Virtualization with Containers and Kubernetes

Introduction with a focus on Docker

Andreas Roth

Training

09.06.2021

Contents

1 Virtualization

2 Containers

3 Docker

4 Kubernetes

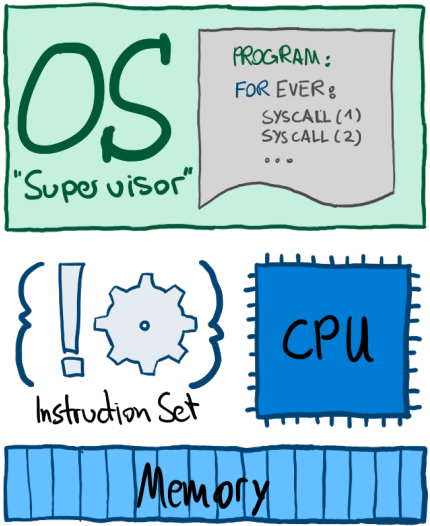
Virtualization

Virtualization

Classical view of **a computer** running **an operating system**, running **a program**.

- OS is in full control of the hardware
- A program runs with lower privilege
- OS assigns resources to program

OS acts as a **supervisor** for programs.



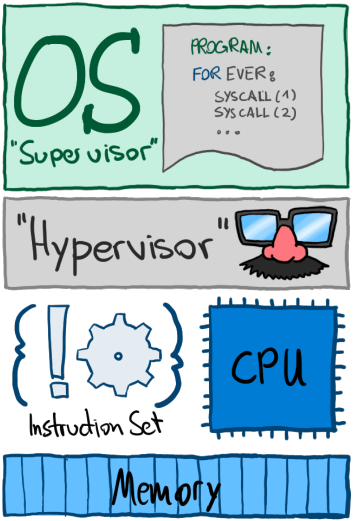
Virtualization

Virtual Machine Monitor[1] (**Hypervisor**)

1. provides environment essentially identical to original machine
2. in full control of system resources
3. Programs run with minimal speed decreases in this environment

Virtual Machine[1]

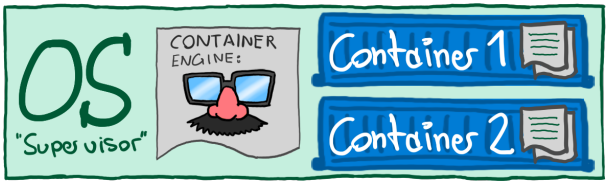
The environment for **programs** to run in, with a **Hypervisor** present, is called a **Virtual Machine**.



[1] Popek G.J., Goldberg R. P. *Formal Requirements for Virtualizable Third Generation Architectures*, Commmunications of the ACM, Volume 7 (17), 1974

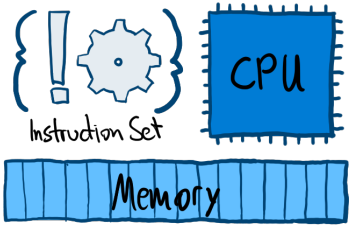
Containers

”Containerization” - A kind of virtualization...



OS-level virtualization[1]

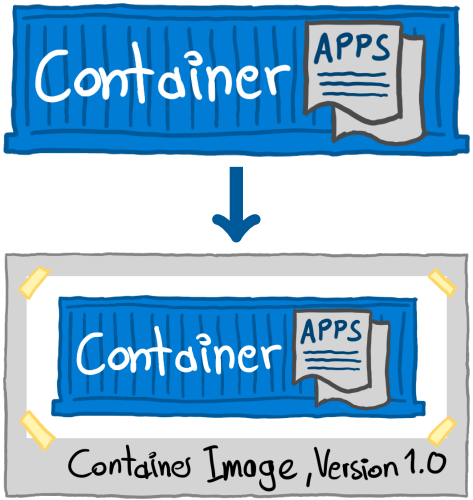
OS allows multiple instances of isolated user spaces (**containers**) to run applications in. They all share the OS Kernel.



- Processes inside the container cannot see anything outside (processes, files, ...). Physically, they are processes running on the same OS Kernel, however.
- There is no need for a virtualization layer below the Kernel to run containers, although there can be one!
- A container engine or runtime manages the containers

[1] *OS-level Virtualization*, Wikipedia 2021, https://en.wikipedia.org/wiki/OS-level_virtualization

Containers



Why?

- Bundle up an application with all its dependencies!
- More lightweight and portable than a virtual machine

How?

- Containers can be shipped as images (distribution infrastructure depends on system / engine)
- Who runs a container based on my image can be sure to have a functionally identical environment, independent of application type

How to make containers?

Advanced use of Linux features!

namespaces

- Limit what a process can see
- Change filesystem root with `chroot`
- Change apparent `pid`s
- ...

cgroups

- Limit what a process can use
- Hierarchical definition tree to assign quantities of memory, cpu cores, ... to processes and groups of processes

Definitely watch those talks:

- Containers from scratch (Liz Rice, 2018): https://www.youtube.com/watch?v=8fi7uSYlOdc
- Namespaces, cgroups and beyond: What are containers made from? (Jérôme Petazzoni, 2015): https://www.youtube.com/watch?v=sK5i-N34im8

Container runtimes / engines

There are / might be several software layers between the user interacting with a **container engine** and the **runtime** that actually runs containers.

Based on Linux namespaces and cgroups:

- LXC
- Docker (Docker engine, containerd)
- Podman

Other mechanisms / systems:

- FreeBSD Jail (Unix)
- Solaris Zones (Unix)
- Windows Containers

Docker

Docker

- Runtime and engine
- Building / managing of images / containers
- Distribution of images (Docker Hub)
- Interface to Windows and MacOS, different mechanisms
- (Orchestration of containers) swarm, `docker-compose`
- Docker engine exists as open source, Docker Desktop and Docker Hub are developed by Docker Inc.

Docker is the most relevant container tool for us.

We will take a short tour of the most important Docker concepts:

- Running containers
- Building images with `Dockerfile`s
- Docker registries (Docker Hub, company self-hosting solutions)
- Pushing / pulling images
- Docker volumes

Code-along exercises

- Feel free to try out the commands presented on the next slides during the training!
- You need a Docker Desktop installation (or Docker Engine on Linux)
- As a preparation, clone github.com/scherbertlemon/docker-training

Container registries

... where prepared container images come from, if you do not build them based on your host system. They are usually organised in versioned image repositories.

Docker Hub hosted by Docker Inc.

- Open registry where you can store your images in **repositories**
- Public repositories are free of charge
- Many officially curated images

Your company solution here

- you might want to host your own registry

- Connect to a registry with `docker login <url>`. If no URL, Docker Hub is the default.
- Generate an Access Token if possible and use it for login instead of your password.

Images and Containers

Image

Read only content of a container, e.g. file system, environment variables, metadata

- Organized in layers (”differences”)
- We usually pull / push images from / to **container registries**

Container

Extracted image content plus writable layer, that can be run on the host OS.

- We usually run processes in a container
- Changes to the file system exist as long as the container is not stopped and removed

Images and Containers


```
# get the offical ubuntu image
docker pull ubuntu:focal
# check the local images
docker image ls
# check the properties of this image
docker image inspect ubuntu:focal
# remove the local image
docker image rm ubuntu:focal
```

```
# get an interactive shell in ubuntu
docker run -it --name ubu ubuntu:focal
# check the list of containers
docker container ls --all
# check the properties of ubu container
docker container inspect ubu
# remove stopped container
docker container rm ubu
```

Building images with Dockerfiles

```
FROM ubuntu:focal

LABEL maintainer="someone"

ENV TZ=Europe/Berlin

RUN apt update \
    && apt install \
    tzdata \
    --yes --no-install-recommends \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /wdir
# goes to wdir
COPY localfile.txt .

ENTRYPOINT [ "date" ]
```

Example that prints the time when run

Dockerfile

... a recipe to perform certain tasks starting from an image, controlled by **directives**, to create a new, modified image.

- find this Dockerfile in `dockerfiles/time`
- Dockerfile reference
- Best practises
- Directives: `FROM`, `ENV`, `RUN`, `COPY`, `ENTRYPOINT`, `CMD`

Building images with Dockerfiles

```
FROM ubuntu:focal

LABEL maintainer="Someone"

ENV TZ=Europe/Berlin

RUN apt update \
    && apt install \
    tzdata \
    --yes --no-install-recommends \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /wdir
# goes to wdir
COPY localfile.txt .

ENTRYPOINT [ "date" ]
```

Example that prints the time when run

```
# build an image from Dockerfile in current
# folder
$ docker build -t time:0.1 .
# run it as a container
$ docker run --rm time:0.1
Sun May 30 19:24:10 CEST 2021
# give an argument to date, UTC time
$ docker run --rm time:0.1 -u
Sun May 30 17:24:50 UTC 2021
```

- tag/name the image with `name:version`
- `.` indicates the build context (current folder)
- `--rm` deletes the container after it has stopped running

Building a more complex Dockerfile

Simple greeting app

- We install miniconda into our container.
- We create a Python environment with `requirements.txt` in `/pysource/env`
- We run the `flask` app defined in `greeting.py` in that environment

- navigate to `dockerfiles/greeting/0.1`
- look at the Dockerfile
- build&run it with

```
docker build -t greeting:0.1 .
docker run --rm -it -p 5000:5000 greeting:0.1
```

- Open your browser at `localhost:5000/hello/yourname`

- `-p host:container` publishes ports from container to host.
- Storage is just in memory: when app is restarted, counter is reset

Running multiple containers

Greeting app with database

- We need to run 2 containers: 1 for the app, 1 for the database
- We put them in a docker network `dbtest` and hook up the app to host port 5000 again

- navigate to `dockerfiles/greeting/0.2`
- different code than `greeting:0.1`
- build it with

```
docker build -t greeting:0.2 .
```

- Did this take as long as building `greeting:0.1`? → build cache
- Expects a postgres database on host `post:5432` with password `holymoly`

```
# create a network
docker network create dbtest
# run postgres container
docker run --rm --name post \
    --network dbtest -d \
    -e POSTGRES_PASSWORD=holymoly \
    postgres:latest
# run app container
docker run --rm --name greet \
    --network dbtest -d -e PG_HOST=post\
    -p 5000:5000 greeting:0.2
# check the network
docker inspect dbtest
```

Docker volumes - persisting data

If we restart the `post` container, the recorded data is lost!

Volumes

Persisting data by mounting persistent storage into the container file system.

Persistent storage can be

- Bindings to host file system
- Named volumes

Use a named volume for `post` container:

```
# stop the container if still running
docker container stop post
# re-run with named volume
docker run --rm --name post \
    --network dbtest -d \
    -e POSTGRES_PASSWORD=holymoly \
    -v pgdata:/var/lib/postgresql/data
    postgres:latest
# look at what you did
docker volume ls
docker inspect pgdata
```

- You can specify bind mounts similarly with `-v hostpath:containerpath`
- Also works for Windows paths, but performance may be inferior

Docker compose - running multi-container apps

- In order to tidy up our mess from before:

```
docker container stop greet post
docker network rm dbtest
```

- Is this not tedious? → `docker compose`!
- Navigate to `dockerfiles/greeting`
- Run to the same effect

```
# run containers in background
# leave -d for interactive
docker compose up -d
# tidy up
docker compose down
```

- `docker-compose.yml` tells docker compose what to do

`dockerfiles/greeting/docker-compose.yml`

```
services:
  greet:
    image: greeting:0.2
    build: ./0.2/
    ports:
      - "5000:5000"
    environment:
      - PG_HOST=post
  post:
    image: postgres:latest
    volumes:
      - pgdata:/var/lib/postgresql/data
    environment:
      - POSTGRES_PASSWORD=holymoly
volumes:
  pgdata:
    external: True
```

Pushing images to registry

- Assume you have built `greeting:0.2` locally.
- Create a repository on Docker Hub or your own registry: `yourname/greeting`
- Tag the image to push with that name and push it:

```
docker tag greeting:0.2 yourname/greeting:0.2
docker push yourname/greeting:0.2
```

- You need to be logged in to your registry (`docker login`)!

Golden rules / best practises for containers

- Keep images small!
  (no unnecessary image content, delete setup files, Multi-stage builds)
- Every container should do one job, and do that well!
  (no need to run multiple applications in the same container, e.g. webserver and database)
- Don't use the writable layer of a container as storage, use volumes!
  (if you store something in your container, bind the location either to a named volume or the host file system)
- Use official images whenever you can!
  (Official Images on Docker Hub. Inside a company network, you can at least use the Dockerfiles as guidance)
- Adhere to the best practises for Dockerfiles!
  as outlined here

Docker tips for salvaging disk space

- Use the `prune` commands when you run out of disk space

```
# with option --all also named local images will be deleted
docker image prune
# in case you did not use --rm with docker run
docker container prune
# or everything at once, except volumes
docker system prune
```

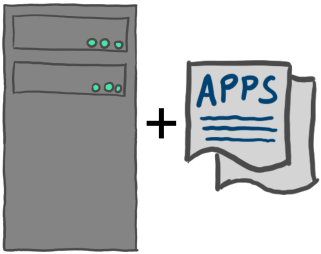
- On Windows: Pruning alone does not help, as the virtual disks of the Linux VM do not shrink.
  (Use Docker Desktop → Troubleshooting → Purge Data)
- On Windows: Use WSL 2 Backend, if possible!
  (Good look behind company proxy server, though!)

Kubernetes

Scaling applications / services

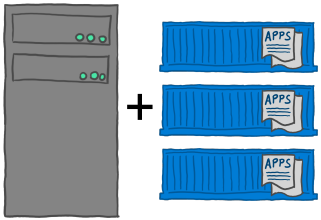
Monolithic Architecture

"Our applications run on one server (e.g. webserver and database). If you need to update, you have to power the whole system down."



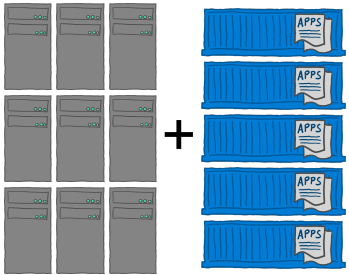
Containerized architecture

"We test our applications and ship them to our production server(s) with all dependencies. On update, downtime is really small."



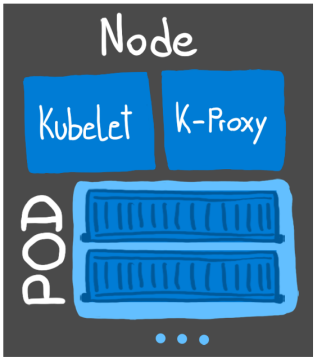
Distributed, managed architecture

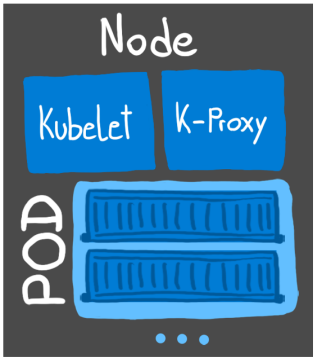
"Our applications consist of independent microservices that communicate via interfaces, they can be replaced / updated without downtime. The system itself dynamically adapts to workload by scaling up/down."



Kubernetes (K8s) according to the documentation:

”Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation.”





Kubernetes objects

Just a few should be mentioned here:

- Deployment: packaged application consisting of several containers run in a Pod. Can be replicated for load balancing.
- Service: A connection to an application, "pod-agnostic" for availability
- Pods, Nodes, etc. are Kubernetes objects too! Get information about them with

```
# info for all objects of one type
kubectl get object
kubectl describe object
# info for a specific object
kubectl get object objectname
kubectl describe object objectname
```

- Control Plane controls a cluster of Nodes
- containers are deployed in Pods on Nodes

Interacting with the cluster: kubectl

kubectl

is a CLI tool to send commands to the API server for creating Kubernetes objects.

`kubectl` creates objects

- imperatively: Everything explicitly on the command line
- declaratively: with a spec representing the desired state (`.yaml` -file)

Let the cluster work to realize your spec:

```
kubectl apply -f kubernetes/greeting-depl-0-1.yaml
kubectl apply -f kubernetes/greeting-depl-0-2.yaml
```

Declarative way is preferred and more practicable!

Create a greeting app deployment and service

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: greeting-depl
spec:
  selector:
    matchLabels:
      app: greetingapp
  replicas: 2 # 2 pods of template
  template:
    metadata:
      labels:
        app: greetingapp
    spec:
      containers:
      - name: greet
        image: greeting:0.1
        ports:
        - containerPort: 5000
```

Example: `kubernetes/greeting-depl-0-1.yaml`

```
kubectl apply \
    -f greeting-depl-0-1.yaml
# see what just happened
kubectl get deployments
kubectl get pods
kubectl logs podname containername
```

Make available by creating a service:

```
kubectl expose deployment \
    greeting-depl \
    --type=NodePort \
    --port 5000
# note the port it gets mapped to
kubectl get services
# run proxy for access on localhost
kubectl proxy
```

Update greeting app deployment

Example: `kubernetes/greeting-depl-0-2.yaml`

- Updated greeting image version
- Added postgres container

Declare the new spec:

```
kubectl apply -f greeting-depl-0-2.yaml
```

- Pods are replaced one by one, without the service terminating!
- Eventually, you will see the newer greeting app.

Tidy up:

```
kubectl delete service greeting-depl
kubectl delete deployment greeting-depl
```

Containers and pods are terminated!

Concluding notes

- replicas > 1: we do not know which of our apps we will get (load balancing, no persistent storage)
- persistent storage is possible, here only stateless applications were shown
- **Not mentioned here:** security, encryption, secrets
- Docker Desktop users can activate a one-node kubernetes cluster in the settings
- Another one-node try-out solution: Minikube

Further reading:

- Kubernetes documentation: Includes lots of learning resources and examples.
- Play with Kubernetes classroom: Interactive training session giving an overview over concepts.

Thank you for participating!