

# Containerization (with Docker)

**Prof. Dr. Dirk Riehle**

**Friedrich-Alexander University Erlangen-Nürnberg**

**ADAP Z01**

Licensed under [CC BY 4.0 International](https://creativecommons.org/licenses/by/4.0/)

# Modern Tech Stacks

- Modern Applications use many different technologies, esp. service-oriented architectures
- Managing a complex tech stack on one machine is hard
- Managing a complex tech stack on different machines and environments is even harder
  - Especially if setting up the environment is done manually

	Service A (Java)	Service B (Java)	Service C (NodeJS)	Frontend (VueJS)	Database A (PostgreSQL)	Database B (MongoDB)
Laptop Dev A	Java 11	Java 11	Node 12.10.0 Current	Node 12.10.0 Current	-	MongoDB 4.2
Laptop Dev B	Java 13	Java 13	Node 12.10.0 Current	Node 12.10.0 Current	Postgres 11.5	-
QA Servers	Java 11	Java 11	Node 10.16.3 LTS	Node 10.16.3 LTS	Postgres 10.10	MongoDB 4.0
Prod Servers	Java 8	Java 8	Node 10.16.3 LTS	Node 10.16.3 LTS	Postgres 10.10	MongoDB 4.0

# Modern Tech Stacks

- Modern Applications use many different technologies and distributed architectures
- Managing a complex tech stack on one machine is hard
- Managing a complex tech stack on different machines is even harder
  - Even if the environment is done manually

Everything finally works!

Not on my machine...

Service C crashed the whole machine, so Service A and the Frontend is down, too!

Which new dependency do I have to install again?

Somehow broke the production! Hm, on my local machine it worked??!?

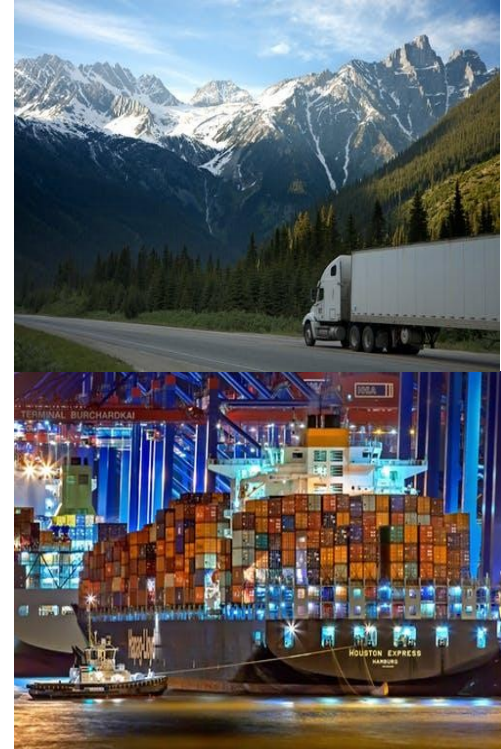
	Service A (Java)	Service B (Java)	Service C (NodeJS)	Frontend (VueJS)	Database A (PostgreSQL)	Database B (MongoDB)
Laptop Dev A	Java 11	Java 11	Node 12.10.0 Current	Node 12.10.0 Current	Postgres 10.10	MongoDB 4.2
Laptop Dev B	Java 13	Java 13	Node 12.10.0 Current	Node 12.10.0 Current	Postgres 10.10	-
QA Servers	Java 11	Java 11	Node 10.16.3 LTS	Node 10.16.3 LTS	Postgres 10.10	MongoDB 4.0
Prod Servers	Java 8	Java 8	Node 10.16.3 LTS	Node 10.16.3 LTS	Postgres 10.10	MongoDB 4.0

# The Transportation Problem

How to transport things of different sizes effectively?



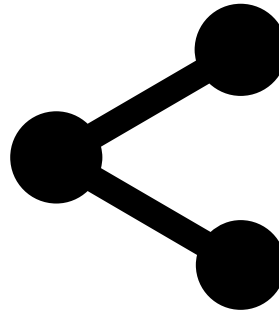
shippable containers



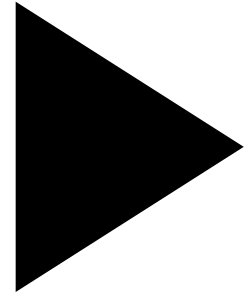
# How Docker Summarize Itself in three Words



**Build**



**Share**



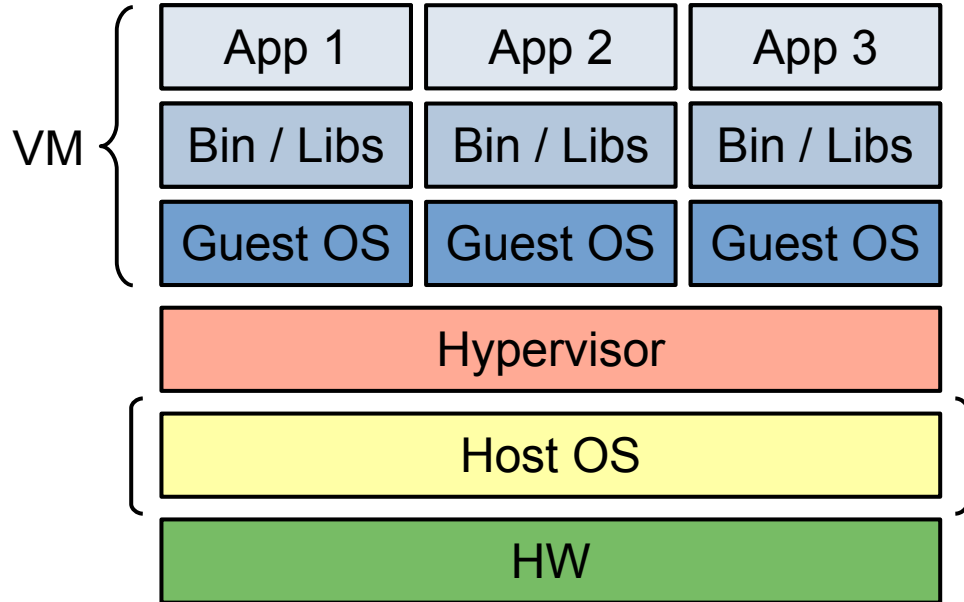
**Run**

# Advantages of Containerization

- Standardized packaging for software and its dependencies
- Consistent environment
  - Correct libraries and runtimes
  - Same environment for dev, QA and production
- Isolation / Sandboxing
  - No impact on other applications
  - Additional security layer (if configured correctly)
- Build once, run anywhere
  - Developer machine vs. on-premise vs. cloud
  - Linux vs. Windows vs. Mac
- Infrastructure as Code
  - Flexibility
  - Code = documentation

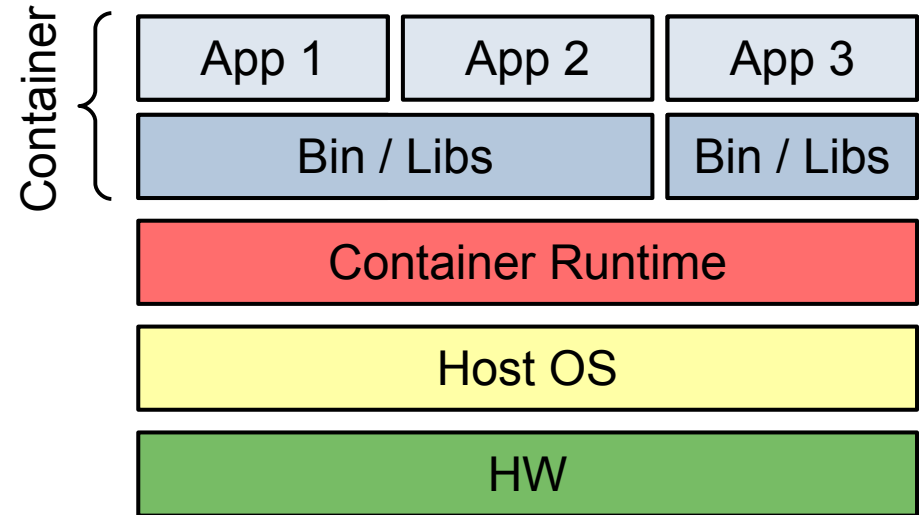
# Virtual Machines vs Containers

## Virtual Machines (VMs)



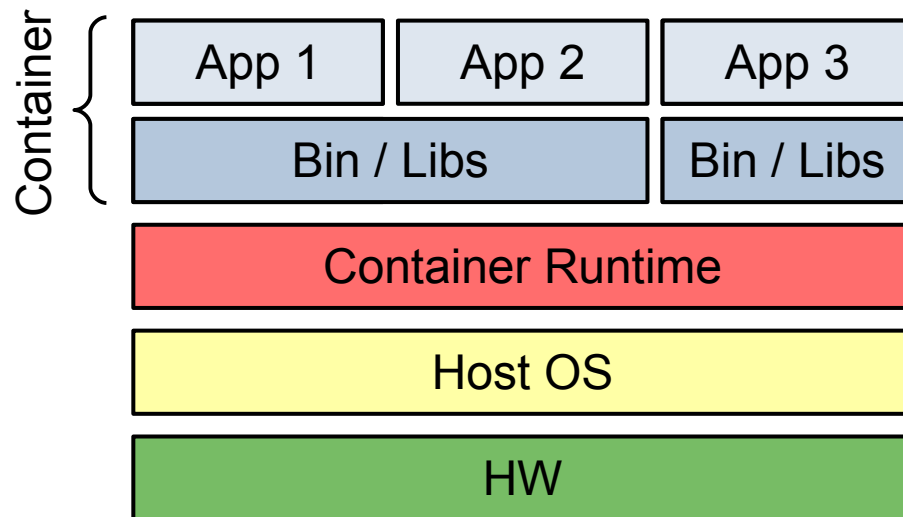
vs.

## Containers



# Virtual Machines vs Containers

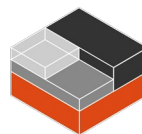
- Benefits of lightweight containers compared to VMs
  - Share host's OS with container results in
    - improved deployment speed
    - faster reboots
    - less resource overhead
      - memory and storage





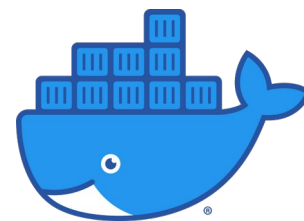
# Container Standards

- Standardization: Open Container Initiative (OCI)
  - Open governance structure to create open industry standards for containers
  - Runtime Specification
  - Image Specification
- Linux Containers (LXC)
  - OS-level virtualization technology
    - Use Linux kernel features
    - Like namespaces and control groups
  - Uses virtual environments (VE) to run isolate applications or an entire OS
- Docker is an extension of LXC with improved capabilities
  - Provides a high-level API
  - Versioning of containers
  - Container reuse (base images)
  - A public registry to share containers
  - Available as Community Edition (CE) or Enterprise Edition (EE)



## Linux Containers

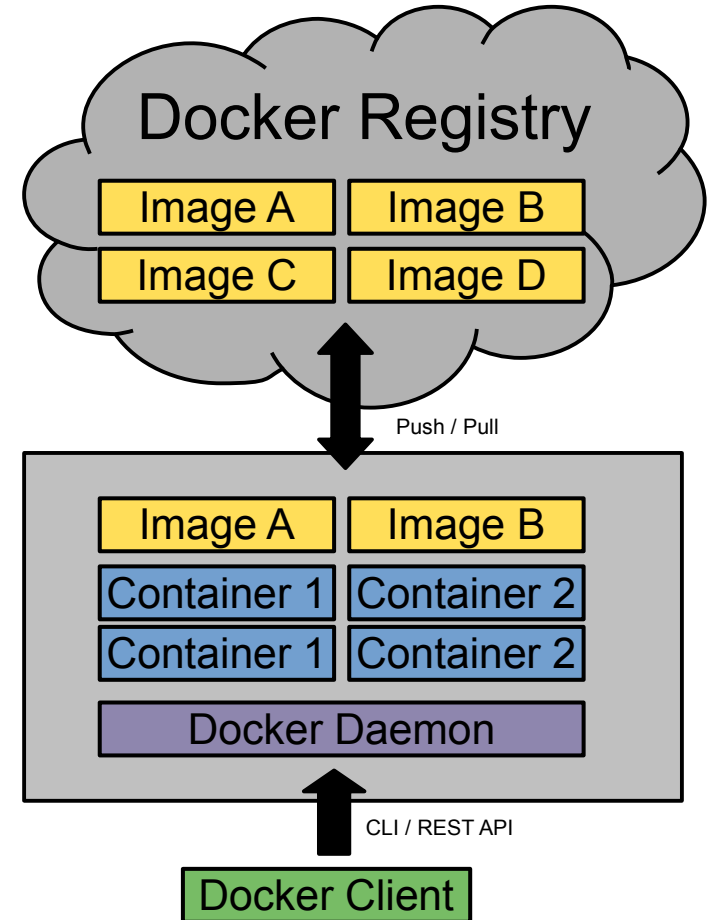
<https://discuss.linuxcontainers.org/uploads/default/original/1X/9a2865f528f7b846cda54335dec298dda6109bb3.png>



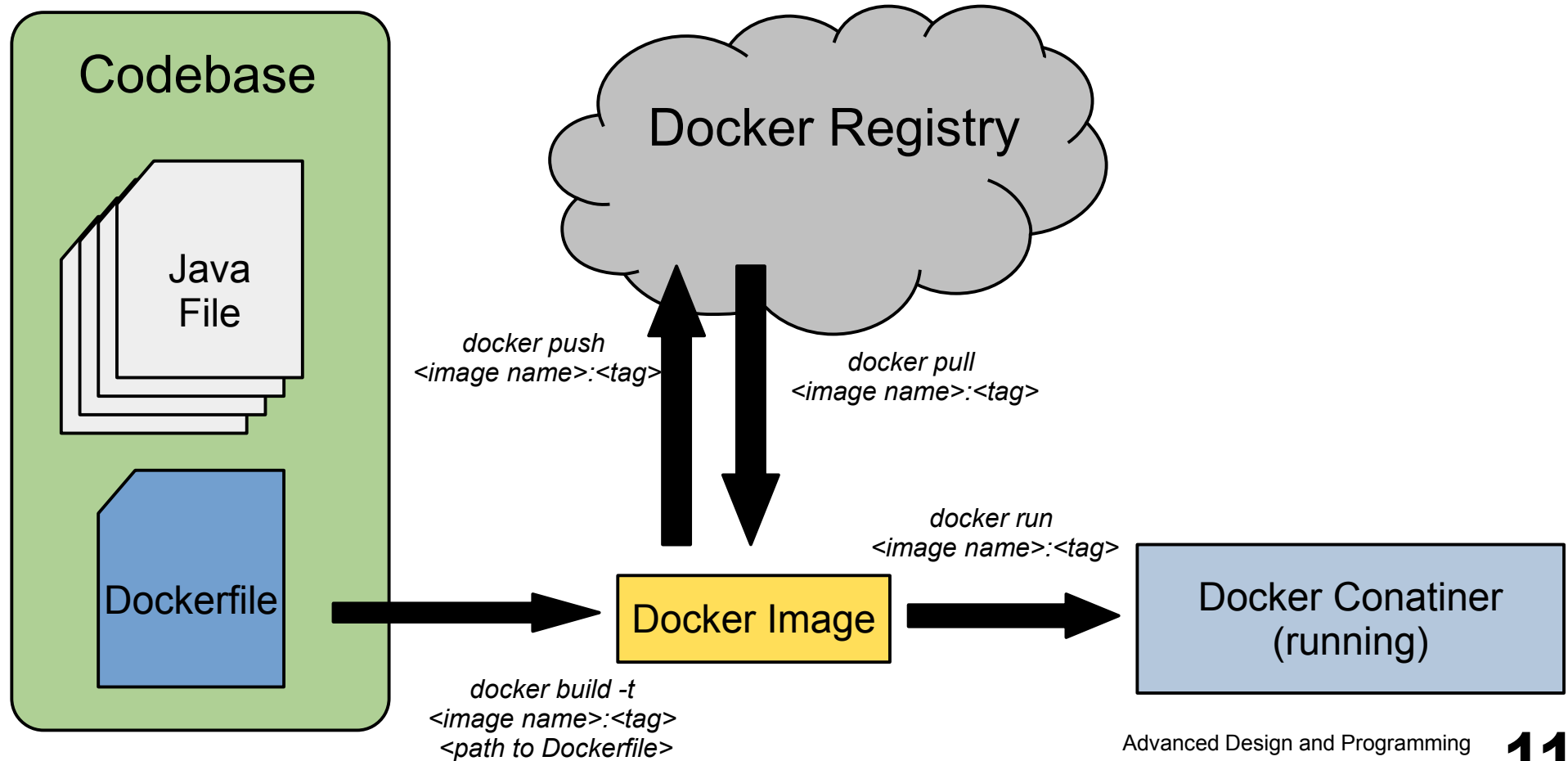
<https://www.docker.com/sites/default/files/d8/2019-07/Moby-logo.png>

# Docker Concepts

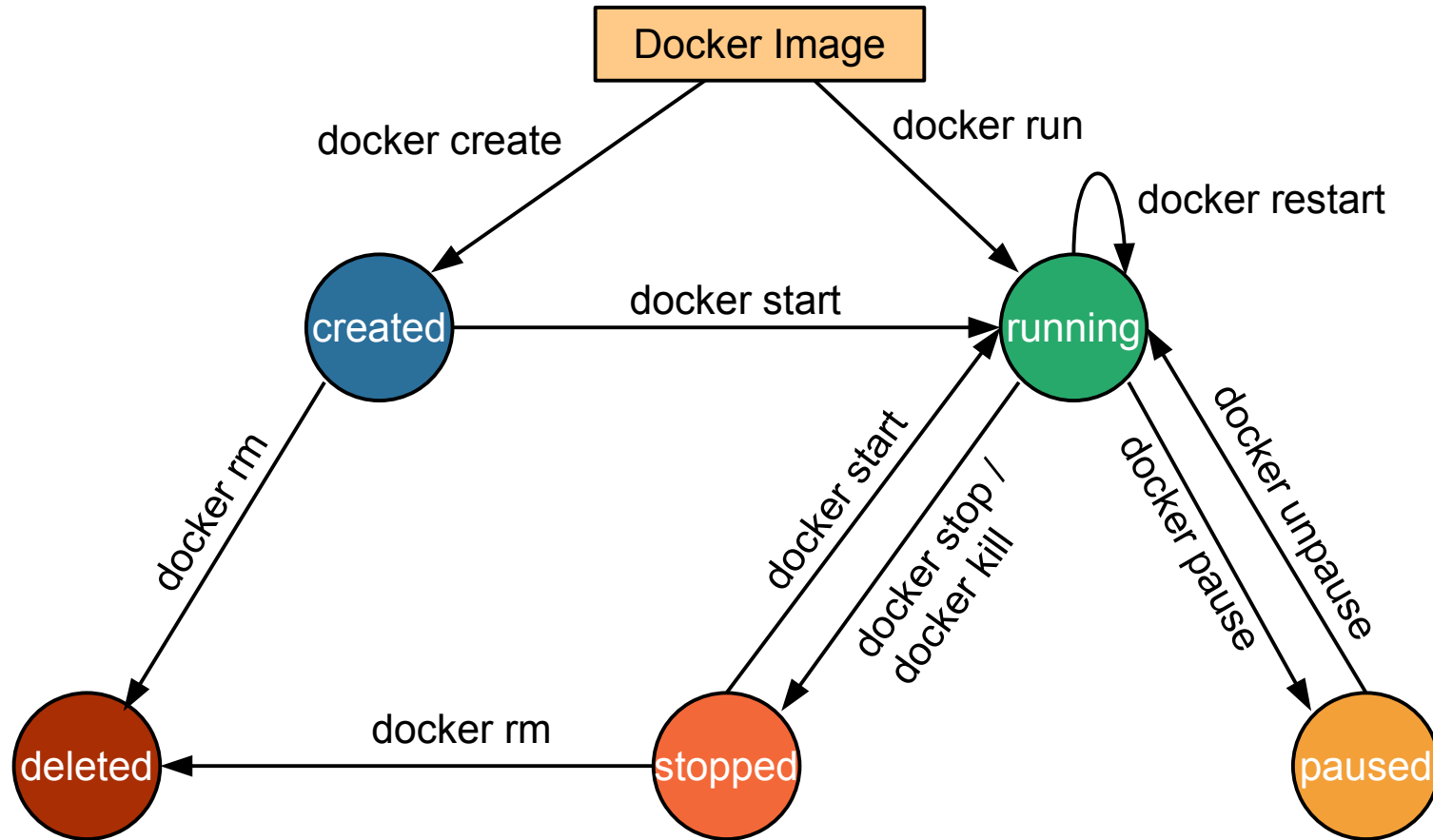
- Docker Client and Daemon
  - Docker Daemon is responsible for all actions that are related to containers
  - Receives commands from Docker Client by CLI or REST API
  - Docker Client and Daemon can be on same machine, but don't have to (client-server architecture)
- Docker Images
  - Executable package to run an application
  - Includes the code, a runtime, libraries, environment variables, and configuration files
- Docker Containers
  - Execution environment for Docker
  - Instance of an Docker Image (created from it)
- Docker Registry
  - Share Docker Images
  - Public vs. private

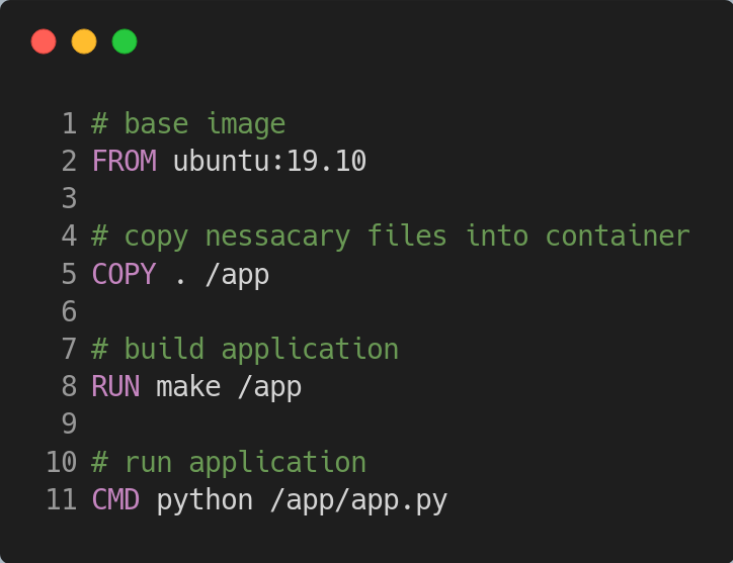


# Docker Workflow



# Docker Container Lifecycle





```
1 # base image
2 FROM ubuntu:19.10
3
4 # copy nessacary files into container
5 COPY . /app
6
7 # build application
8 RUN make /app
9
10 # run application
11 CMD python /app/app.py
```

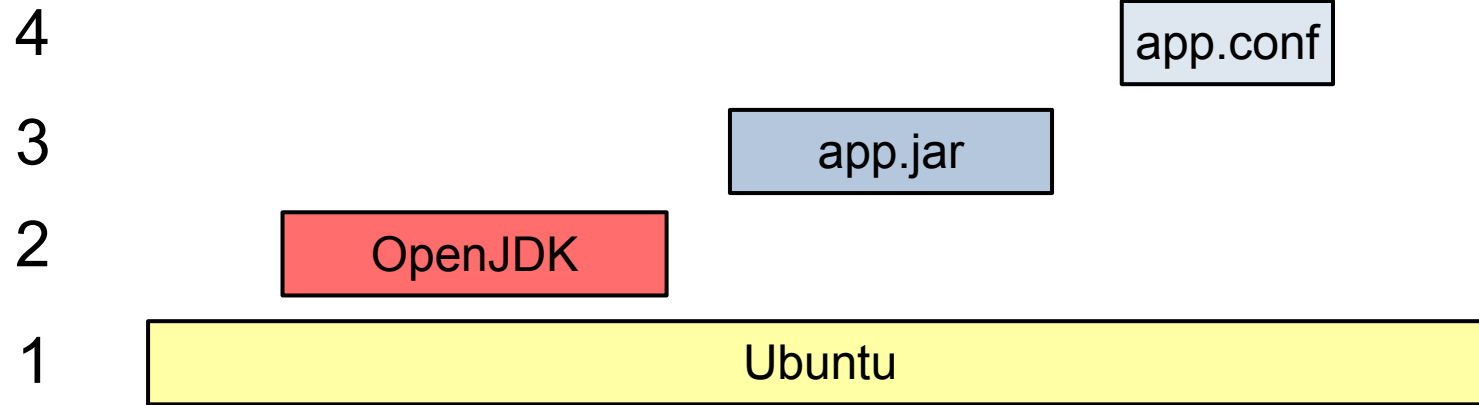
- A Dockerfile is a text document which describes how a Docker image is assembled
- A Dockerfile starts with the definition of the base image
- Each instruction creates a new layer on top of the previous image layer
- **CMD** and **ENTRYPOINT** define the command to be executed when running the image

# Docker Images

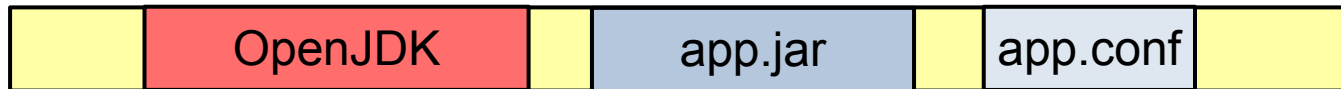
- OOP analogy: Image is like a class and a container is like an instance of a class
- Images are designed to be composed of other images
- Union File System (UFS) is the key technology used for Docker images
  - UFS allows to overlay different file systems and provide a view as a single file system on it
  - Docker allows different implementations of the UFS, e.g. AUFS, Overlay, or BTRFS
- Image layers
  - Are committed by Copy on Write (CoW)
  - Are read only
  - Can be reused to network traffic and storage consumption
  - Reuse of image layers increasing the build speed of new images
- Developers try to avoid unnecessary image layers
  - AUFS limits the amount of image layers to 127

# Docker Image Layers

## Image Layer



## Finale File System



# Docker Image Layers

```
1 # base image
2 FROM ubuntu:19.10
3
4 # copy nessacary files into container
5 COPY . /app
6
7 # build application
8 RUN make /app
9
10 # run application
11 CMD python /app/app.py
```

Container Layer  
(read and write)

6077fef09d Layer 3 0B  
\$ CMD python *app/app.py*

Image Layer  
(read only)

6077fef09d Layer 2 83 kB  
\$ RUN make /app

015f562353 Layer 1 54 kB  
\$ COPY . /app

4bcc1310a6 Layer 0 188.1 MB  
\$ FROM ubuntu:19.10



# Docker Images

- What is the difference regarding the resulting Docker image?

```
1 FROM ubuntu:19.10
2
3 RUN apt-get update
4 RUN apt-get install -y openjdk-13-jre
5
6 RUN apt-get clean
7 RUN rm -rf /var/lib/apt/lists/*
```

```
1 FROM ubuntu:19.10
2
3 RUN apt-get update \
4     && apt-get install -y openjdk-13-jre \
5     && apt-get clean \
6     && rm -rf /var/lib/apt/lists/*
```

# Docker Image Layers: Example multiple commands

- Cleaning up with an extra RUN command doesn't reduce the size of the image, it creates just a new layer with the information that the files does not exist anymore

```
1 Size:      436.39MB
2
3 ID          TAG          SIZE      COMMAND
4 8fae5631ca  adap-example:one-cmd  346.74MiB /bin/sh -c apt-get update ... && rm -rf /var/lib/apt/lists/*
5 70719f393f  ubuntu:19.10          0B        CMD ["/bin/bash"]
6 ...
7 <missing>          68.50MiB  ADD file:5bbdfa140633b135672ff0e1eb1a1b37afcab3616103c0b3d97337c62c5e2a1 in /
```


# Docker Image Layers: Example only one command

- Cleaning up within the same RUN command ensures that unnecessary files are not part of the resulting image

```
1 Size:      457.95MB
2
3 ID          TAG          SIZE      COMMAND
4 466be60494  adap:single-commands  0B        /bin/sh -c rm -rf /var/lib/apt/lists/*
5 6de502522b                0B        /bin/sh -c apt-get clean
6 8282ce4333                346.74MiB /bin/sh -c apt-get install -y openjdk-13-jre
7 9d5a7a35ec                20.55MiB  /bin/sh -c apt-get update
8 70719f393f  ubuntu:19.10          0B        CMD ["/bin/bash"]
9 ...
10 <missing>                68.50MiB  ADD file:5bbdfa140633b135672ff0e1eb1a1b37afcab3616103c0b3d97337c62c5e2a1 in /
```

# Docker Build Example

- The **docker build** command builds an image from a Dockerfile and a context




```
1 # Docker context is current/working directory and represented as dot
2
3 # build image without a name or tag
4 docker build .
5
6 # build image with a name and an optionally tag
7 docker build -t myContainer:v1.0.3 .
8
9 # use a specific Dockerfile
10 docker build -f deploy/docker/Dockerfile -t bestApp:latest .
```

# Persistence in Docker

- By default all file changes inside a container are stored on the writable container layer
  - If the container is deleted, all file changes are also gone
  - Writable container layer is tightly coupled to the host machine and can't be moved easily
  - Data depend on the lifecycle of the container
- Docker provides the following options to store data on the host system
  - Volumes (preferred option)
    - A volume is stored within a directory on the host and is managed by Docker
  - Bind mounts
    - Mounting files or directories from the host system
- Non-persistent mount with tmpfs (Linux only)
  - Store data temporary in the host memory
  - Often used to store secrets temporarily

# Persistence in Docker: Volumes


- Volumes (preferred option)
  - Created and managed by Docker
  - A volume is stored within a directory on the host
  - Easy to backup and snapshot
  - Volumes can be shared among multiple containers
  - Manage volumes using Docker CLI or API
  - Volume drivers allows to store volume on a remote host (e.g. a cloud provider)
- Example
  - Store data of MongoDB outside the container in a volume



```
1 # create a volume
2 docker volume create my-vol
3 # mount volume my-vol as "/data/db" inside the container
4 docker run --name my-db -v my-vol:/data/db mongo
```

# Persistence in Docker: Bind Mounts

- Bind mounts
  - Mount files or directories from the host system into a container
  - Limited functionality compared to volumes
  - File or directory will be created if does not exist already
  - Allows easy sharing of config files or development artifacts to container
- Example
  - Store data of MongoDB outside the container



```
1 # mount "/my/own/datadir" of the host file system
2 # as "/data/db" inside the container
3 docker run --name my-db -v /my/own/datadir:/data/db mongo
```

# Docker Environment Variables

- Environment variables allows us to configure a containerized application
  - With the **-e** parameter you can pass environment variables from the host to the container
  - With **ENV** you can define environment variables in the Dockerfile to provide default values
  - To use environment variables during image build-time you need first to introduce the variable with the **ENV** instruction
  - **ARGS** (build-time variables) can be used to pass variables during build-time
    - Running container can't access **ARG** values

```
1 docker run --name my-postgres -e POSTGRES_PASSWORD=topsecret postgres
```

```
1 # Dockerfile
2 FROM ubuntu:19.10
3 ENV DB_URL=localhost:5432
4 ...
5 CMD ./my-app.sh
```



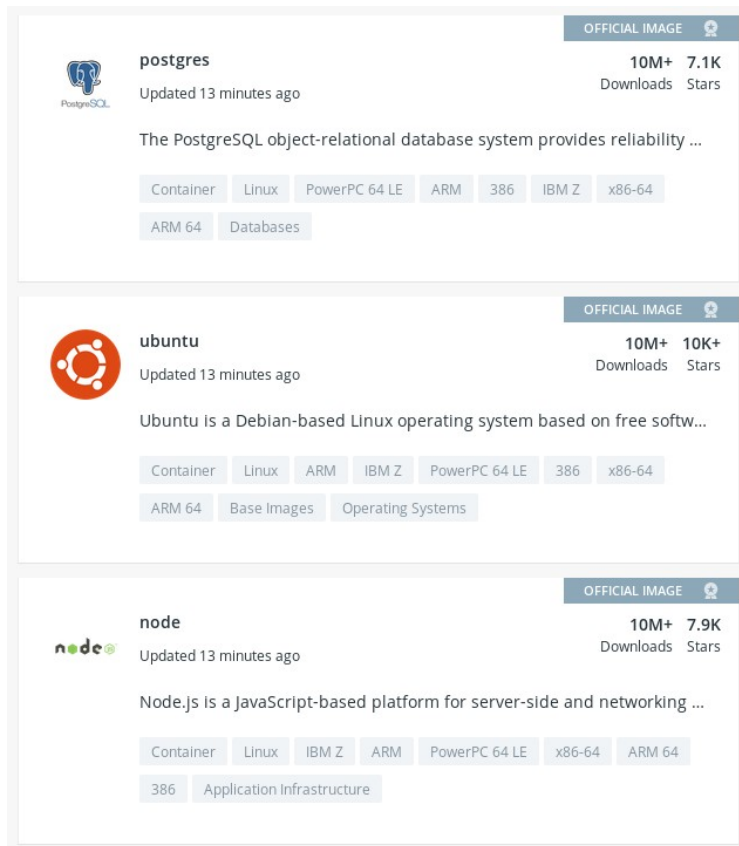
# Publish a Container's Port(s) to the Host

- Default behavior: Host can't access ports of processes inside a container
- **EXPOSE** instruction inside Dockerfile declare ports used inside a container
  - A way of documentation
- Running container with option for port mapping is required
  - Option **-P** publish all exposes ports to the host
  - Option **-p** publish one ore a range of ports to the host

```
1 # publish ports with -p [host port]:[container port]
2
3 docker run --name my-postgres -p 5432:5432 postgres-image
4
5 docker run --name some-nginx -p 8080:80 some-nginx-image
```

# Share Images with the World or a Team

## Docker Hub



The screenshot displays three Docker Hub image cards. Each card includes the image logo, name, 'Updated 13 minutes ago' status, '10M+' downloads, and '7.1K' stars. The 'postgres' card features a description and tags for Container, Linux, PowerPC 64 LE, ARM, 386, IBM Z, x86-64, ARM 64, and Databases. The 'ubuntu' card includes a description and tags for Container, Linux, ARM, IBM Z, PowerPC 64 LE, 386, x86-64, ARM 64, Base Images, and Operating Systems. The 'node' card includes a description and tags for Container, Linux, IBM Z, ARM, PowerPC 64 LE, x86-64, ARM 64, 386, and Application Infrastructure.

**postgres** OFFICIAL IMAGE  
Updated 13 minutes ago **10M+** **7.1K**  
Downloads Stars  
The PostgreSQL object-relational database system provides reliability ...  
Container Linux PowerPC 64 LE ARM 386 IBM Z x86-64  
ARM 64 Databases

**ubuntu** OFFICIAL IMAGE  
Updated 13 minutes ago **10M+** **10K+**  
Downloads Stars  
Ubuntu is a Debian-based Linux operating system based on free softw...  
Container Linux ARM IBM Z PowerPC 64 LE 386 x86-64  
ARM 64 Base Images Operating Systems

**node** OFFICIAL IMAGE  
Updated 13 minutes ago **10M+** **7.9K**  
Downloads Stars  
Node.js is a JavaScript-based platform for server-side and networking ...  
Container Linux IBM Z ARM PowerPC 64 LE x86-64 ARM 64  
386 Application Infrastructure

- Public and private image repositories allow to easy share and manage images
- Docker Hub is the default repository for Docker

# Docker Environment on different OS

- Linux
  - Containers are part of the Linux ecosystem
  - Native isolated processes
  - Direct access to the host Linux kernel
  - Best performance
- Mac (Docker for Mac)
  - Lightweight virtual machine (LinuxKit) running on macOS Hypervisor
  - No direct access to the network stack and native file mounts
  - Uses Unix sockets to bridge host and Docker VM
- Windows
  - Runs LinuxKit VM with the virtualization tool Hyper-V
  - Similar to Docker for Mac
- Docker Toolbox (old solution for Mac and Windows)
  - Runs a Linux VM using Virtual Box
  - Bridges host and Docker VM via TCP connection
  - Lower performance

# Why Container Orchestration

- Modern applications consist of smaller (micro)services that work together
- This way a containerized application consists of multiple container that communicate over network to provide a service
- Managing containers becomes a complex task
  - E.g. deploying the whole application requires custom scripts
- The process of organizing multiple container is called **container orchestration** [1]
- Container orchestration solutions
  - **Docker Swarm** by Docker Inc
  - **Kubernetes** based on Google's work (the gold standard)
  - **Mesos** by Apache

[1] <https://www.hpe.com/de/de/what-is/container-orchestration.html>



- *“Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications.” [2]*
- Horizontal scaling (for medium to large clusters)
  - Spawn new instances of services either by hand, or
  - Automatically based on metrics like CPU usage
  - Optimized usage of a cluster's resources
- Service discovery and load-balancing
- Self-healing features
  - like restart on failure with health monitoring
- Manage deployments
  - Define deployment units
  - Automated rollouts and rollbacks

# Thank you! Questions?

**[dirk.riehle@fau.de](mailto:dirk.riehle@fau.de) – <http://osr.cs.fau.de>**

**[dirk@riehle.org](mailto:dirk@riehle.org) – <http://dirkriehle.com> – [@dirkriehle](#)**

- Original version
  - © Friedrich-Alexander University Erlangen-Nürnberg, all rights reserved
- Contributions
  - Andreas Bauer (2019)
  - Georg Schwarz (2019)