Introduction to containerization with Docker & Kubernetes

Zdravko Chiflishki & Iliya Mihov

Agenda

1. Introduction to Containers
2. Docker Overview - Images/Volumes/Networking
3. Installation & Configuration
4. Docker Commands
5. Lessons Learned
6. Orchestration with Kubernetes
7. Q&A

Why, What and How

Why Containers ?

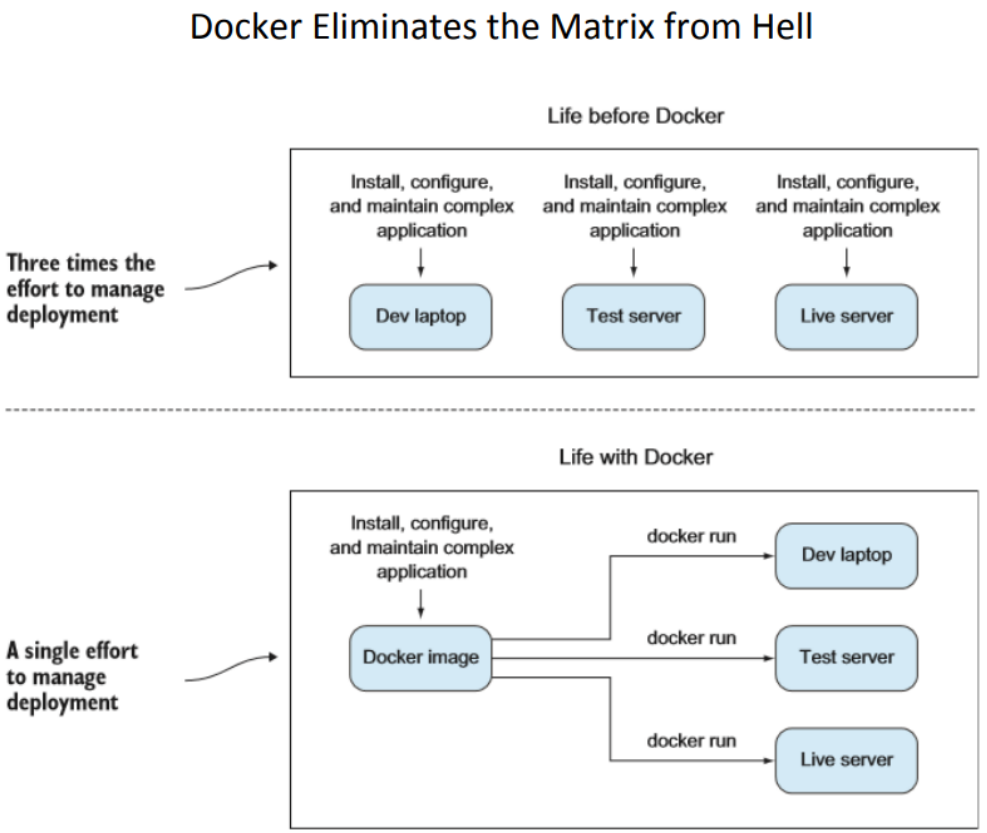
- Software industry has changed

- Before:
  - Monolithic applications
  - Long development cycles
  - Single environment
  - Slowly Scaling Up
- Now:
  - De-Coupled services
  - Fast and iterative improvements
  - Multiple environments
  - Quickly scaling out

- Deployment becomes Complex

- Many different stacks:
  - Languages
  - Frameworks
  - Databases
  - Toolchains
- Many different targets:
  - Individual development environments
  - Pre-production, QA, Staging
  - Production: On-premise, Cloud, Hybrid

What issue does it solve – The Deployment Matrix Problem

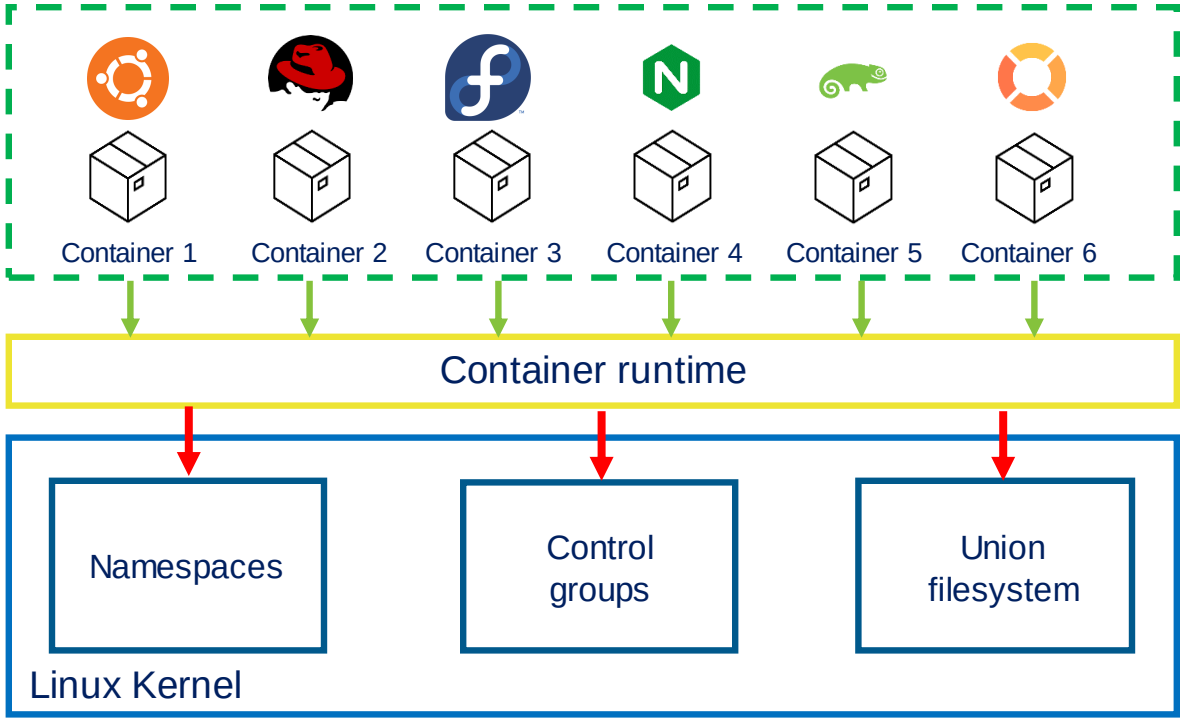


What are Containers?

Let's search in… Google

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.

- Containers are an abstraction over different Linux technologies
- The 3 key primitives that creates what we knows as containers are:
  - Namespaces
  - Control groups (cgroups)
  - Union filesystem



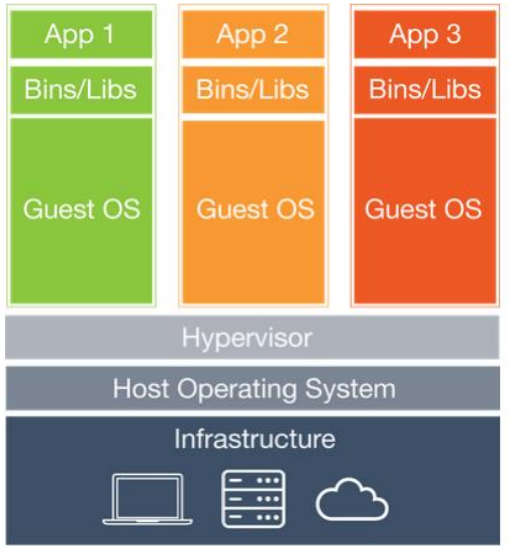
BOSCH

How were containers Introduced ?

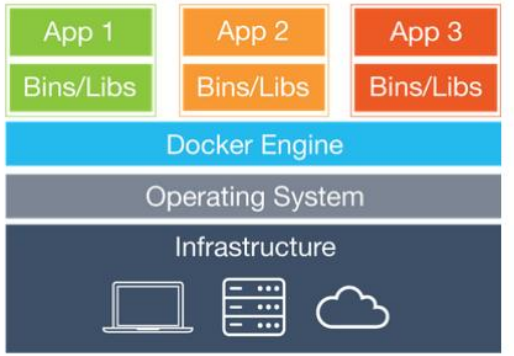
Virtual machines vs Containers

Criteria	VM's	Containers
OS support	Occupies a lot of memory space	Docker containers occupies less space
Boot-up time	Long boot-up time	Short boot-up time
Performance	Running multiple virtual machines leads to unstable performance	Containers have a better performance as they are hosted in a single Docker engine
Scaling	Difficult to scale up	Easy to scale up
Efficiency	Low efficiency	High efficiency
Portability	Compatibility issues while porting across different platforms	Easy portable across different platforms
Space allocation	Data volumes cannot be shared	Data volumes can be shared and reuse among multiple containers

VS



Virtual machines

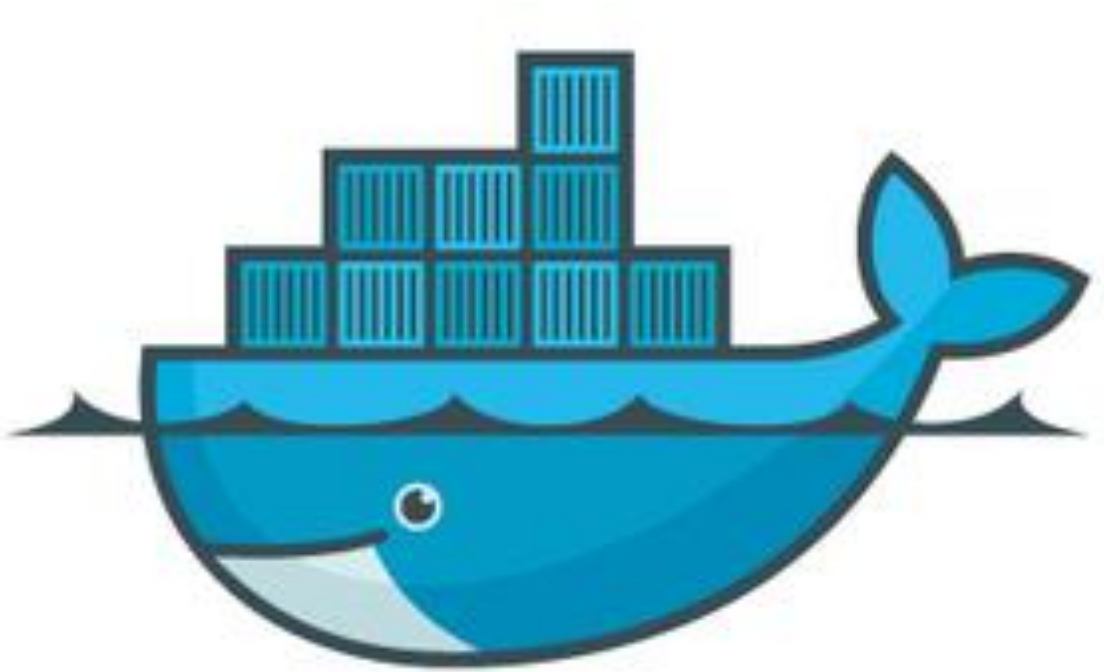


Containers

Docker Overview

Docker History – How it began (1/2)

- Software debuted to the public in Santa Clara at PyCon in 2013.
- Written in: Go
- Products maintained by Docker, Inc.
  - Docker Engine
  - Docker Engine Enterprise
  - Docker Hub
  - Docker Desktop
- Headquarters: San Francisco
- Original Author: Solomon Hykes



Docker History - The Evolution of Docker (2/2)

Docker has fundamentally changed how software is developed and deployed. Its journey has been one of rapid innovation and widespread adoption, setting new standards for efficiency and portability in the software industry.

2013: Project Launch

Docker was unveiled as an open-source project, quickly gaining traction for its ability to package applications into isolated, portable containers.

2014-2016: Ecosystem Growth

Rapid expansion saw the introduction of Docker Hub, Docker Compose, and Docker Swarm, fostering a vibrant ecosystem and community.

2017: Kubernetes Integration

Docker embraced industry standardisation by integrating with Kubernetes, solidifying its role as a cornerstone of cloud-native development.

Present: Continued Innovation

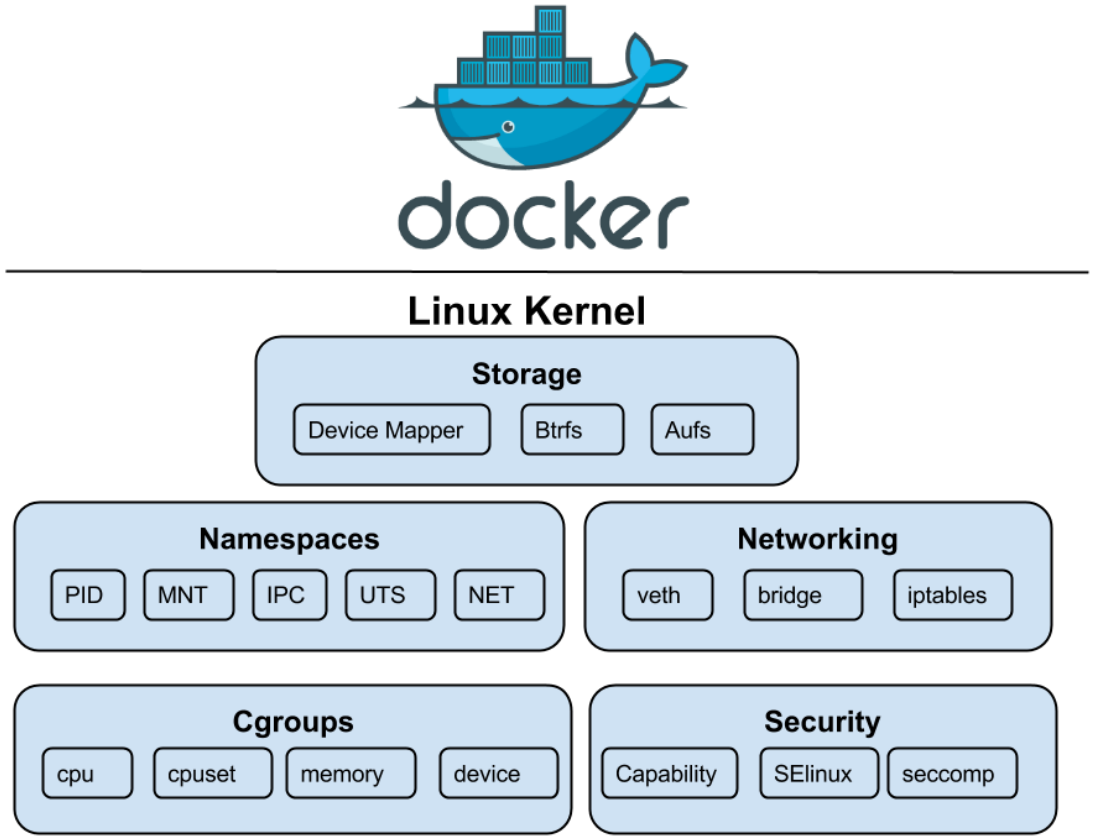
Today, Docker continues to evolve, focusing on enterprise solutions, enhanced security, and seamless integration across diverse development workflows.

Docker Engine

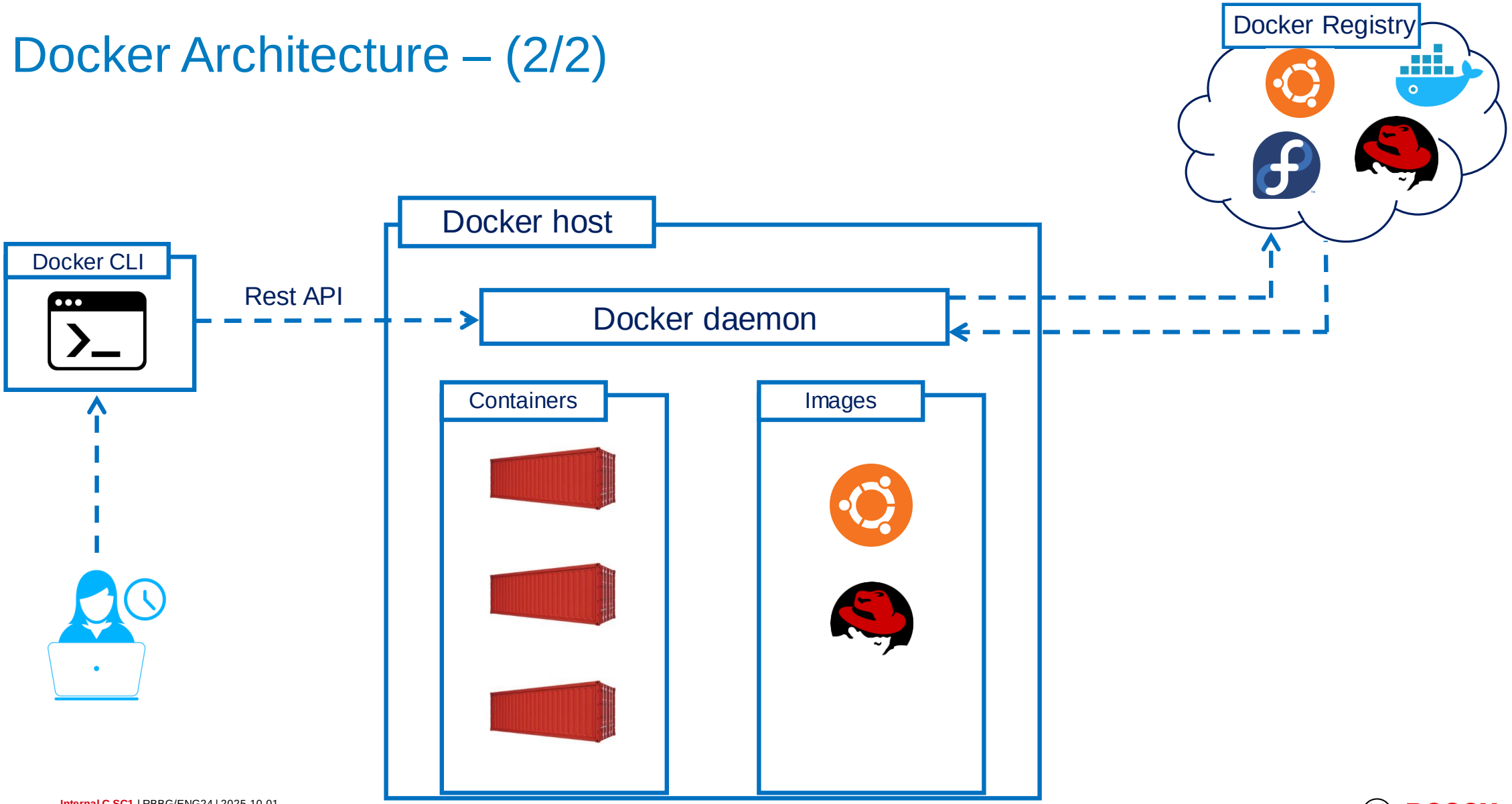
- Open-source Platform
- Enable Separation
- Infrastructure Manager
- Ship Test Deploy
- Reduce Delay
- Isolated Environment
- Lightweight
- Share Container

The Most Reliable Container Platform on the Market with the largest Open Source Community

Docker Architecture – (1/2)



Docker Architecture – (2/2)

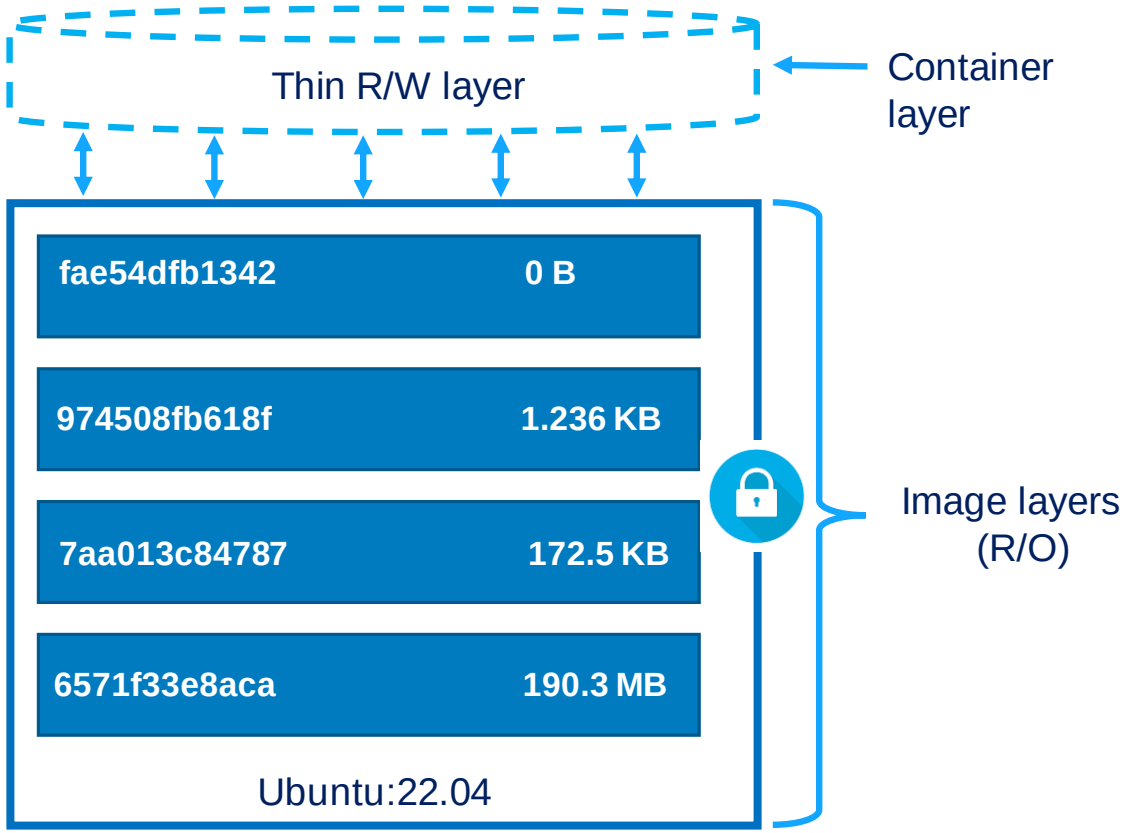


BOSCH

Docker Images

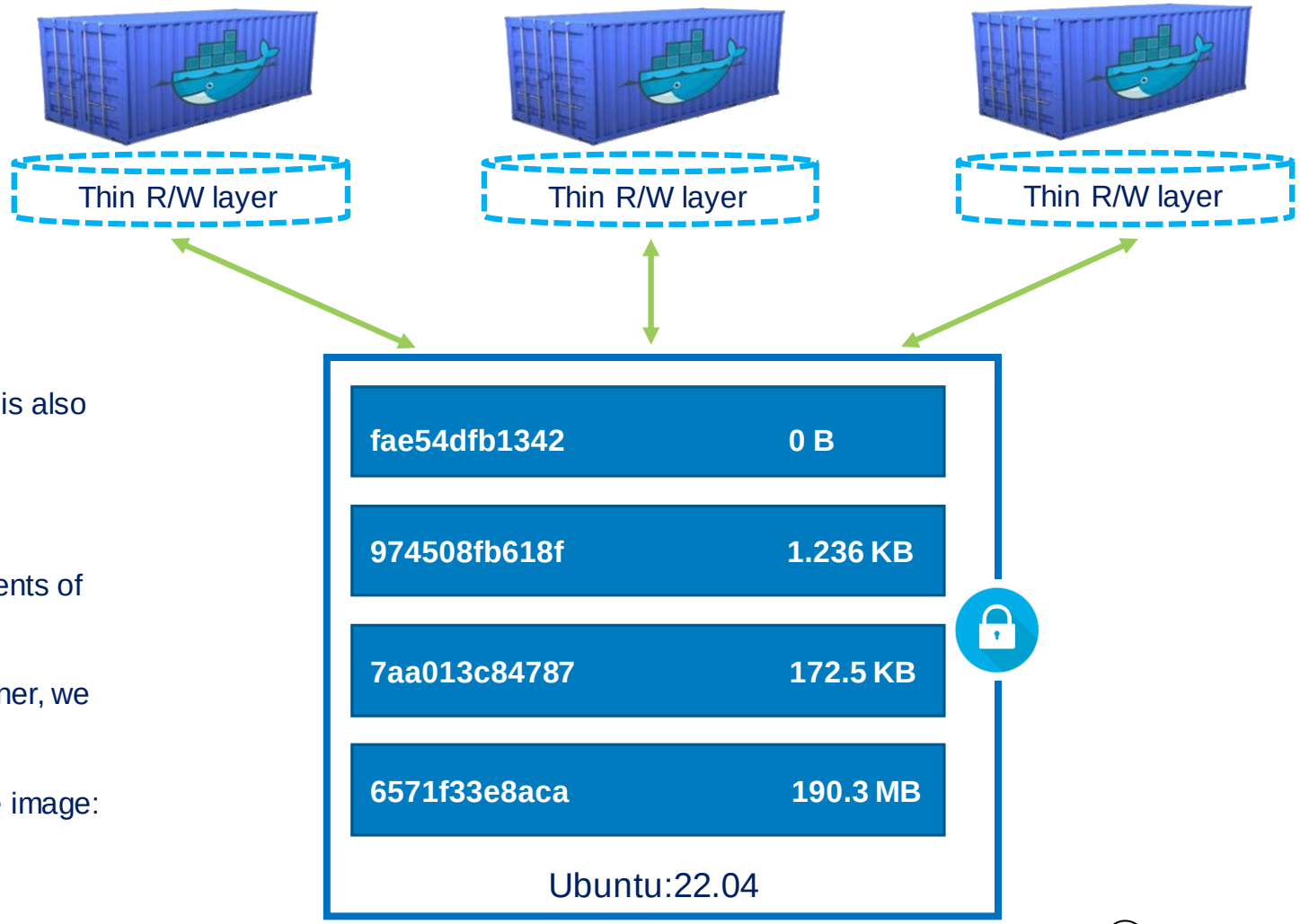
Docker Images

- A Docker image is a read-only template that contains a set of instructions for creating a container that can run on the Docker platform.
- With Docker images we can ensure consistent container environment through different platforms.
- Each of the files that make up a Docker image is known as a **layer**.
- These layers form a series of **intermediate images**, built one on top of the other in stages, where each layer is dependent on the layer immediately below it.
- The difference between container and images layers is the container on top **Thin R/W** (read/write) layer.
- Thanks to that - multiple containers can share access to the same underlying image and yet have their own data state.



Images and Containers layers and size on disk

- Multiple containers can share the same image
- All writes to the container that add new or modify existing data are stored in this writable layer
- When the container is deleted, the writable layer is also deleted
- The underlying image remains unchanged
- Docker uses storage drivers to manage the contents of the image layers and the writable container layer
- To view the approximate size of a running container, we can use the **docker ps -s** command:
- To check usage of containers that uses the same image:
  (**n** * **size**) + (**virtual size** – **size**)



CoW – Copy on Write

- Copy-on-write is a strategy of sharing and copying files for maximum efficiency
- If a file or directory exists in a lower layer within the image, and another layer (including the writable layer) needs read access to it, it just uses the existing file
- The first time another layer needs to modify the file (when building the image or running the container), the file is copied into that layer and modified
- This minimizes I/O and the size of each of the subsequent layers
- Any files the container does not change do not get copied to this writable layer
- The writable layer is as small as possible

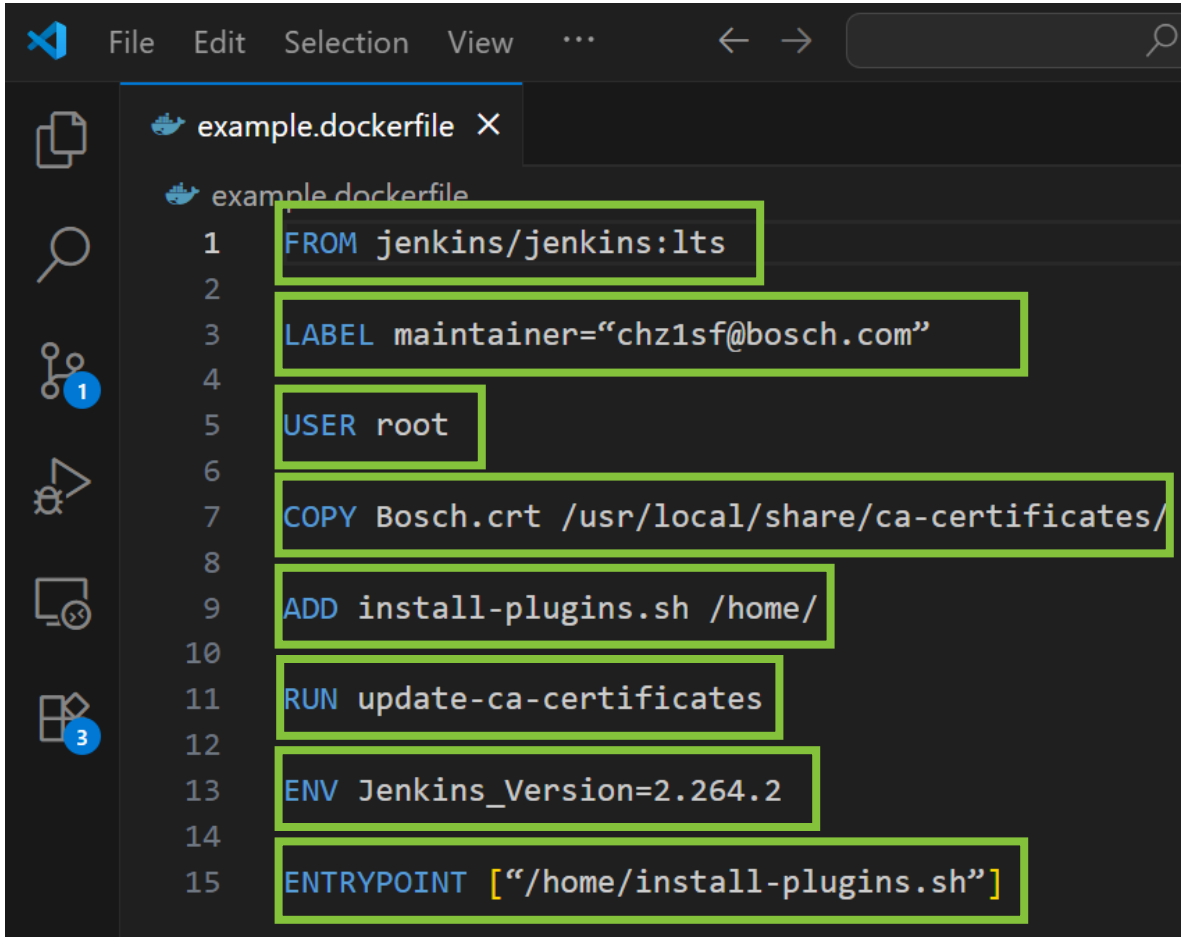
```
CONTAINER ID   IMAGE                     COMMAND   CREATED              STATUS              PORTS   NAMES
1a174fc216cc   acme/my-final-image:1.0   "bash"    About a minute ago   Up About a minute           my_container_5
38fa94212a41   acme/my-final-image:1.0   "bash"    About a minute ago   Up About a minute           my_container_4
1e7264576d78   acme/my-final-image:1.0   "bash"    About a minute ago   Up About a minute           my_container_3
dcad7101795e   acme/my-final-image:1.0   "bash"    About a minute ago   Up About a minute           my_container_2
c36785c423ec   acme/my-final-image:1.0   "bash"    About a minute ago   Up About a minute           my_container_1
```

```
                && docker run -dit --name my_container_2 acme/my-final-image:1.0 bash \
                && docker run -dit --name my_container_3 acme/my-final-image:1.0 bash \
$ sudo ls /var/lib/docker/containers                     me/my-final-image:1.0 bash \
                                                         me/my-final-image:1.0 bash
1a174fc216cccf18ec7d4fe14e008e30130b11ede0f0f94a87982e310cf2e765
1e7264576d78a3134fbaf7829bc24b1d96017cf2bc046b7cd8b08b5775c33d0c  81a0951fcc3fa0430c409
38fa94212a419a082e6a6b87a8e2ec4a44dd327d7069b85892a707e3fc818544  2b00bf05cd5a4343ea513
c36785c423ec7e0422b2af7364a7ba4da6146cbba7981a0951fcc3fa0430c409  46b7cd8b08b5775c33d0c
dcad7101795e4206e637d9358a818e5c32e13b349e62b00bf05cd5a4343ea513  9b85892a707e3fc818544
                1a174fc216cccf18ec7d4fe14e008e30130b11ede0f0f94a87982e310cf2e765
```

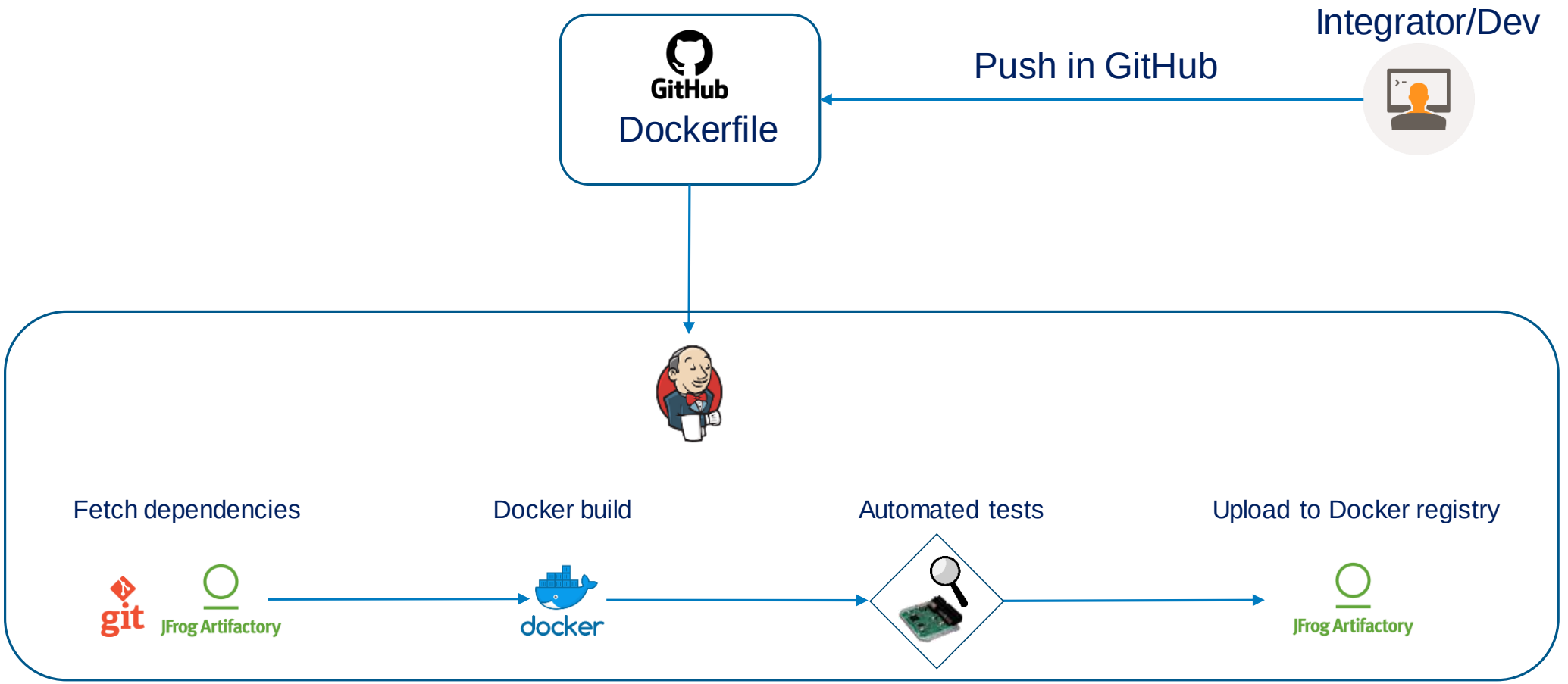
```
$ sudo du -sh /var/lib/docker/containers/*

32K  /var/lib/docker/containers/1a174fc216cccf18ec7d4fe14e008e30130b11ede0f0f94a87982e310cf2e765
32K  /var/lib/docker/containers/1e7264576d78a3134fbaf7829bc24b1d96017cf2bc046b7cd8b08b5775c33d0c
32K  /var/lib/docker/containers/38fa94212a419a082e6a6b87a8e2ec4a44dd327d7069b85892a707e3fc818544
32K  /var/lib/docker/containers/c36785c423ec7e0422b2af7364a7ba4da6146cbba7981a0951fcc3fa0430c409
32K  /var/lib/docker/containers/dcad7101795e4206e637d9358a818e5c32e13b349e62b00bf05cd5a4343ea513
```

Dockerfile

- A **Dockerfile** is a text file that contains all the commands a user could call on the command line to assemble an image.
- **Dockerfile** is basically template for creating or re-creating docker images
- Every instruction in Dockerfile represent separate layer during build process and creating of new image
- To build docker image out of a **Dockerfile** we have to use Docker **build** command.
- Useful docker build command arguments:
  - **-t** name and optionally tag the new image
  - **--no-cache** do not use already cached layers
  - **--pull** always attempt to pull newer version of the base image
  - **-f** Name of the dockerfile ('PATH/Dockerfile')
  - **--quiet** Suppress the build output and print image ID on success
- To check the rest of arguments we can pass to the docker build command we can run:
  - **docker build --help**



Docker image creation



BOSCH

Docker cache

- Docker uses cache mechanism to optimize:
  - storage utilization
  - build time

1st build of the Dockerfile
Clean build

```
docker build -t hello-world-react-docker:latest ./
Sending build context to Docker daemon  716.8kB
Step 1/7 : FROM node:12.13.1-buster-slim AS dev
12.13.1-buster-slim: Pulling from library/node
000eee12ec04: Already exists
d8fdfd9a7a73: Already exists
7f1fa99d89fa: Already exists
019be8afd290: Already exists
fc4a5846a37d: Already exists
Digest: sha256:1d1028c639a31211a16ca39832f388375d933e913141b7
Status: Downloaded newer image for node:12.13.1-buster-slim
 ---> c3dc5946eb6f
Step 2/7 : WORKDIR /home/app
 ---> Running in 9dc5abb2bc36
Removing intermediate container 9dc5abb2bc36
 ---> a6b034f19bd2
Step 3/7 : COPY ./package.json ./package-lock.json* ./
 ---> f48ff16b7965
Step 4/7 : RUN npm install
 ---> Running in 66840ccfa916

> core-js@2.6.11 postinstall /home/app/node_modules/babel-run
> node -e "try{require('./postinstall')}catch(e){}"
```

Build time ~ 5-10 minutes

2nd build of the Dockerfile
No changes

```
docker build -t hello-world-react-docker:latest ./
Sending build context to Docker daemon  716.8kB
Step 1/7 : FROM node:12.13.1-buster-slim AS dev
 ---> c3dc5946eb6f
Step 2/7 : WORKDIR /home/app
 ---> Using cache
 ---> a6b034f19bd2
Step 3/7 : COPY ./package.json ./package-lock.json* ./
 ---> Using cache
 ---> f48ff16b7965
Step 4/7 : RUN npm install
 ---> Using cache
 ---> 3b8e9d61a414
Step 5/7 : COPY public /home/app/public
 ---> Using cache
 ---> bf158f61c0e1
Step 6/7 : COPY src /home/app/src
 ---> Using cache
 ---> b6be0f20bedd
Step 7/7 : CMD ["npm", "start"]
 ---> Using cache
 ---> 4c2e20f75095
Successfully built 4c2e20f75095
Successfully tagged hello-world-react-docker:latest
```

Build time ~ 2 seconds

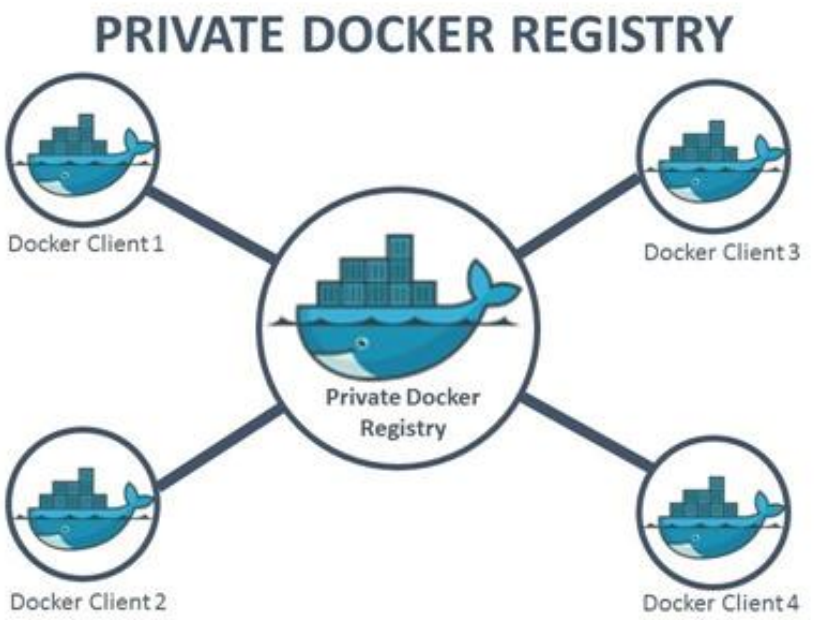
3rd build of the Dockerfile
Change in **src/App.js**

```
docker build -t hello-world-react-docker:latest ./
Sending build context to Docker daemon  716.8kB
Step 1/7 : FROM node:12.13.1-buster-slim AS dev
 ---> c3dc5946eb6f
Step 2/7 : WORKDIR /home/app
 ---> Using cache
 ---> a6b034f19bd2
Step 3/7 : COPY ./package.json ./package-lock.json* ./
 ---> Using cache
 ---> f48ff16b7965
Step 4/7 : RUN npm install
 ---> Using cache
 ---> 3b8e9d61a414
Step 5/7 : COPY public /home/app/public
 ---> Using cache
 ---> bf158f61c0e1
Step 6/7 : COPY src /home/app/src
 ---> 9d9d9024438c
Step 7/7 : CMD ["npm", "start"]
 ---> Running in 682b7a3f8cf3
Removing intermediate container 682b7a3f8cf3
 ---> d74f3d5b06c4
Successfully built d74f3d5b06c4
Successfully tagged hello-world-react-docker:latest
```

Build time ~ 5-10 seconds

Docker Registry

- A server that hosts your **Docker** Images.
- It acts in a fashion analogous to repositories in that you can push and pull images from the **registry**. You can use a public **registry** like **Docker** hub, or you can setup your own for private images.
- Tightly control where your images are being stored
- Fully ownership your images distribution pipeline
- Integrate image storage and distribution tightly into your in-house development workflow



How to use private Docker repository?

1st configure Docker daemon to connect with respective private repository. For this we have to modify:

- Linux - **/etc/docker/daemon.json**
- Windows - **C:\ProgramData\docker\config\daemon.json**

```
{
 "insecure-registries" : ["jfrog.sofia.bosch.com:8443", "kubernetes.sofia.bosch.com:30080"]
}
```

2nd Setup images to be uploaded to our private registry:

- We have to TAG our image to include Docker private registry in its name:
  - docker build -t ${image_name} .
  - **docker build -t jfrog.sofia.bosch.com:8443/VW/Integrity:v25 .**
  - docker tag ${image_id} ${image_name}:${TAG}
  - **docker tag 351160ba910d jfrog.sofia.bosch.com:8443/VW/linux:v3**

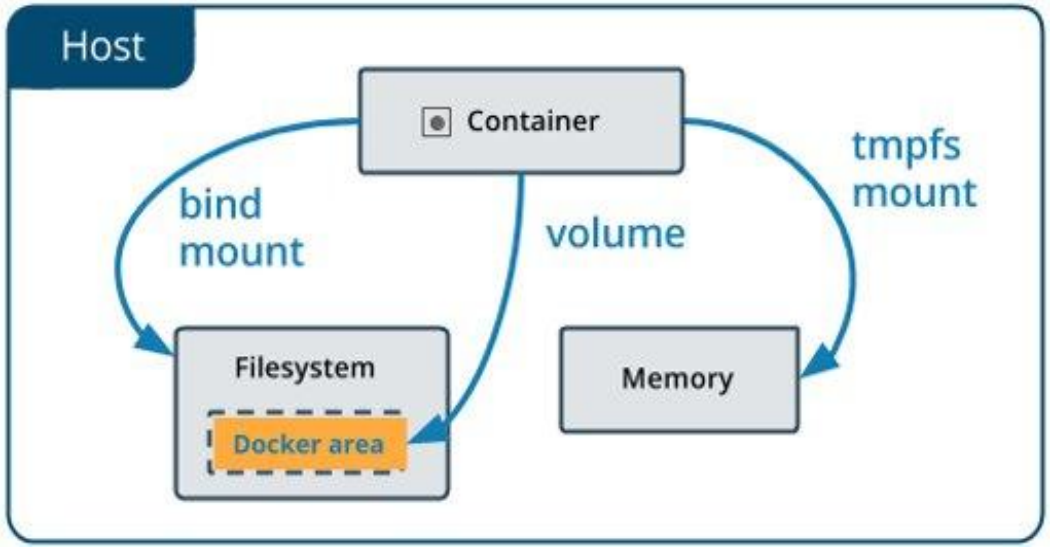
3rd Now we can download and upload images to our private repository:

- docker pull/push ${repository}/${path_to_image}:TAG
- **docker pull/push jfrog.sofia.bosch.com:8443/VW/Integrity:v25**

Docker Volumes

Docker Volume

- A **volume** is a specially-designated directory within one or more containers that bypasses the Union File System. **Volumes** are designed to persist data, independent of the container's life cycle. **Docker** therefore never automatically deletes **volumes** when you remove a container, nor will it "garbage collect" **volumes** that are no longer referenced by a container.
- **Volumes** are easier to back up or migrate than bind mounts.
- You can manage **volumes** using Docker CLI commands or the **Docker** API.
- **Volumes** work on both Linux and Windows containers.
- **Volumes** can be more safely shared among multiple containers.
- **Volume** drivers let you store **volumes** on remote hosts or cloud providers, to encrypt the contents of **volumes**, or to add other functionality.

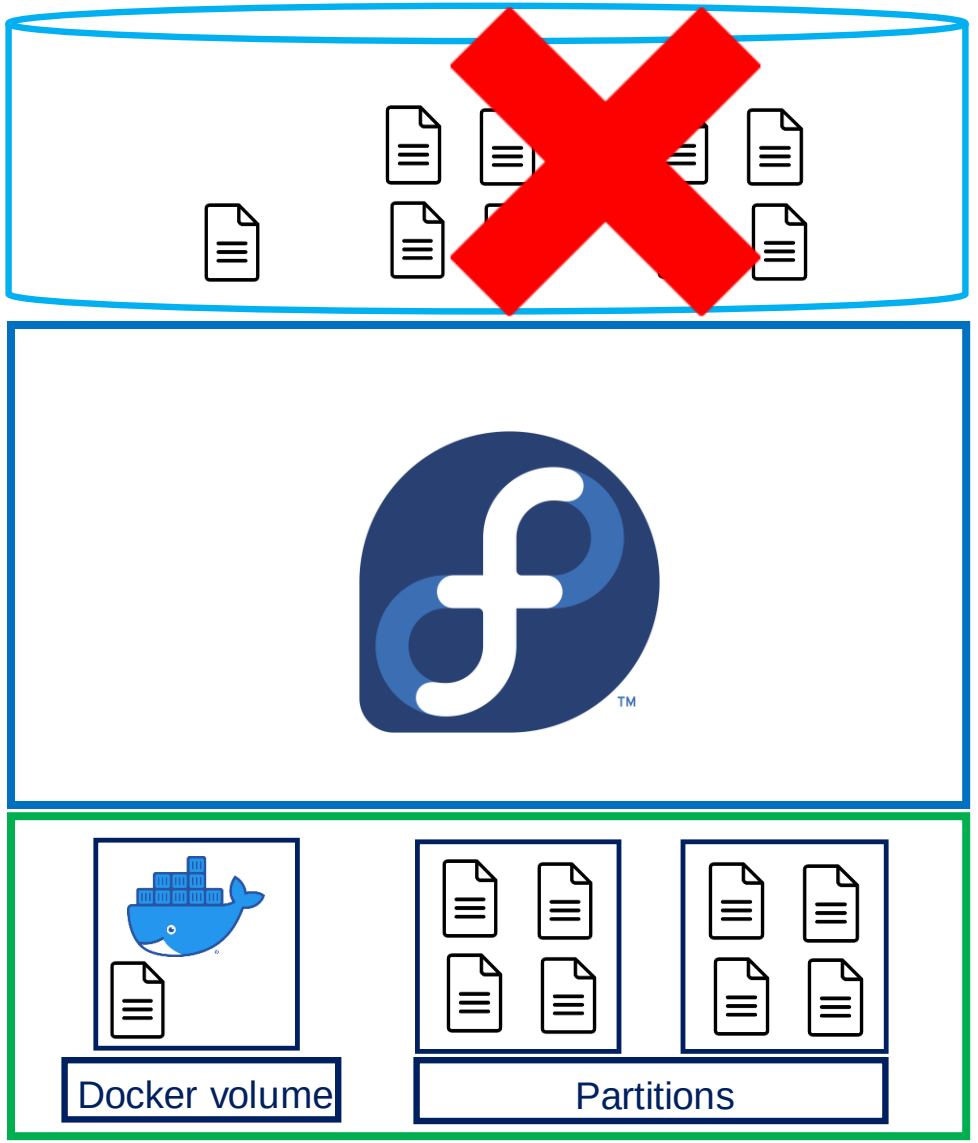


Docker Volume Usage

Container
Thin R/W layer

Image
R/O layers

Host Storage

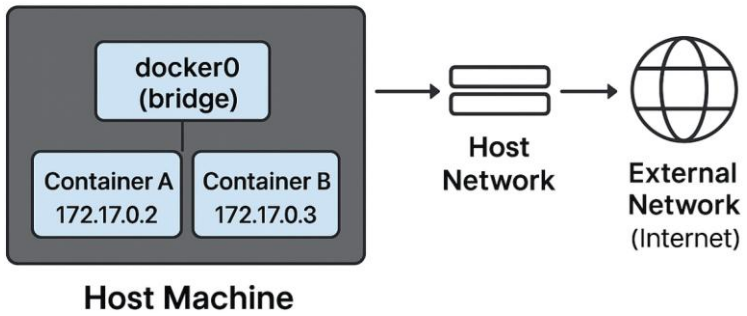


BOSCH

Docker Networking

Understanding Docker Networking

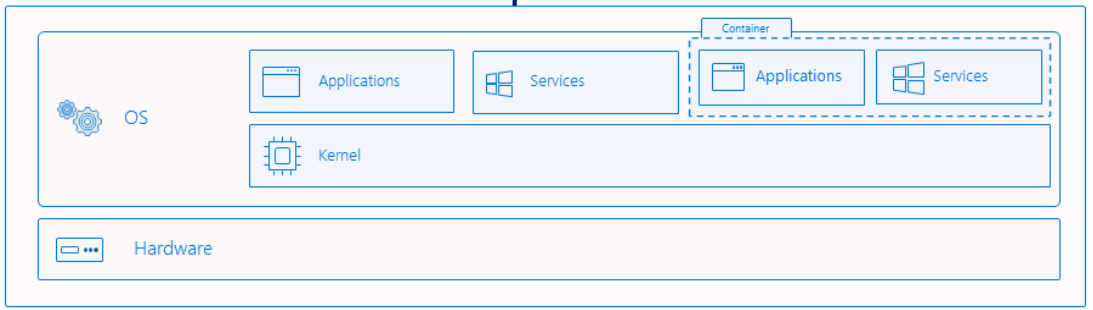
- Docker networking allows containers to communicate with each other, the host, and external networks.
- Each container gets its own **network namespace** (isolated network stack).
- Docker provides several **built-in network drivers** for different communication needs.
- **Key Docker Network Types:**



Network Type	Description	Use-Case
Bridge	Default Network; containers communicate on the same host via virtual bridge	Local container-to-container
host	Removes network isolation; container shares host network stack	High performance, host-level apps
none	No network; full isolation	Security, sandboxing
Overlay	Connects containers across multiple Docker Hosts	Swarm or multi-host setup
macvlan	Assigns MAC address to containers, appearing as physical devices	Integration with existing network

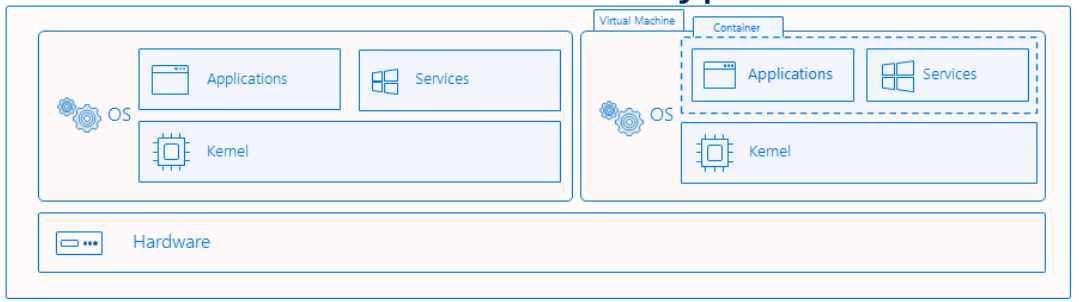
Isolation modes

Process isolation
--isolation=process



- “Traditional” isolation mode for containers
- Isolation through namespace, resource control and process isolation technologies
- All containers shares the same Kernel as host OS
- Approximately the same as how Linux containers run

Hyper-V isolation
--isolation=hyperv



- Containers runs inside of a highly optimized virtual machine
- Containers gets their own kernel
- This mode offers enhanced security
- Broader compatibility between host and container versions
- Hardware-level isolation between each container as well as the container host

- Windows Server default isolation level is **Process**
- Windows 11 Pro and Enterprise default isolation level is **Hyper-V**

BOSCH

Version compatibility

Settings → System → About →

Edition	Windows 11 Enterprise
Version	23H2
Installed on	21.10.2024 r.
OS build	22631.5909

- Cannot run container newer than the host OS version
- Older versions are supported only in **Hyper-V** isolation mode
- To run in **Process** isolation host OS and container version should be same
- Possible misbehavior can be observed when host OS version and container build version are different

Windows 11 host OS compatibility

Container base image OS version	Supports Hyper-V isolation	Supports process isolation
Windows Server 2025	✅ [1]	✅ [1]
Windows Server 2022	✅	✅
Windows Server 2019	✅	❌
Windows Server 2016	✅	❌

1. Supported from Windows 11 24H2 (`Build 2600`) onwards.

BOSCH

Docker Installation

Docker Installation on Ubuntu – Setup Apt Repository (1/2)

```
#Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "${UBUNTU_CODENAME:-$VERSION_CODENAME}") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

Docker Installation on Ubuntu – Setup Docker Service (2/2)

- Installation Method:

  sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin

  sudo systemctl enable docker

  sudo systemctl start docker

  sudo systemctl status docker

  sudo docker run hello-world

- Post Installation Steps:

  sudo groupadd docker

  sudo usermod -aG docker $USER

  newgrp docker

  docker run hello-world

Docker Installation – The easy way

```
curl -fsSL https://get.docker.com  -o get-docker.sh
sudo sh ./get-docker.sh  --dry-run
```

Docker Commands

Docker Cheat Sheet

- docker --version
- docker --help
- docker pull
- docker run
- docker build
- docker login
- docker push
- docker ps
- docker images
- docker stop
- docker kill
- docker rm
- docker rmi
- docker exec
- docker commit
- docker import
- docker export
- docker container
- docker compose
- docker swarm
- docker service

Basic Docker Commands (1/4)

docker build

The `docker build` command builds Docker images from a Dockerfile and a “context”.

$ docker build [OPTIONS] PATH | URL | -

EXAMPLE USAGE

1) $ docker build -f Dockerfile
2) $ docker build -t registry/image:latest
3) $ docker build --cpus="4.0" .
4) $ docker build --memory=8192m .
5) $ docker build --no-cache .
6) $ docker build --output .

OPTIONS

1)	-f	Name of the Dockerfile (Default is 'PATH/Dockerfile')
2)	-t	Name and optionally a tag in the 'name:tag' format
3)	--cpus	CPU shares (relative weight)
4)	--memory	Memory limit
5)	--no-cache	Do not use cache when building the image
6)	--output	Output destination (format: type=local,dest=path)

BOSCH

Basic Docker Commands (2/4)

docker run

This command executes a Docker Image on your machine and creates a running Container out of it.

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

EXAMPLE USAGE

1) $ docker run -d image
2) $ docker run -it image
3) $ docker run --cpus="4.0" image
4) $ docker run --memory=8192m image
5) $ docker run --entrypoint=/bin/bash image
6) $ docker run -p 80:8080 image

OPTIONS

1)	-d	Run container in background and print container ID
2)	-it	The -it instructs Docker to allocate a pseudo-TTY connected to the container's stdin
3)	--cpus	Number of CPUs
4)	--memory	Memory limit
5)	--entrypoint	Overwrite the default ENTRYPOINT of the image
6)	-p 80:8080	This binds port 8080 of the container to TCP port 80

Basic Docker Commands (3/4)

docker exec

The `docker exec` command runs a new command in a running container.

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

EXAMPLE USAGE

1) $ docker exec -d container cat /etc/hosts
2) $ docker exec --env var=1 container bash
3) $ docker exec --privileged container bash
4) $ docker exec --tty container bash
5) $ docker exec --user bosch container bash
6) $ docker exec -w container pwd

OPTIONS

1) -d	Detached mode: run command in the background
2) --env	Set environment variables
3) --privileged	Give extended privileges to the command
4) --tty	Allocate a pseudo-TTY
5) --user	Username or UID (format: <name|uid>[:<group|gid>])
6) --workdir	Working directory inside the container

BOSCH

Basic Docker Commands (4/4)

docker images

The default `docker images` will show all top level images, their repository and tags, and their size.

$ docker images [OPTIONS] [REPOSITORY[:TAG]

EXAMPLE USAGE	OPTIONS	
1) $ docker images -a	1) -a	Show all images (default hides intermediate images)
2) $ docker images --digests	2) --digests	Show digests
3) $ docker images --filter	3) --filter	Filter output based on conditions provided
4) $ docker images --format	4) --format	Pretty-print images using a Go template
5) $ docker images --no-trunc	5) --no-trunc	Don't truncate output
6) $ docker images --quiet	6) --quiet	Only show image IDs

BOSCH

Docker Lessons Learned

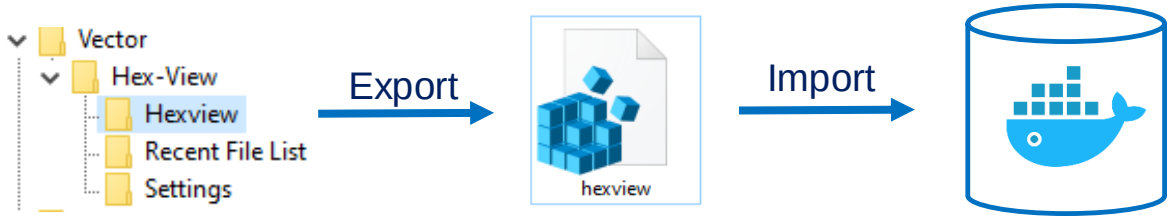
Docker desktop - Lessons learned 1/2

1st Challenge:

How to run legacy mixed graphical/command line **.exe** programs inside Windows container (like HexView)?

Solution:

Regedit



Dockerfile

COPY hexview.reg c:/
RUN REG IMPORT c:/hexview.reg

HexView

VECTOR

BOSCH

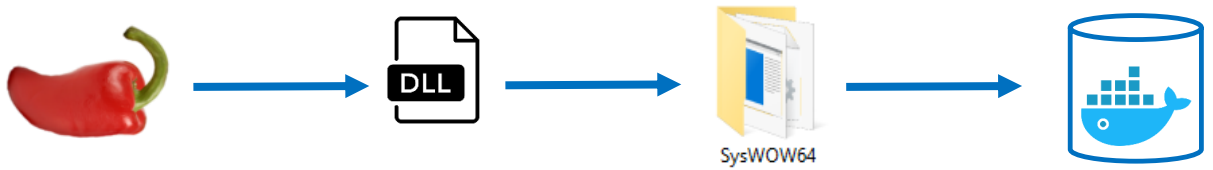
Docker desktop - Lessons learned 2/2

2nd Challenge:

How to run successful 32-bits programs and utilities inside Windows containers without mounting host folders?

Solution:

CFF Explorer → Missing DLL's → Copy SysWOW64 → Updated image



HexView

VECTOR

Dockerfile

COPY SysWOW64 c:/SysWOW64
RUN powershell Robocopy C:\SysWOW64\ C:\Windows\SysWOW64\ /XC /XN /XO ; \ Remove-Item C:\sysWOW64 -Recurse

BOSCH

Docker container as Jenkins agent

Subgroups and projects | Shared projects | Archived projects | Search by name | Last created

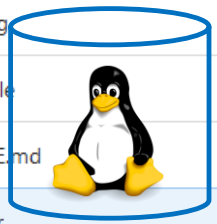
Linux Base Image ★ 1 2 months ago

Windows Base Image ★ 0 2 months ago

Repository content

https://git.bosch.com/devops/ci-cd-automation/docker

Name	Last commit
.repoconfig	first version
.ssh	first version
PortableGit	first version
host_tools/jazz_6.0.6.1.IBM/jazz	first version
jre1.8.0_72	first version
.gitconfig	first version
Dockerfile	Optimize dockerfile
README.md	Update README and Dockerfile
agent.jar	first version
jenkins-agent.ps1	first version

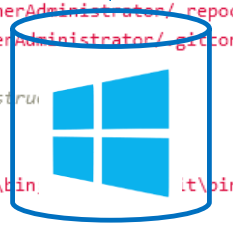


Linux Base Image

git@git.bosch.com:devops/ci-cd-automation/docker/linux-base-image.git

Dockerfile

```
# Indicates that the windowsservercore image will be used as the base image.
FROM python:2.7-windowsservercore-1809

# Copy jenkins slave connection files and java
COPY jre1.8.0_72 c:/jre1.8.0_72
COPY jenkins-agent.ps1 C:/ProgramData/Jenkins/jenkins-agent.ps1
COPY agent.jar C:/ProgramData/Jenkins/agent.jar

# Copy tools needed for successful fetch from Git and RTC
COPY host_tools c:/host_tools
COPY PortableGit c:/PortableGit

# Project related configuration files for successful pull from GitLab with Git/Repo
COPY .ssh c:/users/ContainerAdministrator/.ssh
COPY .repoconfig c:/users/ContainerAdministrator/.repoconfig
COPY .gitconfig c:/users/ContainerAdministrator/.gitconfig

# Insert your project related instru

# path env variables
ENV Path="c:\PortableGit\mingw64\bin                    t\bin;c:\PortableGit\usr\bin;c:\PortableG
        repo=repo

# Sets a command or process that will be executed                    is run from the new image.
ENTRYPOINT ["powershell.exe", "-f", "C:/ProgramData/Jenkins/jenkins-agent.ps1"]
```



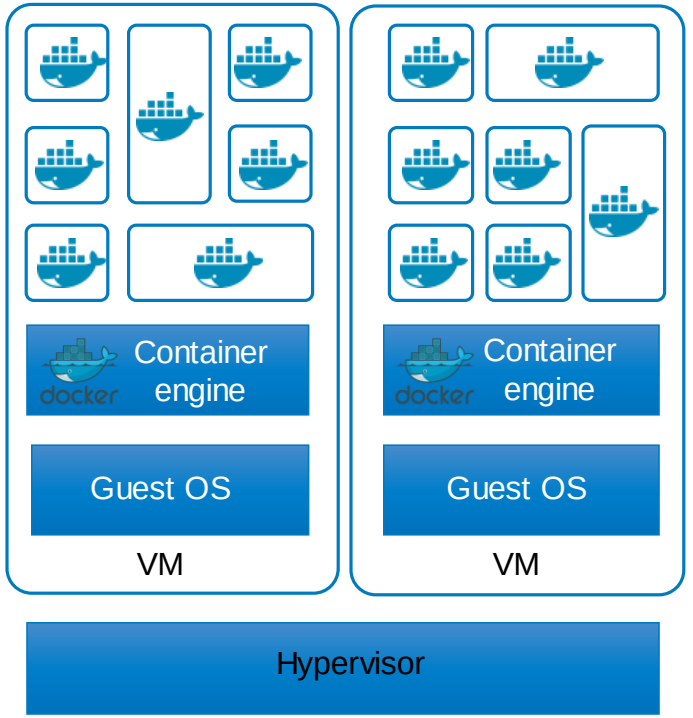
Windows Base Image

git@git.bosch.com:devops/ci-cd-automation/docker/linux-base-image.git

BOSCH

Kubernetes Orchestration

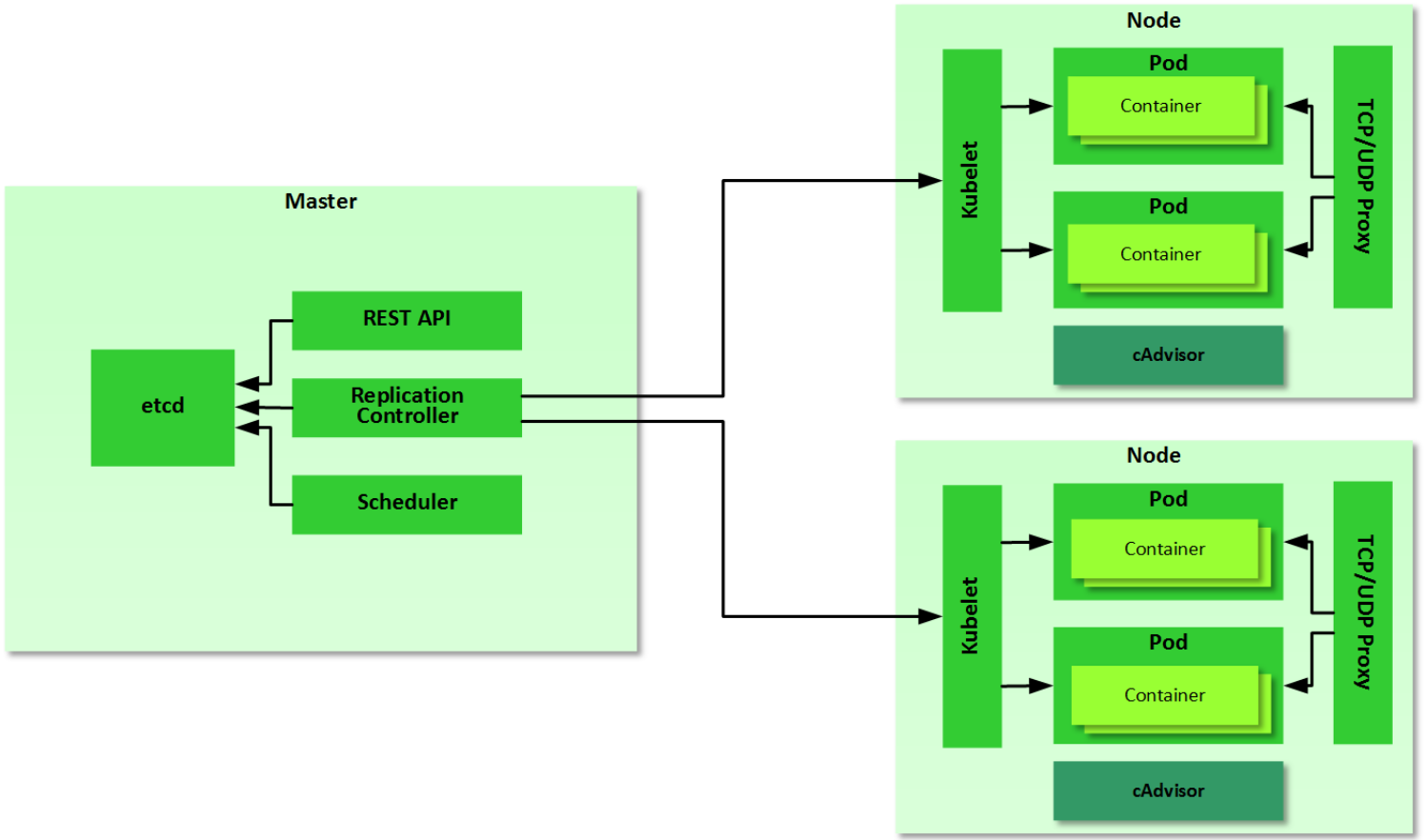
Docker needs platform



Docker is just a container engine many things are not there

- Networking
- Storage
- Service discovery
- Container scheduling
- Placement and load balancing
- Routing
- Self healing

Kubernetes architecture



Resources and Valuable Links

- https://www.docker.com/
- https://docs.docker.com/
- https://hub.docker.com/
- https://github.com/docker
- https://labs.play-with-docker.com/
- https://kubernetes.io/docs/home/
-

Questions & Answers

BOSCH