Technische Hochschule Rosenheim
Technical University of Applied Sciences

Microservices

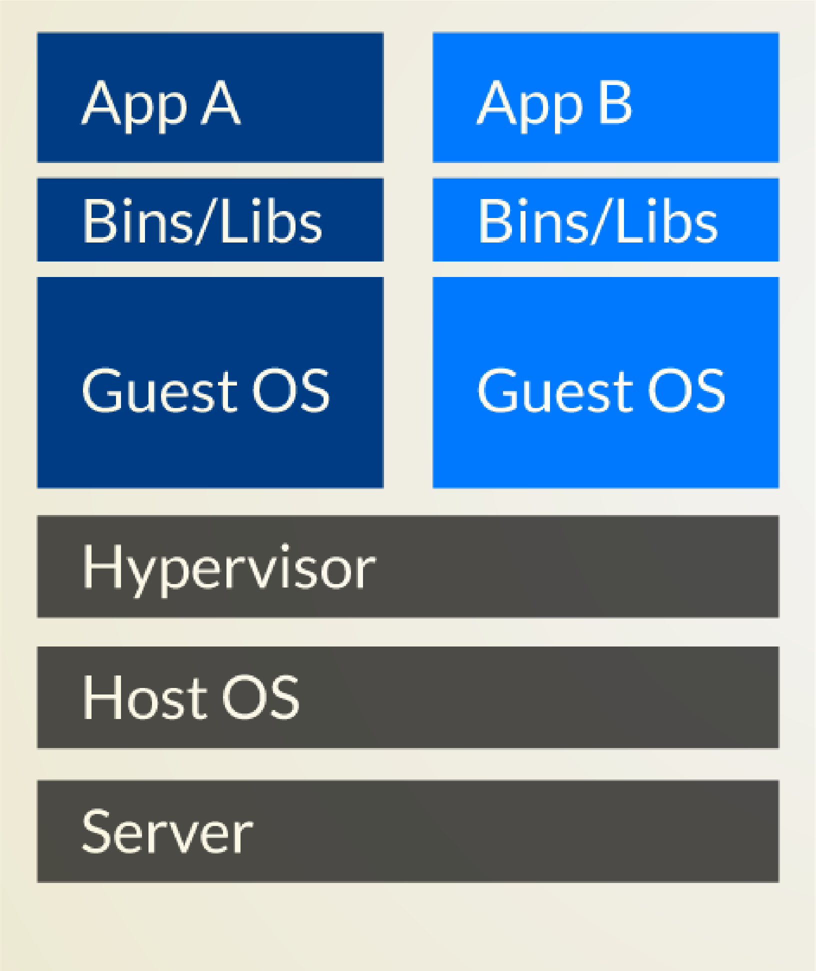
Container - Basics

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Content

1. Container vs. virtualization
2. Container basics
3. History of Docker
4. Docker CLI
5. Creating custom images
6. Dockerfiles
7. docker-compose

Container vs. virtualization



App A | App B

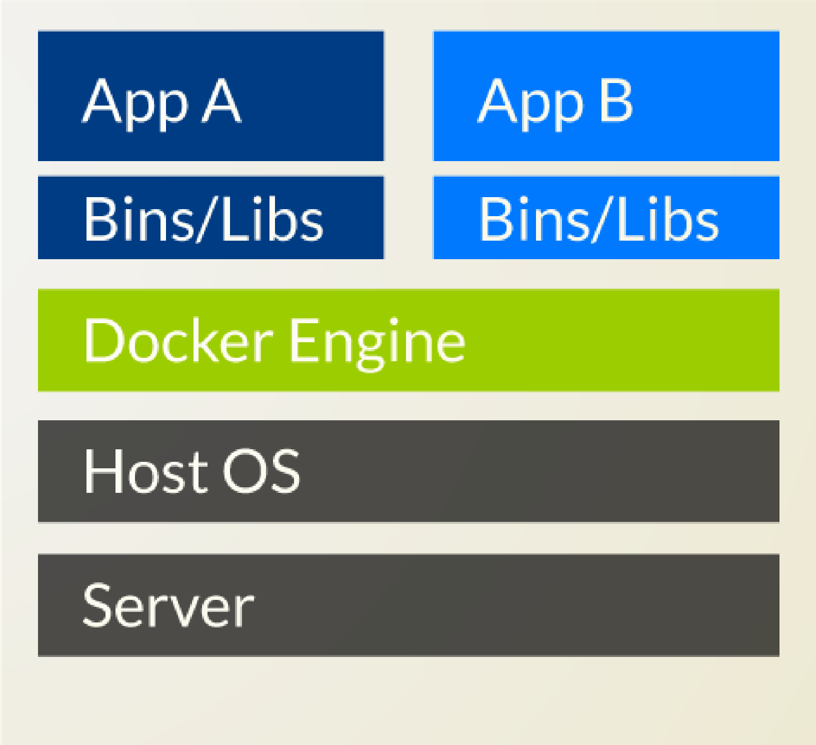
Bins/Libs | Bins/Libs

Guest OS | Guest OS

Hypervisor

Host OS

Server



App A | App B

Bins/Libs | Bins/Libs

Docker Engine

Host OS

Server

Virtualization - Pros

- Many excellent hypervisors available
- Feels like "ordinary" systems for administrators and developers
- No special knowledge needed

Virtualization - Cons

- Resource overhead for every virtual machine
- Takes longer to setup if no special tools are used (Puppet, Chef,... covered later)

Containers - Pros

- Lower resource overhead
- Very fast setup when prebuilt container images are available
- Isolation of every container
- Stable environment

Containers - Cons

- Training of developers and administrators required
- Getting complex if cluster is required (Kubernetes, DC/OS, Swarm)
- Backup

Container basics

Every container engine (e.g. Linux Containers (LXC) or Docker) is based on two Linux Kernel features:

- Process namespces
- Control groups (cgroups)

Namespaces

- Introduced in the Linux Kernel 2002
- Inspired by Plan 9 from Bell Labs
- Later on extended to support Process ID (PID) namespaces

Namespaces used in Docker:

- pid - PID isolation
- net - managing network interfaces
- ipc - managing access to inter-process communication (IPC) resources
- mnt - managing filesystem mount points
- uts - isolating kernel and version identifiers (Unix Timesharing System (UTS))

Control groups

- cgroups are used to restrict the resources a process or process group can allocate
- cgroups can be used to restrict classic resources e.g. RAM or CPU shares a proceses can allocate
- it is also possible to utilize cgroups to restrict access to certain devices

History of Docker

- 01/19/2013 - Initial commit
- 03/01/2013 - First announcement
- 10/29/2013 - Rebranding of dotCloud to Docker Inc.
- 07/01/2014 - Microsoft, Red Hat, IBM, Docker,... joined the Kubernetes project
- 10/15/2014 - Microsoft announces Docker support in Windows Server 2016
- 07/21/2015 - First Kubernetes version
- 06/08/2016 - Native Docker support with Hyper-V on Windows
- 06/20/2016 - Docker Swarm is built-in with Docker 1.12
- 07/19/2018 - Docker for Mac/Windows 18.06 stable ships with Kubernetes built-in

CONTAINERS, CONTAINERS
EVERYWHERE
makeameme.org

Why Docker?

- No more time consuming setup of servers:
    - Dependencies
    - Configurations
    - Documentation for administrators
- Every application/component can be packaged in a container (almost)
- Upgrades of containers are fast (if done right)
- Developers to not need to setup a heavy development environment but just start a few containers (docker-compose!)

- Administrators "only" have to pull the container images to deploy to production
- Containers can easily scale out (think of 5 containers of the service instead of just 1)
- Rollback of an entire application/a single component is possible by switching the container image (with proper tagging)

Why Docker in Microservices?

- Create a container per service
- Scale out a single service instead of the whole application by deploying more containers
- Continuous integration & continuous delivery
  - E.g. automatically build new container images
  - Deploy new containers to testing, staging (and optionally production) environments depending on which branch you are building
- Existing eco systems for service discovery and distributed configuration (see chapters 10 and 11)
- Clustering solutions available (Kubernetes, Docker Swarm, DC/OS Mesos,...)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Docker CLI - Basics

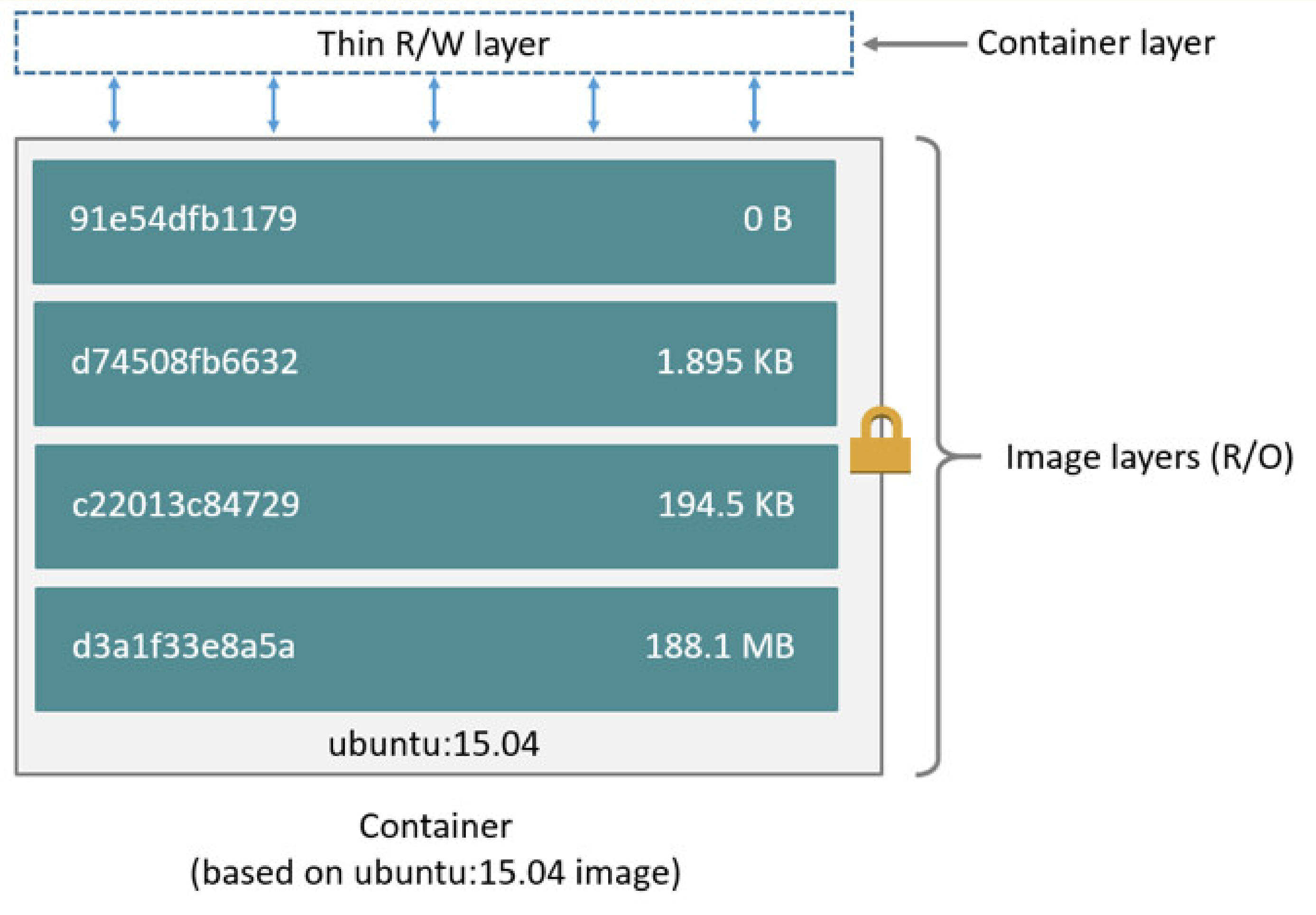
CMDlet	Explanation
`docker --help`	
`docker ps`	Show running containers
`docker ps -a`	Show all existing containers (including stopped)
`docker images`	Show all local images
`docker run –ti <image[:version]>`	Start a new interative container
`docker run –d <image[:version]>`	Start a new container i daemon mode (in the background)

CMDlet	Explanation
docker rm <container id/name>	Removes an existing container if it is stopped
docker rm –f <container id/name>	Removes an existing container even it is still running
docker exec –ti <container id/name>/bin/bash	Attaches a Bash instance to a running container
docker rmi <image id>	Removes a local container image
docker stop <container id/name>	Stops a running container
docker commit <container id/name> [repository[:tag]]	Create a new image from an existing container

Docker CLI - Build and registry

CMDlet	Explanation
`docker build [[registry/]user/image name]`	Create a new container based on a Dockerfile in the same directory
`docker build [[registry/]user/image name:tag]`	Create a new container based on a Dockerfile and add a tag to it
`docker push [[registry/]user/image name]`	Push a built image to a registry (default is Docker Hub)
`docker login [registry`	Login to a (private) Docker registry

Docker images



Concepts

- Every image consists of one or multiple layers
- Every layer is a kind of a snapshot and can be removed
- It is possible to inspect how a specific layer was created
- Each layer is nothing more than a `.tar.gz` archive which will be applied to a base image whenever a container is created
- When an image is rebuilt, the Docker daemon recognizes which layers aren't affected and is keeping them as they are to speed up the build process

Creating a new Docker image

- There are two ways to create a new Docker image:
  - Create a new container, do all required changes on your own e.g. via bash and commit the changes
  - Create a Dockerfile, describe all changes which have to be made to the base image and build it with the Docker CLI
- Most container are built with with Dockeriles because it is easier to make small changes and recreate an image. Building container manually is only acceptable for proof-of-concepts or development (exceptions are container images built with e.g. Ansbile more on that later)

Sample Dockerfile

```
# our base image
FROM alpine:3.5

# Install python and pip
RUN apk add --update py2-pip
# upgrade pip
RUN pip install --upgrade pip
# install Python modules needed by the Python app
COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt
# copy files required for the app to run
COPY app.py /usr/src/app/
COPY templates/index.html /usr/src/app/templates/
# tell the port number the container should expose
EXPOSE 5000
```

Source

Dockerfile - Basics

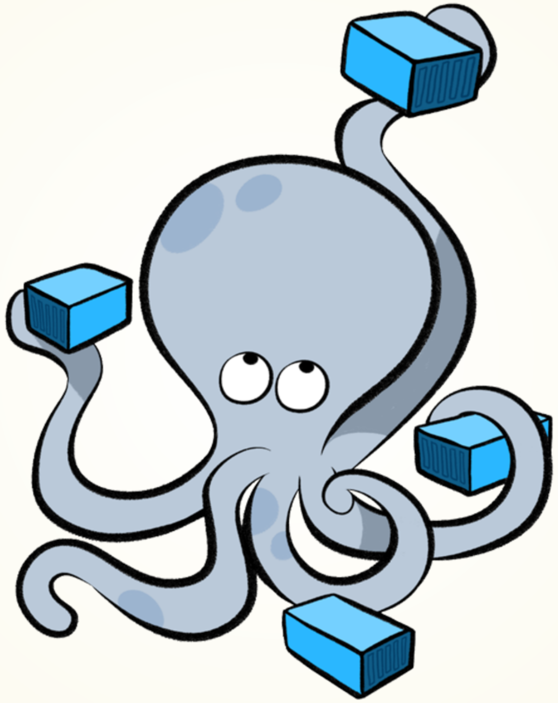
Command	Explanation
`FROM <image>[:tag]`	Declares base image which will be used
`RUN <command>`	Command to run while building the container (creates a new layer)
`CMD ["executable" [, "param1", "param2", ...]]`	Provide a default command when a new container is started
`EXPOSE <port number>`	Declare a port which will be exposed by the container (e.g. 80 for nginx or Apache web server)

Command	Explanation
`ENV <key> <value>`	Declare an environment variable for the container
`ARG <key> [<value>]`	Declare a build argument and optionally set a default value
`ADD <src> <dest>`	Copy files or directories from local or remote URLs into the container image
`COPY <src> <dest>`	Copy files or directories from local URLs into the container image
`ENTRYPOINT ["executable"[, "param1", "param2", ...]`	Declare the entrypoint of the container when it is started

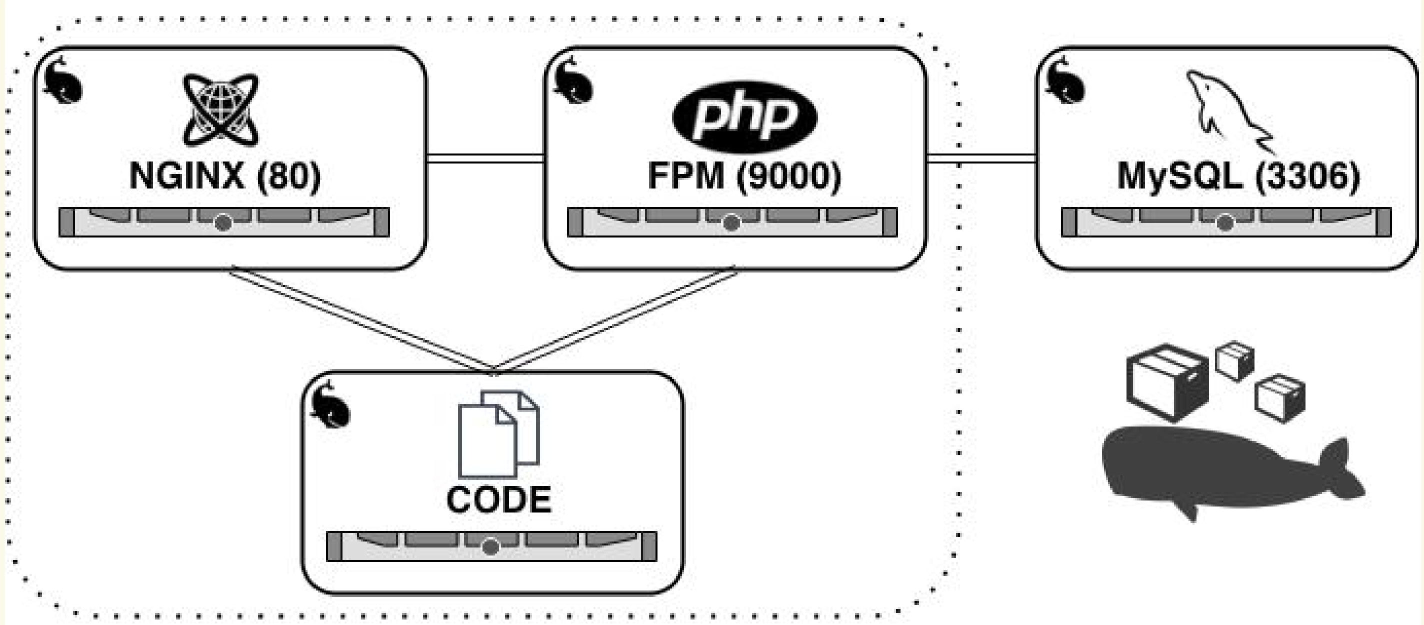
Command	Explanation
`VOLUME [ "/data" ]`	Declare a mount point to share data between the host and a container or between containers (persistence!)
`USER <user[:group]>`	Set the user context (and optionally the group) for all following `RUN`, `CMD` and the `ENTRYPOINT` in the Dockerfile
`WORKDIR /path/to/workdir`	Sets the working directory for every following `RUN`, `CMD`, `ENTRYPOINT`, `ADD` or `COPY` command, can be used multiple times in one Dockerfile, the directory will be created if it does not exist, the path can also be relative

See also Docker builder reference

docker-compose



Sample stack



Source

Concepts

- Tool to create multi-container applications
- Define all services the application consists of
- Separate services optionally in multiple networks
- Configure services (set environment variables, expose ports, mount volumes and so on)
- Start and stop a multi-container application by running a single command (`docker-compose up` or `docker-compose down`)
- An extended version is used to deploy multi-container applications to docker Swarm

- Other Docker cluster systems use similar formats (e.g. Pod definitoin in Kubernetes)
- Docs: https://docs.docker.com/compose/compose-file/compose-file-v2/
- Cheatsheet: https://devhints.io/docker-compose