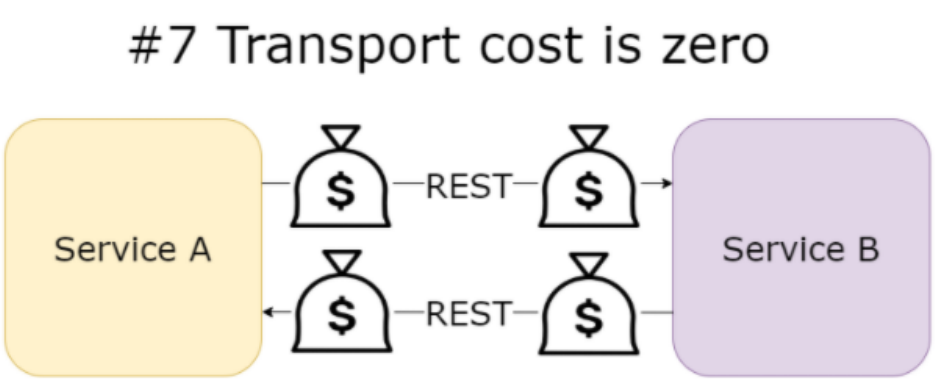
Exactly minimum needed to run and execute the application inside the container and/or to full fill image goal (hosting app, debugging, etc)

Nothing more, nothing less.

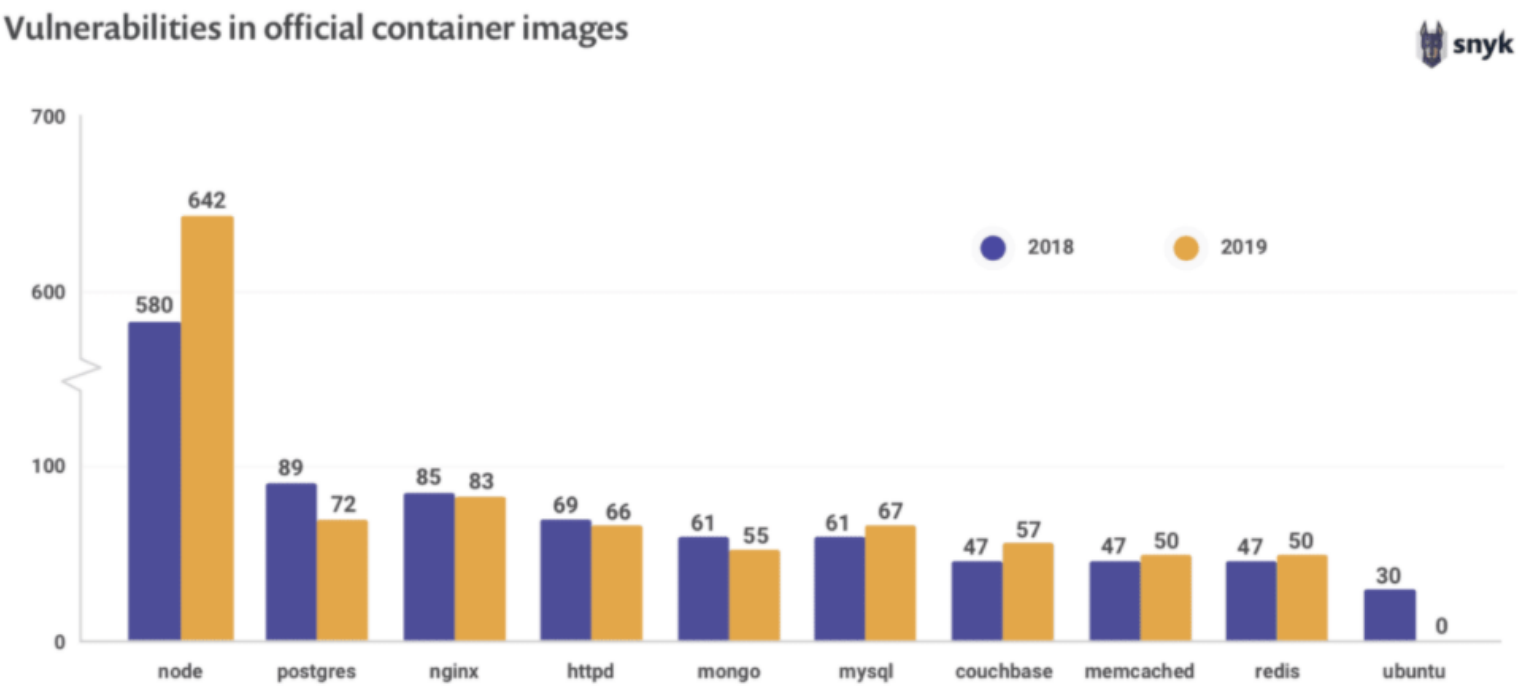
what image should contain?

Fallacies of Distributed Computing





what image should contain?



docker image ls

- List images (without intermediate)
- shortcut: *docker images*

```
docker image ls \
 [OPTIONS] \
 [REPOSITORY[:TAG]]

Options:
  -a, --all
      --digests
  -f, --filter filter
      --format string
      --no-trunc
  -q, --quiet
```

docker image pull

- Download image (with all intermediate images)
- shortcut: *docker pull*

```
docker image pull \
 [OPTIONS] \
 NAME[:TAG|@DIGEST]

Options:
  -a, --all-tags
      --disable-content-trust
      --platform string
  -q, --quiet
```

demo - list and pull

Instructions:

- Run: **docker image ls**
- Run: **docker image ls -q**
- Run: **docker image ls NAME**
- Run: **docker image ls --format "{{.Repository}}: {{.Size}}"**
- Run: **docker image pull python**

image naming

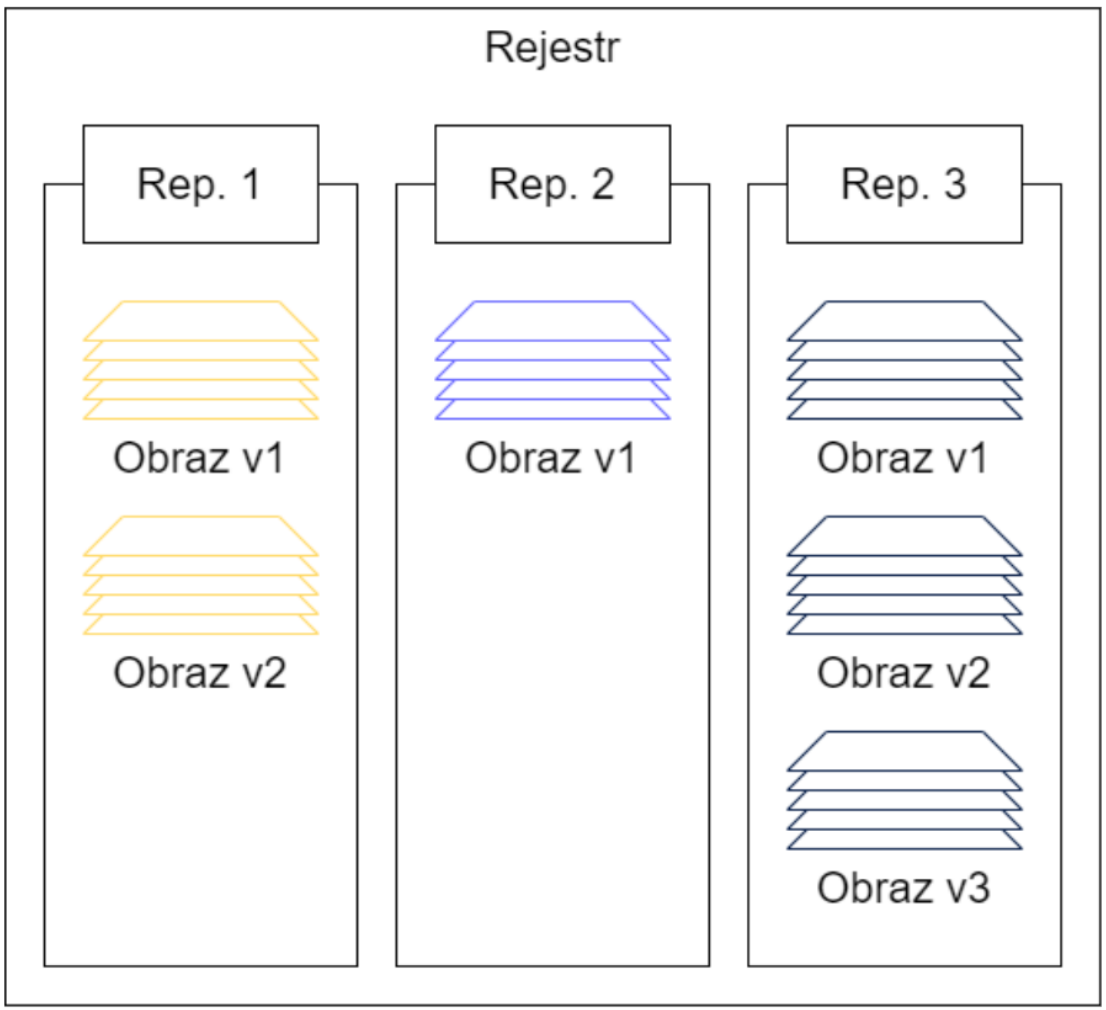


image naming schema

```
<namespace>/<repository>:<tag>
alpine:latest
python:latest
mongo:4
gutek/dumpster:v1
```

note on tag: latest

Theory:

tag latest == latest version

Practice:

tag latest != latest version

(latest stable, latest supported, latest LTS etc)

Two containers with latest tag might point to different image

note on tags

Good

- Group images per version:
  - tag 3 -> 3.1.1, 3.1
  - tag 3.1 -> 3.1.1
  - tag 3.1.1 -> version 3.1.1
- Human readable for of identifying images

Good

- Tags are mutable - you can push a new version of the image on the same tag
- You can't depend on it

so how to identify image?

Image ID

- Each image has sha256 hash identifier
- This is created based on local json file of the image
- this identifier is local to our computer
- **DONT USE to IDENTIFY IMAE**

Digest

- is cryptography hash of manifest file
- the manifest file is stored in the registry
- it uniquely identifies images across computers
- **USE TO IDENTIFY IMAGES**

image naming schema

with digest

```
<namespace>/<repository>@digest
python@sha256:077ca380466998790fa1e7d0d93ca6faa8bfd522e6ffa6d493183014f642e953
gutek/dumpster@sha256:76ddbed27cf8395b88dcf1027e0a8507d6638724f9b2827098e85489c4c4d050
```

3 types of registries

- official
- non-official
- externals

Official

- maintained by Docker
- tested, trusted, "secured"

alpine - https://hub.docker.com/_/alpine/

python - https://hub.docker.com/_/python/

etc

Non-Official

- maintained by user/company
- we should have limited trust

gutek/klubectl - https://hub.docker.com/r/gutek/kubectl/

gutek/dumpster - https://hub.docker.com/r/gutek/dumpster/

External

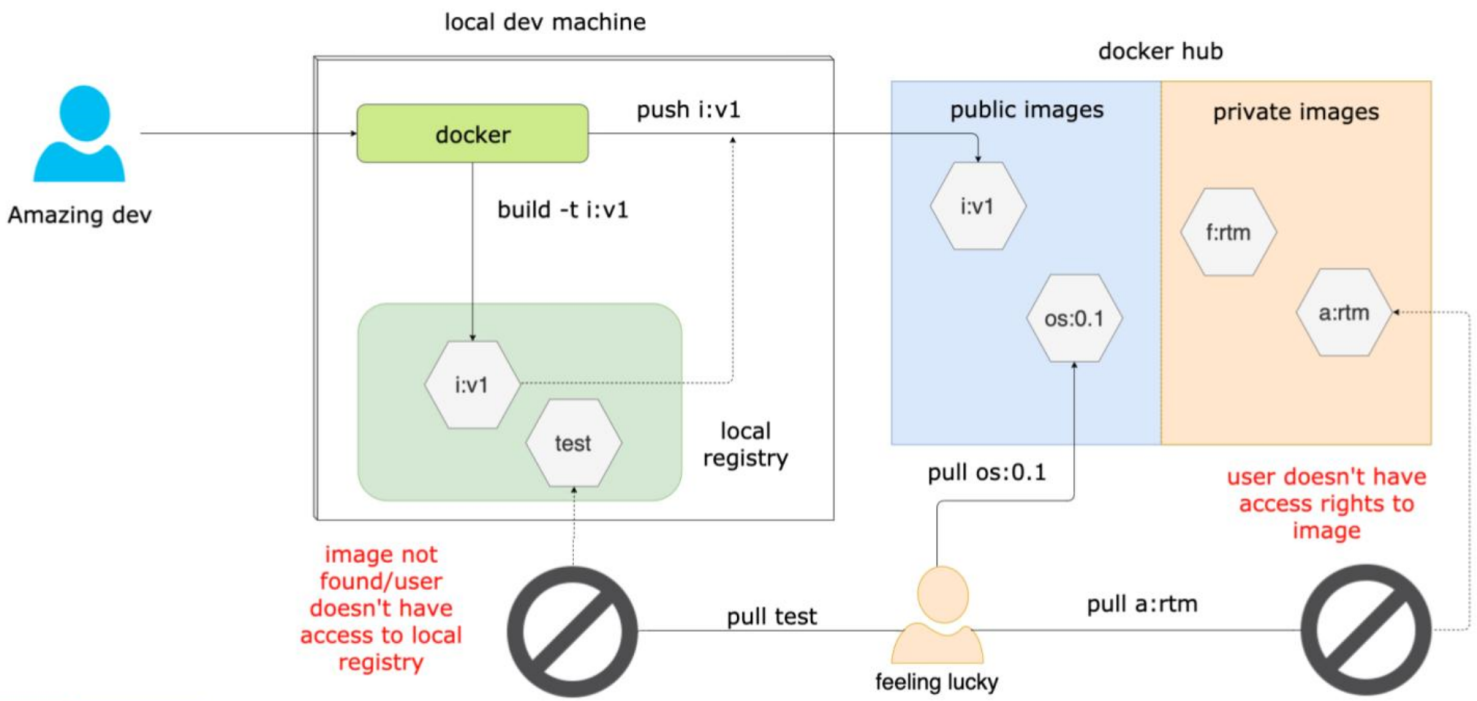
- Not hosted on docker hub
- can be used to secure our images and or to host images by orders

<registry url>/<namespace>/<repository>**:**<tag>

mcr.mircrosoft.com/dotnet/sdk:5.0

gcr.io/google-containers/busybox

Images and Repositories



docker image rm

- Removes images and all intermediate images if not used by containers/other images
- IMAGE is identified by: name with tag, id, digest

```
docker image rm \
 [OPTIONS] \
 IMAGE [IMAGE ...]

Options:
 -f, --force
     --no-prune
```

exercise - image rm

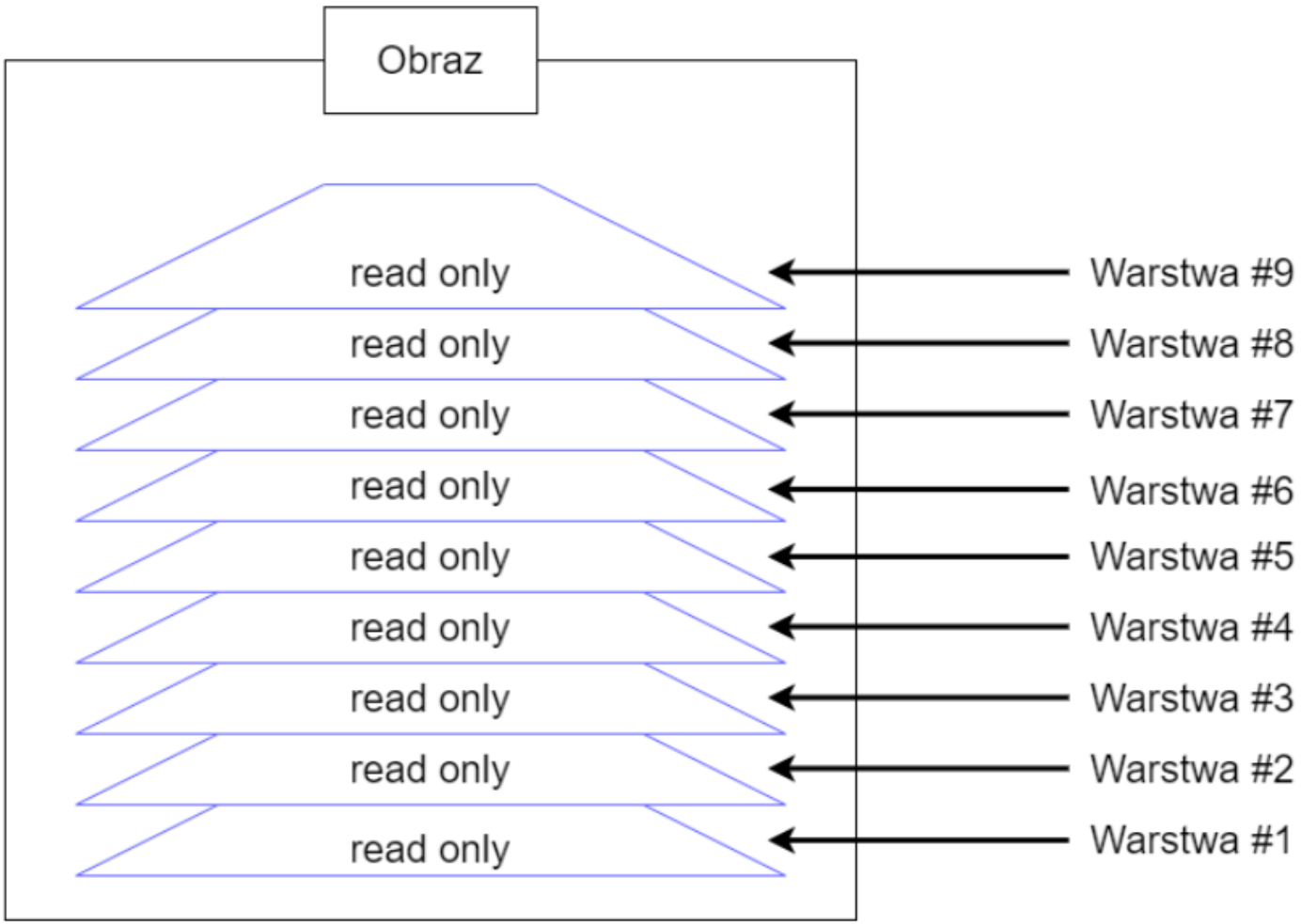
Instructions:

- Remove all images from your system
- Do it with one liner
- Hint: use two commands you've learn today, one for listing another one for removing

Time Cap: 10 min

Images and Layers

What is Layer?



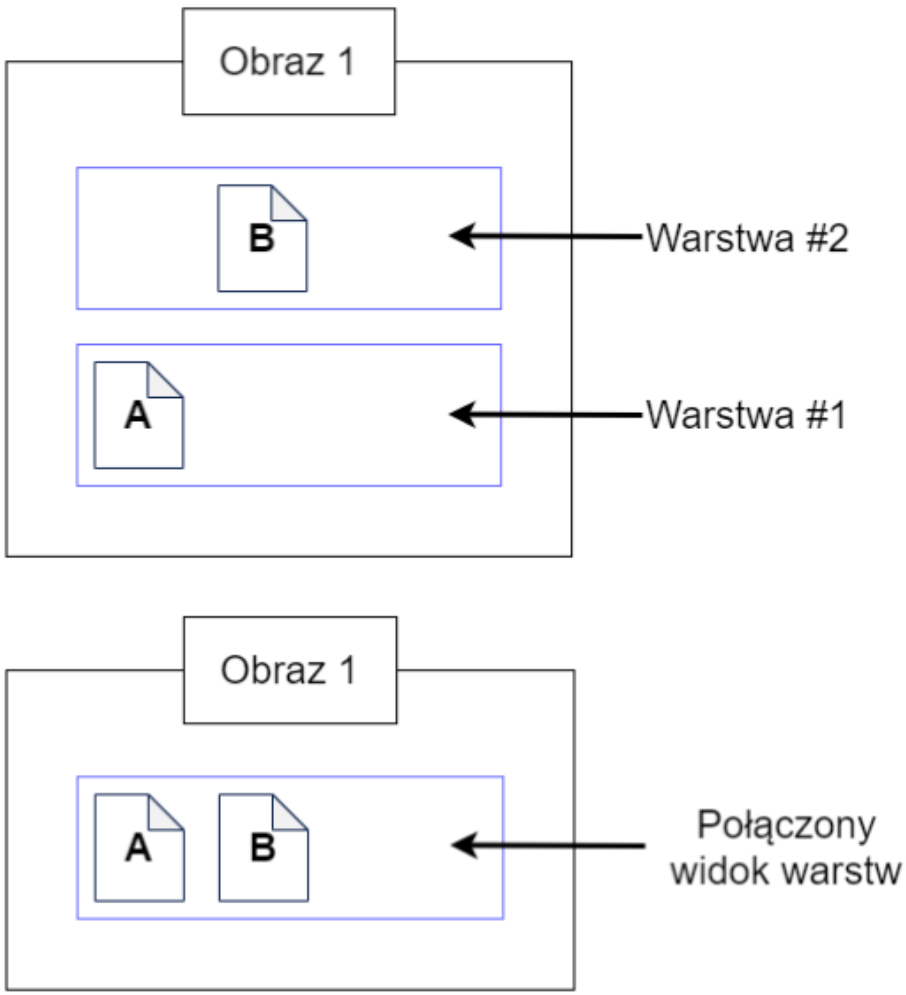
demo - image pull

Instructions:

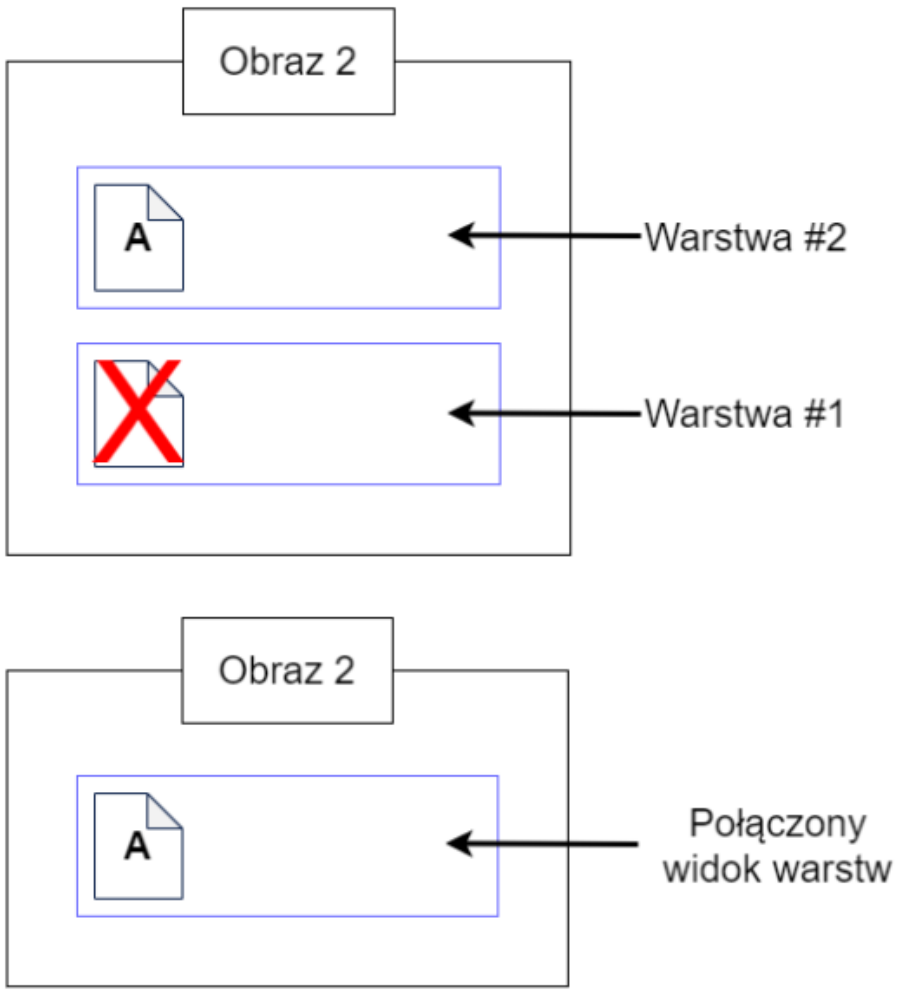
- Run: **docker image pull python:latest**

- Show layers

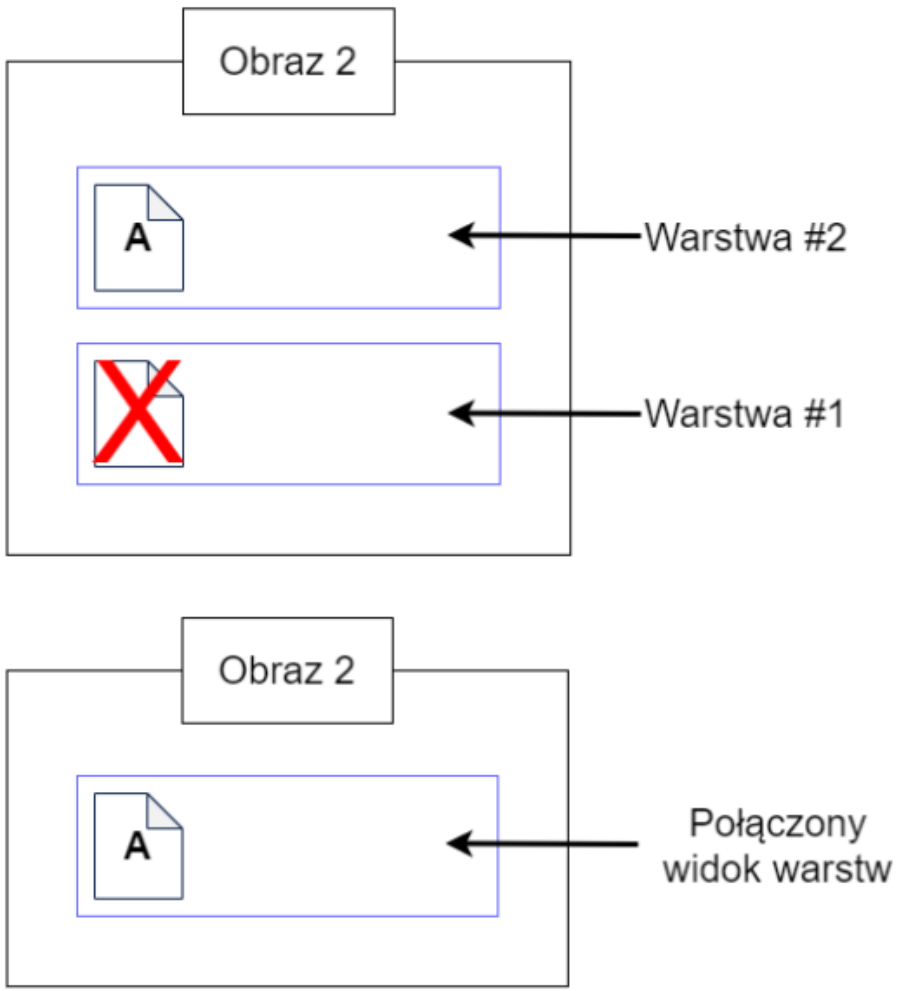
Layers and files



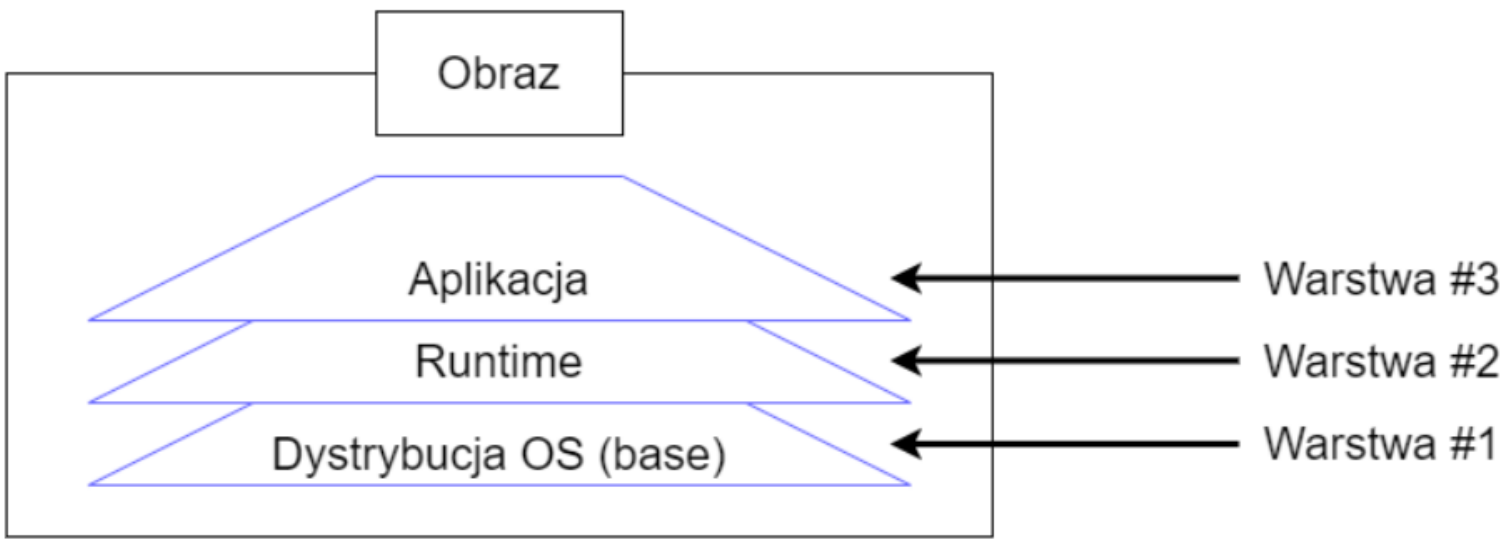
Layers and files



Layers and files



So what does the image contains?



docker image inspect

- Returns image configuration, detailed information

```
docker image inspect \
 [OPTIONS] \
 IMAGE [IMAGE ...]

Options:
 -f, --format string
```

docker image history

- Returns image build history

```
docker image history \
 [OPTIONS] \
 IMAGE

Options:
      --format string
  -H, --human
      --no-trunc
  -q, --quiet
```

docker manifest inspect

- Returns image manifest in the registry
- that includes information about images per platform

```
docker manifest inspect \
  [OPTIONS] \
  [MANIFEST_LIST] MANIFEST

Options:
      --insecure
  -v, --verbose
```

demo - inspect/history

Instructions:

- Run: **docker image inspect python**
- Run: **docker image history python**
- Run: **docker manifest inspect python**

But why do we need layers?

- Layers can be cached (faster download)
- Layers can be reused (smaller footprint)
- Layers are immutable
- Layers are not dependant on each other

Summary

- Image is for of packing up application
- Image is build from layers
- Layers are independent and are cached
- Each images has at least one tag and unique digest ID
- Images are downloaded from registries
- We've used commands to manage images or to see details about images on our computer

exercise - layers and size

Instructions:

- use following images gutek/pd:v1, gutek/pd:v2 and python:3
- answer questions
  - how many layers does gutek/pd:v1 and python:3 have?
  - what's the difference between gutek/pd:v1 and v2?
  - which layer from python:3 is the biggest?

Time Cap: 10 min

exercise - images larger than 10MB

Instructions:

- list all images larger than 10MB

Time Cap: 20 min

exercise - get size of all images

Instructions:

- get size of all images
- get true size of all images (not discussed)
- why sizes are different?

Time Cap: 15 min

Build

Build images

Two ways

Imperative

- using docker commands like run, commit, save and export
- can't be easily reproduced
- can't say what's included in the image

Declarative

- using Dockerfile
- we list commands that describe the end state
- we know whats included in the image
- we can easily replicate steps

docker image build

- Build an image from Dockerfile
- Alias: docker build

```
docker image build \
 [OPTIONS] \
 PATH | URL | -

Options:
      --... many options
  -f, --file string
      --no-cache
  -q, --quiet
      --secret stringArray
      --ssh stringArray
  -t, --tag list
```

Minimal command to build image

```
docker build .
```

Minimal "proper" command to build an image

```
docker build -t NAME:TAG .
```

demo - build image

Instructions:

- Run: **docker image build -t pd:latest .**
- Run: **docker run -d -p 5000:5000 pd**
- Run: **curl http://localhost:5000/**
- Run: **docker rm -f ID**

Build context

- Context of the build
- Contains everything that we've provided as a path to docker build PATH parameter
- We can't use files outside the build context
- We can exclude files from the build context using .dockerignore (like .gitignore)
- Use Dockerfile to build image

What is Dockefile ?

- State, instruction that specifies what we want and what we need in image
- Its a source file
- It should be treated as such - take care of it, be generous and clear about what it does and what it exposes
- It normally is names Dockerfile but it doesn't need to, if it does not use -f to specify name of Dockerfile during build

Dockerfile DSL

expressions and commands

Comments

- Add comment like you know TODO etc ;)

```
# comment in Dockerfile

FROM alpine:latest # comment on last line
# FROM alpine:latest this whole line will be commented
```

FROM

- Defines base image on which we are building our own
- With two exceptions: first line of Dockerfile

```
FROM <registry url>/<namespace>/<repository>:<tag>
FROM scratch
FROM python:latest
FROM python:3
```

LABEL

- Allows to provide metadata information about image
- Helps with maintenance and support

```
LABEL <key>="<value>"

LABEL version="1.1.1"
LABEL maintainer="kuba@poznajdocker.pl"
LABEL name="Hello Poznaj Docker"
LABEL description="Aplikacja przykładowa"
```

WORKDIR

- Sets the working directory in the image during the build process
- If directory does not exists, creates one

```
WORKDIR /test/code
WORKDIR debug
WORKDIR poznajdocker
```

demo - WORKDIR

Instructions:

- Show Dockerfile

- Run: **docker run --rm -it $(docker build -q .)**

RUN

- Executes command during the build process
- Each RUN creates new layer

```
RUN command
RUN ["/bin/sh", "-c", "command"]
RUN ping == RUN ["/bin/sh", "-c", "ping"]
RUN echo "Poznaj" \
  && echo " Docker"
```

RUN - Secrets

- We can pass secrets during build by ENV or ARG, but we know this is not secure
- There is a better way: use mount type secret
- This secret is only available during build time

```
# passing secret during build
# src - file that will be injected, id will be its name
docker build -t pd --progress=plain --secret id=pd,src=t.txt .

# extending RUN with --mout=type=secret
RUN --mount=type=secret,id=<id> <command> <param>
# return content of the t.txt file
RUN --mount=type=secret,id=pd cat /run/secrets/pd

# change secret path (dst)
RUN --mount=type=secret,id=<id>,dst=<dest_path> <command>
RUN --mount=type=secret,id=pd,dst=/tajne/haslo cat /tajne/haslo
```

demo - SECRETS

Instructions:

- Show Dockerfile

- Run: **echo TAJNE_HASLO_POZNAJ_DOCKER > tajne.txt**

- Run:

docker build --no-cache -t secret --secret id=pd,src=tajne.txt .

- Take a look at log

- Run: **docker image history secret**

- Run: **docker run --rm secret ls /run/secrets**

RUN - Cache

- Persist cache between builds
- Speeds up build process and limit bandwidth usage

```
# persist cache between builds for path /home/gradle/.gradle
RUN --mount=type=cache,target=<path> <command> <param>
RUN --mount=type=cache,target=/home/gradle/.gradle \
    gradle build --no-daemon
```

SHELL

- Sets shell under which RUN will execute

```
SHELL ["/bin/bash", "-c"]
RUN ping

SHELL ["/bin/zsh", "-c"]
RUN ping
```

ARG

- Pass dynamic arguments to build process
- Arguments are stored as plain text
- All ARG before FROM are not available after FROM

```
# does not contains value,
# we need to provide it during build
ARG version
ARG owner=team@poznajdocker.pl

FROM alpine:$version

# we need to re-declare it
ARG version
ARG owner # is set to team@poznajdocker.pl

LABEL version=$version

RUN echo $version

# USAGE
docker build -t pd --build-arg version=10 .
```

exercise - ARG

Instructions:

- Use ARG to define the version of the image
- Use ARG with a default value and override it
- Display version of an image with ECHO during the build
- Were you able to see the values of ARG?
- TIP: use one of the commands around inspecting, seeing things about images and layers, and build processes

Time Cap: 10 min

ENV

- Defines ENV variables available in container and during image build time
- Helps to define what ENV variables our application is using
- If we do not need ENV, use ARG

```
ENV <NAME>=<value>
ENV POZNAJ=doker

# env declared during RUN
RUN DOCKR=poznaj; echo $DOCKR > ./hello2
```

demo - ENV

Instructions:

- Show Dockerfile

- Run: **docker run --rm -it $(docker build -q .)**

EXPOSE

- Does "nothing"
- Updates manifest file of the image
- Allow option -P from docker run command to work
- It's used for documentation purpose

```
# default is tcp
EXPOSE <PORT>/[tcp|udp]

# exposes port 5000 on tcp
EXPOSE 5000

#exposes port 5001 on udp
EXPOSE 5001/udp
```

USER

- Sets user under which everything commands like RUN executes as well as our application
- Use value above 1000
- Make sure that user has rights to do whatever is needed

```
USER UID[:GID]
USER UID # domyślny GID:0, czyli root

USER 1001:1001
```

demo - USER

Instructions:

- Run: **docker run --rm alpine:latest id**
- Show Dockerfile
- Run: **docker run --rm $(docker build -q .) id**
- Uncomment user creation, and re-run build
- Run: **docker run --rm $(docker build -q .) id**

ADD and COPY

- Copy files from build context to the image
- COPY copies only files
- ADD allows extractions of tar files and adding files from URL (DANGEROUS!!!)
- COPY can use output from different FROM in Dockerfile
- ADD/COPY creates new layer

```
ADD/COPY [--chown=<user>:<group>] <src>... <dest>
ADD/COPY [--chown=<user>:<group>] ["<src>",... "<dest>"]

# using ADD
FROM scratch
ADD alpine-minirootfs-3.12.2-x86_64.tar.gz /
CMD ["/bin/sh"]
```

CMD & ENTRYPOINT

- Defines how our image can be used
- Are last commands in Dockerfile
- Are not required - container will finish working straight away
- Can be combined

```
# will create shell process under PID 1
CMD/ENTRYPOINT <command> <parameter> <parameter>

# will not create shell process (recommended)
CMD/ENTRYPOINT ["<command>", "<parameter>", "<parameter>"]
```

CMD & ENTRYPOINT

- CMD defines a default program that should be executed during container run. This program can be overridden with different. Should be used when we creating a generic image.
- ENTRYPOINT defines a program that should be executed and does not let to override it, but allows to pass parameters to it. Should be used when we creating image that is a tool.

demo - CMD & ENTRYPOINT

Instructions:

- Run: **docker run --rm -it $(docker build -q -f CmdDockerfile .)**
- Run: **docker run --rm -it $(docker build -q -f CmdDockerfile .) hostname**
- Run: **docker run --rm -it $(docker build -q -f EntrypointDockerfile .)**
- Run: **docker run --rm -it --entrypoint hostname $(docker build -q -f EntrypointDockerfile .)**

exercise - PING tool

Instructions:

- Create Dockerfile and build image that can be used as a ping tool

- This tool should by default ping localhost

- However, we would like to be able to override the ping host

- We do not want to be able to execute any other command, just ping

Time Cap: 10 min

Multi-Stage Dockerfile

Multi-stage

- Is constructed from many FROM
- Last FROM is the FROM of our image
- Previous FROM's are not stored/saved and after build are discarded
- Allow us to create an optimized version of the image containing only things we want to have
- To identify FROM, we can assign an alias to it

```
FROM x as build
FROM z as runtime
```

Image to Optimize

```
FROM python:latest

WORKDIR /code

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY src/ .

CMD [ "python", "./server.py" ]
```

STEP 1: adding alias

```
FROM python:latest AS builder

WORKDIR /code

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY src/ .

CMD [ "python", "./server.py" ]
```

STEP 2: clearing builder

```
# setting tag
FROM python:3 AS builder

# we do not need it
# WORKDIR /code

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY src/ .

CMD [ "python", "./server.py" ]
```

STEP 3: install requirements

```
FROM python:3 AS builder

COPY requirements.txt .

# adding --user so we can use it later
RUN pip install --user -r requirements.txt

COPY src/ .

CMD [ "python", "./server.py" ]
```

STEP 4: introduce runtime

```
FROM python:3 AS builder

COPY requirements.txt .
RUN pip install --user -r requirements.txt

# runtime FROM
FROM python:3-slim AS runtime

COPY src/ .

CMD [ "python", "./server.py" ]
```

STEP 5: copy code

```
FROM python:3 AS builder

COPY requirements.txt .
RUN pip install --user -r requirements.txt

# runtime FROM
FROM python:3-slim AS runtime

WORKDIR /code

COPY --from=builder /root/.local/ /root/.local
COPY src/ .

CMD [ "python", "./server.py" ]
```

STEP 6: finalize image

```
FROM python:3 AS builder

COPY requirements.txt .
RUN pip install --user -r requirements.txt

FROM python:3-slim AS runtime

WORKDIR /code
COPY --from=builder /root/.local/ /root/.local
COPY src/ .
ENV PATH=/root/.local:$PATH

CMD [ "python", "./server.py" ]
```

demo - Multi-Stage vs Normal

Instructions:

- Run: **docker build -t m:s -f SDockerfile .**
- Run: **docker build -t m:m -f MDockerfile .**
- Run: **docker images m**
- Take a look at image size column

exercise - Create Multi-Stage for dotnet app

Instructions

To build execute in code folder:

dotnet publish -c Release

docker build -t webapp .

docker run --rm -it -p 5000:80 webapp

you can see Dockerfile, its not a perfect one.

you can also see BisDockerfile that does build dotnet app in a container

Try to improve this using a multistage build.

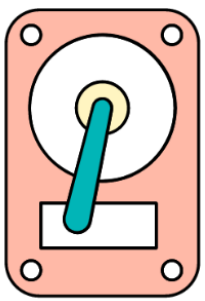
Tip: use mcr.microsoft.com/dotnet/aspnet:5.0 for final image

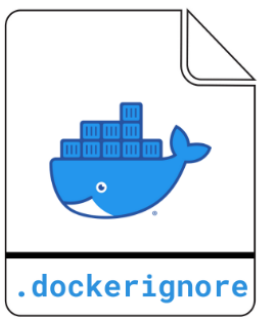
Tip: remember about COPY --from=FROM_ALIAS

Time Cap: 15 min

Best Practices

#01: Build Context





#02: Order of instructions

Bad

```
1  FROM node:latest
2
3  WORKDIR /app
4
5  COPY . .
6
7  RUN npm config set registry \
8      http://registry.npmjs.org/
9  RUN npm install
10
11
12
13 EXPOSE 8080
14 CMD [ "node", "index.js" ]
```

Good

```
1  FROM node:latest
2
3  WORKDIR /app
4
5  COPY package*.json ./
6
7  RUN npm config set registry \
8      http://registry.npmjs.org/
9  RUN npm install
10
11 COPY . .
12
13 EXPOSE 8080
14 CMD [ "node", "index.js" ]
```

#03: Copy Specific files

Bad

```
1  FROM node:latest
2
3  WORKDIR /app
4
5  COPY package*.json ./
6
7  RUN npm config set registry \
8      http://registry.npmjs.org/
9  RUN npm install
10
11 COPY . .
12
13 EXPOSE 8080
14 CMD [ "node", "index.js" ]
```

Good

```
1  FROM node:latest
2
3  WORKDIR /app
4
5  COPY package*.json ./
6
7  RUN npm config set registry \
8      http://registry.npmjs.org/
9  RUN npm install
10
11 COPY index.js ./
12
13 EXPOSE 8080
14 CMD [ "node", "index.js" ]
```

#04: Combine actions

Bad

```
1  FROM node:latest
2
3  WORKDIR /app
4
5  COPY package*.json ./
6
7  RUN npm config set registry \
8      http://registry.npmjs.org/
9  RUN npm install
10
11 COPY index.js ./
12
13 EXPOSE 8080
14 CMD [ "node", "index.js" ]
```

Good

```
1  FROM node:latest
2
3  WORKDIR /app
4
5  COPY package*.json ./
6
7  RUN npm config set registry \
8      http://registry.npmjs.org/ \
9      && npm install
10
11 COPY index.js ./
12
13 EXPOSE 8080
14 CMD [ "node", "index.js" ]
```

#05: Sort commands

Bad

```
1  FROM node:latest
2
3  WORKDIR /app
4
5  COPY package*.json ./
6
7  RUN npm config set registry \
8      http://registry.npmjs.org/ \
9      && npm install \
10     socket.io \
11     express
12
13 COPY index.js ./
14
15 EXPOSE 8080
16 CMD [ "node", "index.js" ]
```

Good

```
1  FROM node:latest
2
3  WORKDIR /app
4
5  COPY package*.json ./
6
7  RUN npm config set registry \
8      http://registry.npmjs.org/ \
9      && npm install \
10     express \
11     socket.io
12
13 COPY index.js ./
14
15 EXPOSE 8080
16 CMD [ "node", "index.js" ]
```

#06: Remove unused depedencies

Bad

```
1  FROM node:latest
2
3  WORKDIR /app
4
5  COPY package*.json ./
6
7  RUN npm config set registry \
8      http://registry.npmjs.org/ \
9      && npm install
10
11 COPY index.js ./
12
13 EXPOSE 8080
14 CMD [ "node", "index.js" ]
```

Good

```
1  FROM node:latest
2
3  WORKDIR /app
4
5  COPY package*.json ./
6
7  RUN npm config set registry \
8      http://registry.npmjs.org/ \
9      && npm install --no-optional
10
11 COPY index.js ./
12
13 EXPOSE 8080
14 CMD [ "node", "index.js" ]
```

#07: Clear cache

Bad

```
1  FROM node:latest
2
3  WORKDIR /app
4
5  COPY package*.json ./
6
7  RUN npm config set registry \
8      http://registry.npmjs.org/ \
9      && npm install --no-optional
10
11
12 COPY index.js ./
13
14 EXPOSE 8080
15 CMD [ "node", "index.js" ]
```

Good

```
1  FROM node:latest
2
3  WORKDIR /app
4
5  COPY package*.json ./
6
7  RUN npm config set registry \
8      http://registry.npmjs.org/ \
9      && npm install --no-optional \
10     && npm cache clear
11
12 COPY index.js ./
13
14 EXPOSE 8080
15 CMD [ "node", "index.js" ]
```

#08: Choose base image

Bad

```
1  FROM node:latest
2
3  WORKDIR /app
4
5  COPY package*.json ./
6
7  RUN npm config set registry \
8      http://registry.npmjs.org/ \
9      && npm install --no-optional \
10     && npm cache clear
11
12 COPY index.js ./
13
14 EXPOSE 8080
15 CMD [ "node", "index.js" ]
```

Good

```
1  FROM node:12
2
3  WORKDIR /app
4
5  COPY package*.json ./
6
7  RUN npm config set registry \
8      http://registry.npmjs.org/ \
9      && npm install --no-optional \
10     && npm cache clear
11
12 COPY index.js ./
13
14 EXPOSE 8080
15 CMD [ "node", "index.js" ]
```

#09: Take smallest image

```
1  REPOSITORY          TAG                 SIZE
2  node                12-slim             153MB
3  node                12                  908MB
4  node                12-alpine           80.9MB
```

#10: Split application

- The application should be responsible for ONE thing only
- This allows horizontal scaling depending on responsibility (we do not need PESEL verification scaled to 100 instances because we have our Invoicing system scaled to 100 instances)
- We can re-use containers when needed (helpful with sidecar pattern)

#11: Use Multistage Build

```
1  FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS build
2  WORKDIR /app
3
4  COPY *.sln .
5  COPY aspnetapp/*.csproj ./aspnetapp/
6  RUN dotnet restore
7
8  COPY aspnetapp/. ./aspnetapp/
9  WORKDIR /app/aspnetapp
10 RUN dotnet publish -c Release -o out
11
12 FROM mcr.microsoft.com/dotnet/core/aspnet:3.1 AS runtime
13 WORKDIR /app
14 COPY --from=build /app/aspnetapp/out ./
15 ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

#12: Change default user

```
1  FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS build
2  WORKDIR /app
3
4  COPY *.sln .
5  COPY aspnetapp/*.csproj ./aspnetapp/
6  RUN dotnet restore
7
8  COPY aspnetapp/. ./aspnetapp/
9  WORKDIR /app/aspnetapp
10 RUN dotnet publish -c Release -o out
11
12 FROM mcr.microsoft.com/dotnet/core/aspnet:3.1 AS runtime
13 WORKDIR /app
14
15 USER 1001:1001
16
17 COPY --from=build /app/aspnetapp/out ./
18 ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

#134: Use own base image

```
1  FROM company/dotnet-sdk:3.1 AS build
2  WORKDIR /app
3
4  COPY *.sln .
5  COPY aspnetapp/*.csproj ./aspnetapp/
6  RUN dotnet restore
7
8  COPY aspnetapp/. ./aspnetapp/
9  WORKDIR /app/aspnetapp
10 RUN dotnet publish -c Release -o out
11
12 FROM company/dotnet-rte:3.1 AS runtime
13 WORKDIR /app
14
15 USER 1001:1001
16
17 COPY --from=build /app/aspnetapp/out ./
18 ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

#145: Use distroless base image

- Distroless was create by Google
- Distroless is an image that does not have linux distro as base
- It's more secure as it does not have problems of security holes in Linux distros
- More info: https://github.com/GoogleContainerTools/distroless

exercise - All i One

Instructions:

- Improve on Multi-stage build from previous exercise

Time Cap: 15 min

Docker compose & networks

Docker compose

Help commands

- To way to access compose
- Just use help to discover options

```
docker-compose help
```

```
docker compose help
```

docker compose

- Allows to run multiple containers
- Build some of them
- Manage dependencies

```yaml
version: "3.9"
services:
  db:
    image: postgres
    environment:
      - POSTGRES_DB=postgres
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=pass
  web:
    build: .
    command: python manage.py
    volumes:
      - .:/code
    ports:
      - "8000:8000"
    depends_on:
      - db
```

Parts

- Version

Compose file format	Docker Engine release
Compose specification	19.03.0+
3.8	19.03.0+
3.7	18.06.0+
3.6	18.02.0+
3.5	17.12.0+
3.4	17.09.0+
3.3	17.06.0+
3.2	17.04.0+
3.1	1.13.1+
3.0	1.13.0+
2.4	17.12.0+
2.3	17.06.0+

```
version: "3.9"
services:
  wordpress:
    image: wordpress
    ports:
      - "8080:80"
    networks:
      - my_net
volumes:
  db-data:
networks:
  my_net:
```

Parts

- Services

```
version: "3.9"
services:
  wordpress:
    image: wordpress
    ports:
      - "8080:80"
    networks:
      - my_net
volumes:
  db-data:
networks:
  my_net:
```

Parts

- Volumes

```
version: "3.9"
services:
  wordpress:
    image: wordpress
    ports:
      - "8080:80"
    networks:
      - my_net
volumes:
  db-data:
networks:
  my_net:
```

Parts

- Networks

```
version: "3.9"
services:
  wordpress:
    image: wordpress
    ports:
      - "8080:80"
    networks:
      - my_net
volumes:
  db-data:
networks:
  my_net:
```

Networks

Network in docker

- Network drivers
- IPtables

User-defined bridge

- default
- Used in container to container communication
- Every container created without detailed spec uses user-definied bridge

Host networks

- Uses host operating system
- **Doesn't** use namespace isolation for network

Overlay networks

- Network between multiple machine with containers
- Used in Docker Swarm (☠)
- Cloud and product vendors use their own solutions

MacVLAN networks

- Works only on Linux Host. No Docker Dekstop support
- Docker routes traffic to your container using its MAC address
- Don't try this at home 😅

demo - docker network

Instructions:

- Run cmd: **docker network ls**
- Run cmd: **docker network inspect bridge**

Practical guide

walkthrough - docker compose

part 1

- App:
  - simple dotnet v5
  - SQL Server 2017 in Docker
- Repo: https://github.com/ptrstpp950/v5-dockerComposeAspNetCore

docker-compose up

- Builds, (re)creates, starts, and attaches to containers for a service
- Unless they are already running, this command also starts any linked services.

```
docker-compose up \
  [OPTIONS] \
  [--scale SERVICE=NUM...]
  [--] [SERVICE...]

Options:
-d, --detach
    --force-recreate
    --no-recreate
    --build
  [much more]
```

demo - first run

part 2

- Repo: https://github.com/ptrstpp950/v5-dockerComposeAspNetCore
- First step is to run: **docker-compose up --build**
- This will take a while (downloading images + build +...)
- Why we need **--build**?
- Check http://127.0.0.1:8090/
- Add some books

exercise - where is database?

part 3

- Press CTRL+C to stop app
- Run it again do you see your books?
- Try to delete database 😅

walkthrough - persistent DB

part 4

- Add volume pointing to:

  ${APPDATA}/MyServiceDB/mssql:/var/opt/mssql/data
- Rerun app + add books + stop it
- Remove container or use docker-compose down
- Rerun app again

```
ms-sql-server:
    image: mcr.microsoft.com/mssql/server:2017-latest-ubuntu
    environment:
      ACCEPT_EULA: "Y"
      SA_PASSWORD: "Youtube2021"
      MSSQL_PID: Express
    ports:
      - "1433:1433"
    volumes:
      - ${APPDATA}/MyServiceDB/mssql:/var/opt/mssql/data
```

docker-compose down

- Stops containers and removes containers, networks, volumes, and images created by `up`.
- By default, the only things removed are:

  - Containers for services
  - Networks defined in the `networks` section
  - The default network, if one is used

  Networks and volumes defined as `external` are never removed.

```
docker-compose down \
  [OPTIONS] \

Options:
      --rmi type
   #type='all'or'local'
 -v, --volumes
      --remove-orphans
 -t, --timeout TIMEOUT
```

walkthrough - access network

part 5

- Run **docker network ls** to get network names
- Run **docker network inspect NAME** to get IP address of web application
- Run any container with shell: **docker run -it docker/getting-started /bin/sh**
- Try to access web app with IP address or name on port 80 (check it in docker compose file)
- Run it again with option **--network NAME**
- Do you see difference? 😅

walkthrough - shut it down :)

part 6

- Run **docker-compose down** to clean up

Summary

- We know how to run multiple containers locally
- We know how to use volumes and save data
- We know docker network types
- We know how to access docker network
- We know how to clean up :)

CI/CD & tools

Creating pipeline

Good pipeline - problems

- How to build a good pipeline?
- What should be included in it?
- What is necessary?
- What is optional?

Build in Docker or not?

Advantages

- We need only Docker on build agent

Disadvantages

- Docker cache can be a problem (like any other)
- There can be a "security issue" - building container inside a container

Static check

- Use linter to check syntax (for example hadolint)
- Check security requirements (for example conftest with Open Policy Agent)
- What should we also consider:
  - base images
  - force using labels
  - link to commit and/or build

Build

- The easiest part 😅
- When run tests:
  - during the build?
  - after the build?
- The cache is very important (we will talk about it later)
- The tags are important too (do you remember what should be in a good tag?)

Security check

- CVE - Common Vulnerabilities and Exposures
- Almost all images have some CVE
- Nice tools:
  - trivy + output in JUnit format
  - docker scan - build in, but requires parsing result

docker scan

- Paid option
- CLI uses external SNYK database

```
docker scan [OPTIONS] IMG

Options:

      --accept-license
      --dependency-tree
      --exclude-base
-f,   --file string
      --group-issues
      --json
      --login
      --reject-license
      --severity string
      --token string
      --version
```

Push + registry

- We talked about them (in Azure use ACR)
- What should you remember:
  - "Near" - registry should be near target infrastructure
  - Other options - like continuous CVE check
  - Feature from the future - auto-cleanup 😅

Advanced tips & tricks

- Remember about cache but do not expect something spectacular
- .dockerinnore changes a lot
- Use BuildKit (it is faster)
- Consider company images
- Make builds repeatable

Cache

Agents

- Hosted
- Self-hosted
- "Mixed"

Problems

- Build queue
- Checkout time
- Cache in build phase:
  - Dependencies (NuGet, npm, maven, pip, ...)
  - Docker cache
  - Cache ...
- Costs
- Other

Hosted agent

- Always "clean" and updated
- Always ready (until we have $$$)
- Usually fast
- No simple cache included

Self-Hosted agent

- With power comes great responsibility 😎
- Cache by default in:
  - disk (for package managers)
  - Docker
- "Clean time"
- More $$$

“Mixed” agent

- Hosted agent + cache
- Advantages of both
- Hard to create

Docker agent

- Easy to use / copy / create new instances
- Introduce security issue
  - using bind-mount with Docker socket
  - pipeline inside can run root scripts on host

Multistage cache

Cache - for what?

- Source code
- External dependencies
- Docker layers

How to share dependencies

- Use local artifact repository
- Use volume mount
- Share volume between agents

Multistage problems

- We create more than one image during build
- Usually we push only final one
- Final image is smaller, but ...

docker target

- Build "selected" layer
- Than we can push & pull
- All next layers will be skipped

```
docker build \
  --target [STAGE]\
  [OTHER OPTIONS]
```

demo - docker target

- Open Dockerfile from docker-compose demo (repo: https://github.com/ptrstpp950/v5-dockerComposeAspNetCore)
- Rename build-env to restore-env
- Before COPY . ./ add: FROM restore-env AS build-env
- Run:
  - docker build --target restore-env .
  - docker build --target build-env .
  - docker build .
- Do you seedifferences?

cache-from

- More than one cache-from is ok :)
- A way to use cache

```
docker build \
  --cache-from=[IMAGE] \
  [OPTIONS]
```

Azure DevOps example

(another idea 😎)

```yaml
pool:
  vmImage: 'Ubuntu-16.04'
steps:
  - task: Cache@2
    displayName: Cache task
    inputs:
      key: 'docker | "$(Agent.OS)" | cache'
      path: $(Pipeline.Workspace)/docker
      cacheHitVar: CACHE_RESTORED  #Variable to set to 'true' when the cache is restored

  - script: |
      docker load -i $(Pipeline.Workspace)/docker/cache.tar
    displayName: Docker restore
    condition: and(not(canceled()), eq(variables.CACHE_RESTORED, 'true'))

  - script: |
      mkdir -p $(Pipeline.Workspace)/docker
      docker save -o $(Pipeline.Workspace)/docker/cache.tar cache
    displayName: Docker save
    condition: and(not(canceled()), or(failed(), ne(variables.CACHE_RESTORED, 'true')))
```

Source: Pipeline caching - Azure Pipelines | Microsoft Docs

buildx - cache-to

- Something new (not supported everywhere)
- Supports: registry, disk and inline

```
# registry
docker buildx build \
  --cache-to=user/app:cache .
docker buildx build \
  --cache-to=type=registry,ref=user/app .

# disk
docker buildx build \
  --cache-to=type=local,dest=path/to/cache .

# inline
docker buildx build --cache-to=type=inline .
```

demo -

- It probably won't work
- Run:

```
docker buildx build \
  --cache-to=type=local,dest=path/to/cache .
```

Secrets

Discussion

- Where to store secrets?
- How about accessing them?

The best pipeline ever

The best pipeline

- Checkout
- Static checks (linter + security rules)
- Optionally: get/mount cache
- Build (remember about good tags)
- Run security check + save result
- Push to registry
- Optionally: store data to cache

Piotr Stapp

- twitter: @ptrstpp950
- stapp.space
- piotr.stapp@gmail.com

Jakub Gutkowski

- twitter: @gutek
- GutkowskiJakub
- kuba@gutek.pl