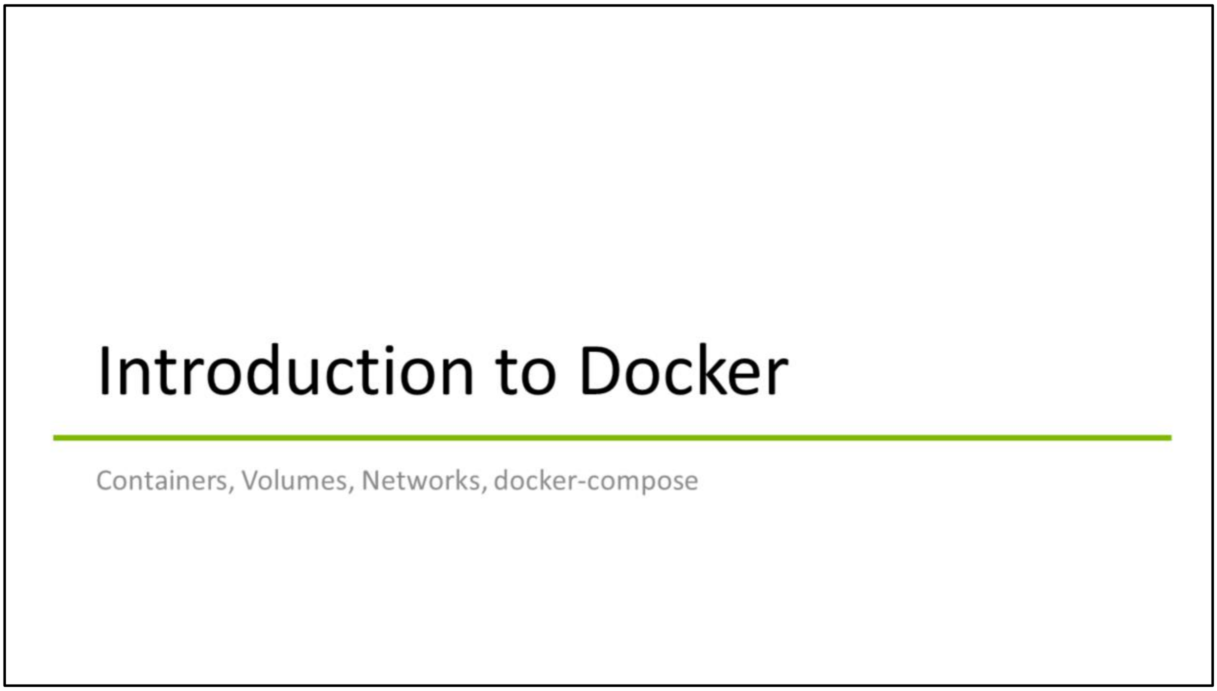
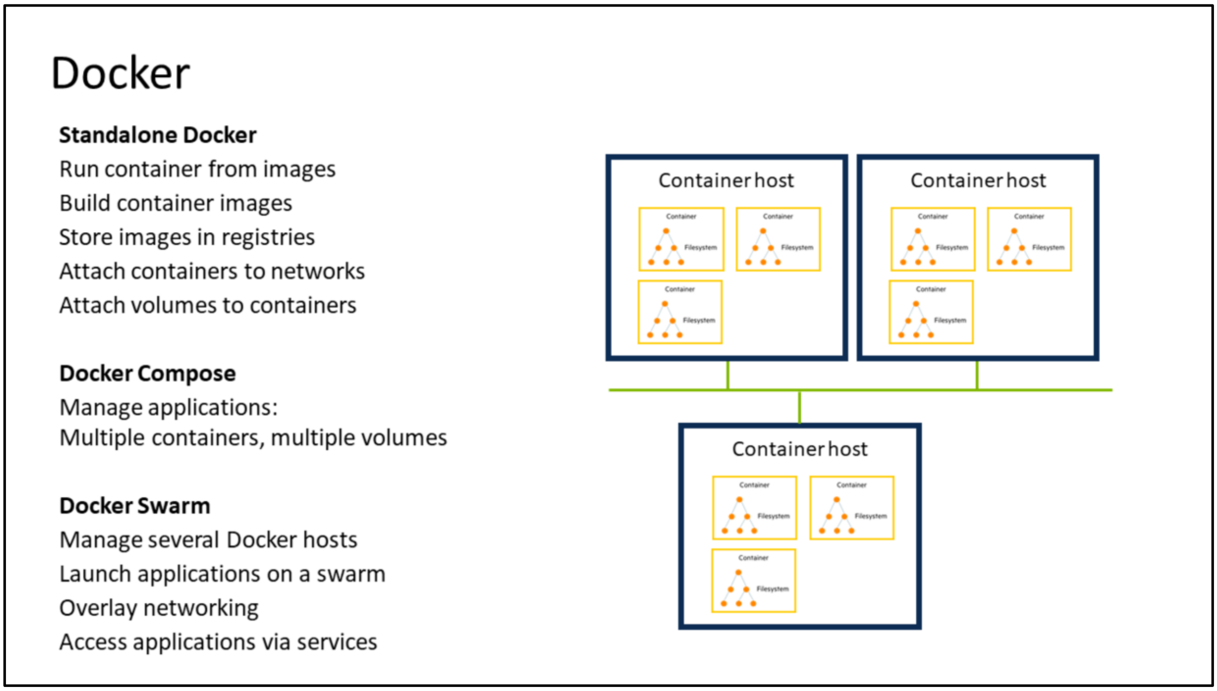
Module 2 Containerizing Applications

Docker, Podman, Buildah, Skopeo



Introduction to Docker

Containers, Volumes, Networks, docker-compose



Docker

Standalone Docker

Run container from images

Build container images

Store images in registries

Attach containers to networks

Attach volumes to containers

Docker Compose

Manage applications:
Multiple containers, multiple volumes

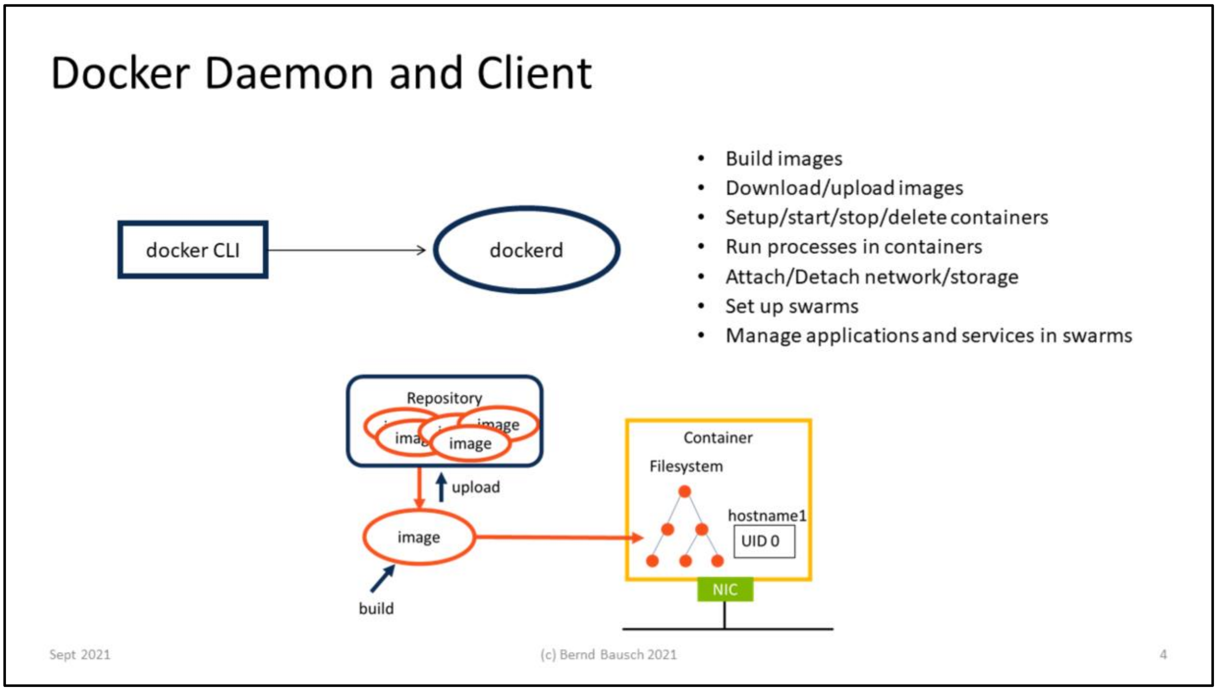
Docker Swarm

Manage several Docker hosts

Launch applications on a swarm

Overlay networking

Access applications via services



Docker Daemon and Client

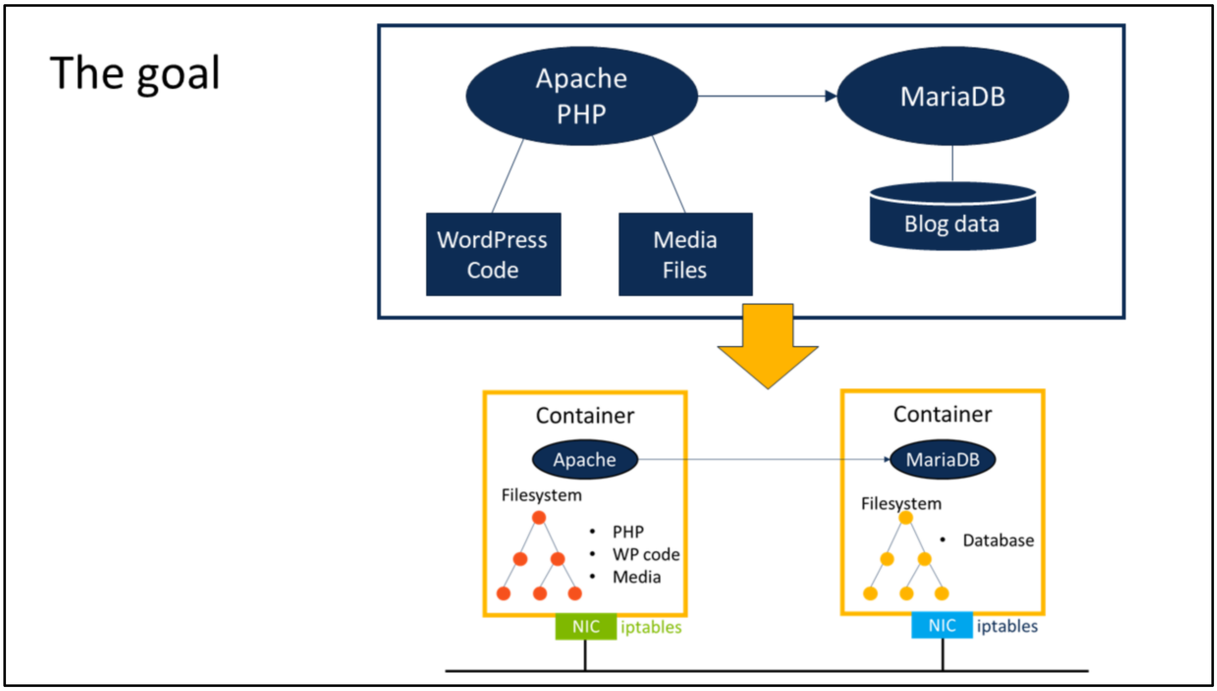
- Build images
- Download/upload images
- Setup/start/stop/delete containers
- Run processes in containers
- Attach/Detach network/storage
- Set up swarms
- Manage applications and services in swarms

Sept 2021 (c) Bernd Bausch 2021 4

Docker uses the **Docker daemon** to perform the work. It runs as a service in the background of the Docker container host and performs all the tasks listed on the previous slide.

Communication with the Docker daemon goes via a RESTful API. Interactive users have the docker CLI, which issues the API request, and developers can use the Go or Python SDK to write applications that talk to the daemon.

Traditionally, the Docker daemon needs to run as root, and the Docker CLI can only communicate with it when it runs as root as well. Rootless Docker is now possible but won't be covered in this course.



The goal

Apache PHP

MariaDB

WordPress Code

Media Files

Blog data

Container

Apache

Filesystem

- PHP
- WP code
- Media

NIC iptables

Container

MariaDB

Filesystem

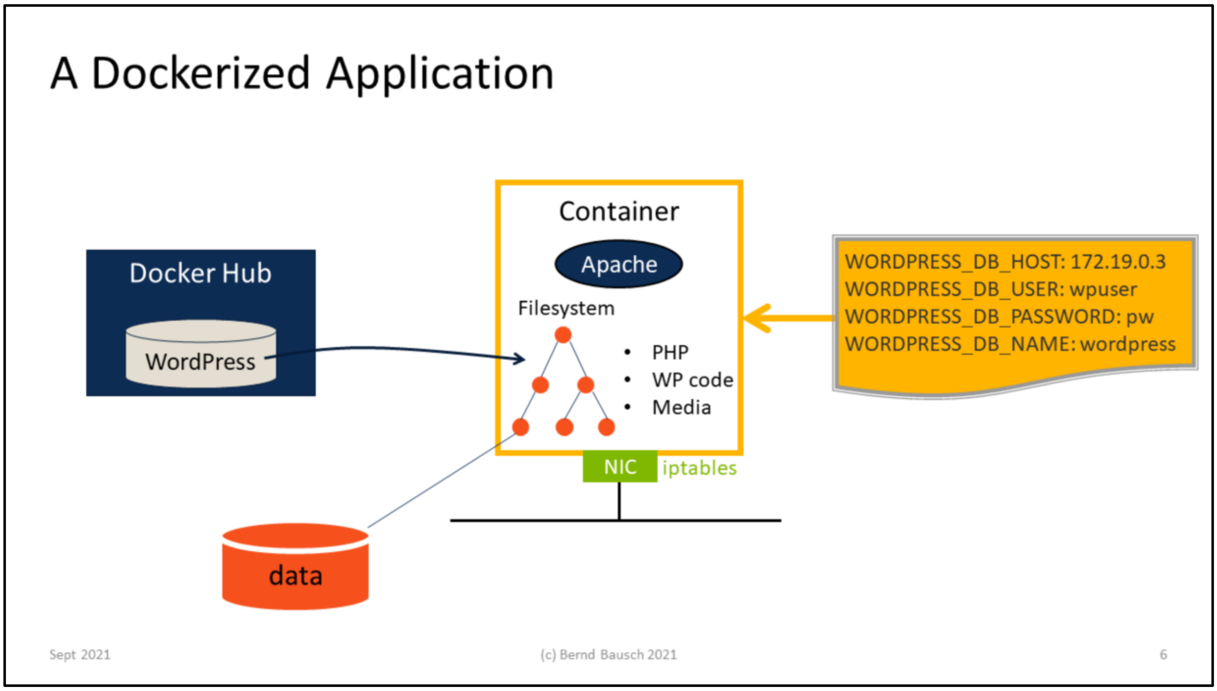
- Database

NIC iptables

How do we convert a traditional application, such as the WordPress website depicted here, to a container solution? In this example, the application is turned into two containers, one for the webserver including the WordPress PHP code, and the other for the database.

The database contains the web site's content, which is valuable and must not be lost. We want to keep it at a location that is safe and available.

The WordPress PHP code in the web server container doesn't change during the life of a site (except for updates) and doesn't require particularly secure storage, but WordPress keeps media files in a directory on the web site, not in the database. This media folder should be treated with the same caution as the database.



A Dockerized Application

Docker Hub

WordPress

Container

Apache

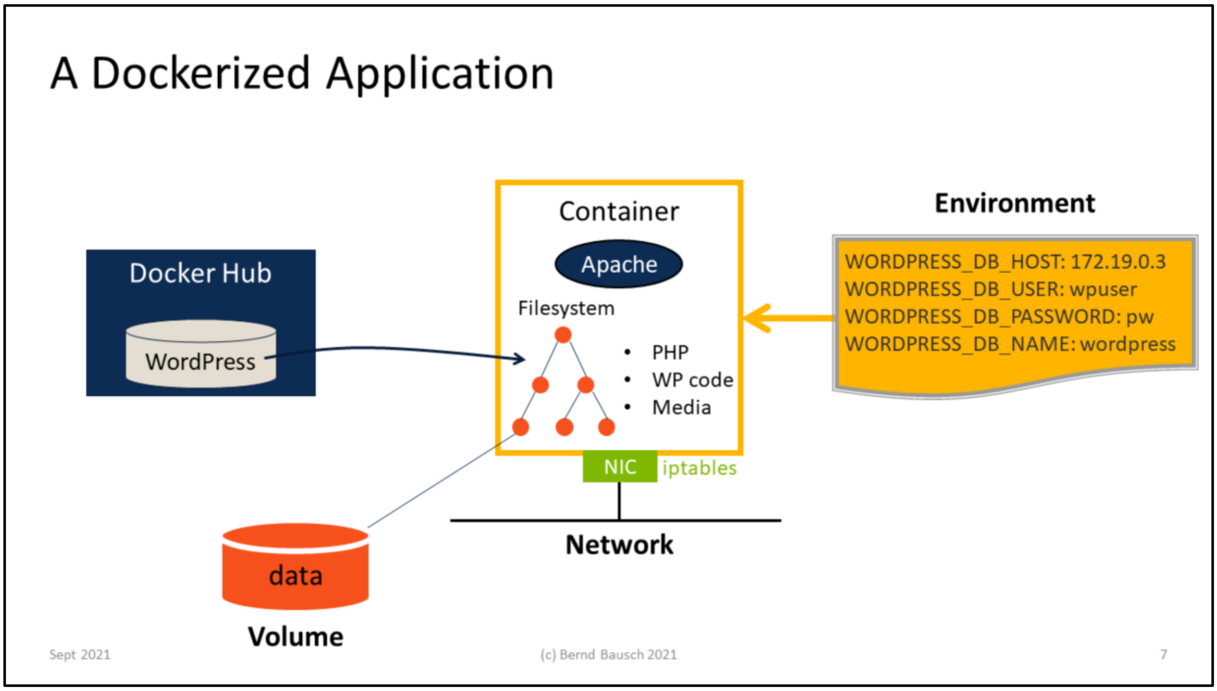
Filesystem

- PHP
- WP code
- Media

NIC iptables

data

```
WORDPRESS_DB_HOST: 172.19.0.3
WORDPRESS_DB_USER: wpuser
WORDPRESS_DB_PASSWORD: pw
WORDPRESS_DB_NAME: wordpress
```



A Dockerized Application

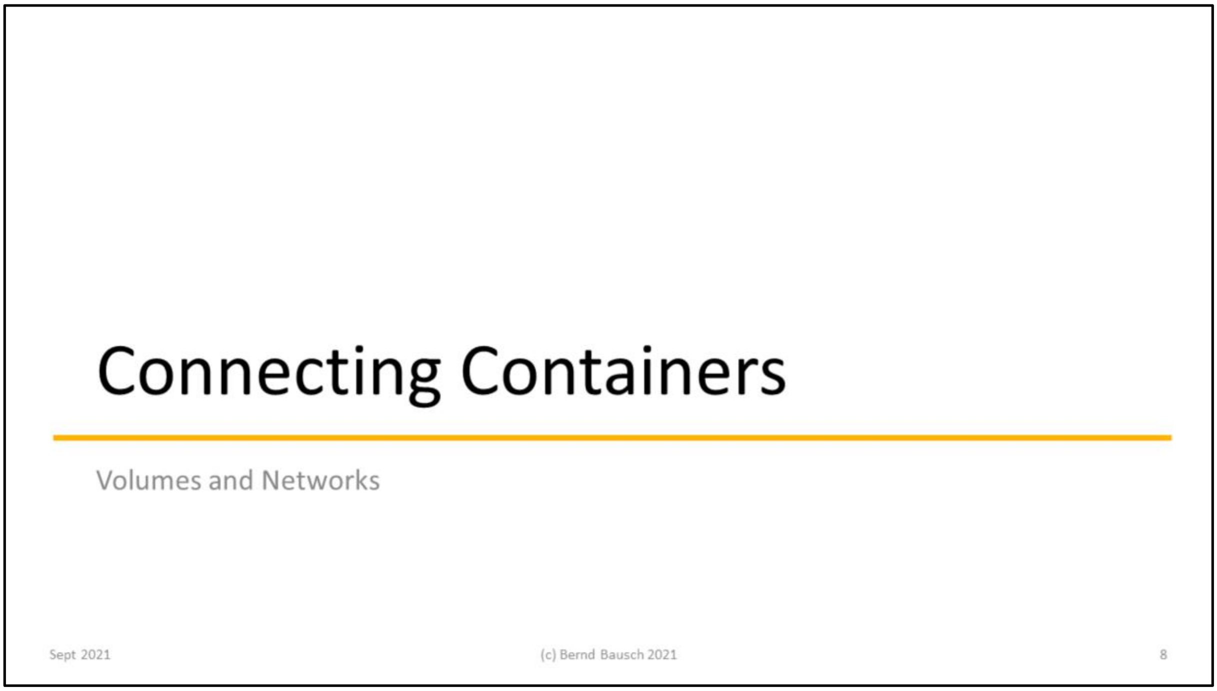
WORDPRESS_DB_HOST: 172.19.0.3
WORDPRESS_DB_USER: wpuser
WORDPRESS_DB_PASSWORD: pw
WORDPRESS_DB_NAME: wordpress

Sept 2021 (c) Bernd Bausch 2021 7

It is not sufficient to just create a container to run the application in. It must be connected to a network, and may require external storage. The latter is named **volume** in Docker terminology. A volume is mounted by a container, but its lifecycle is separate from the container's.

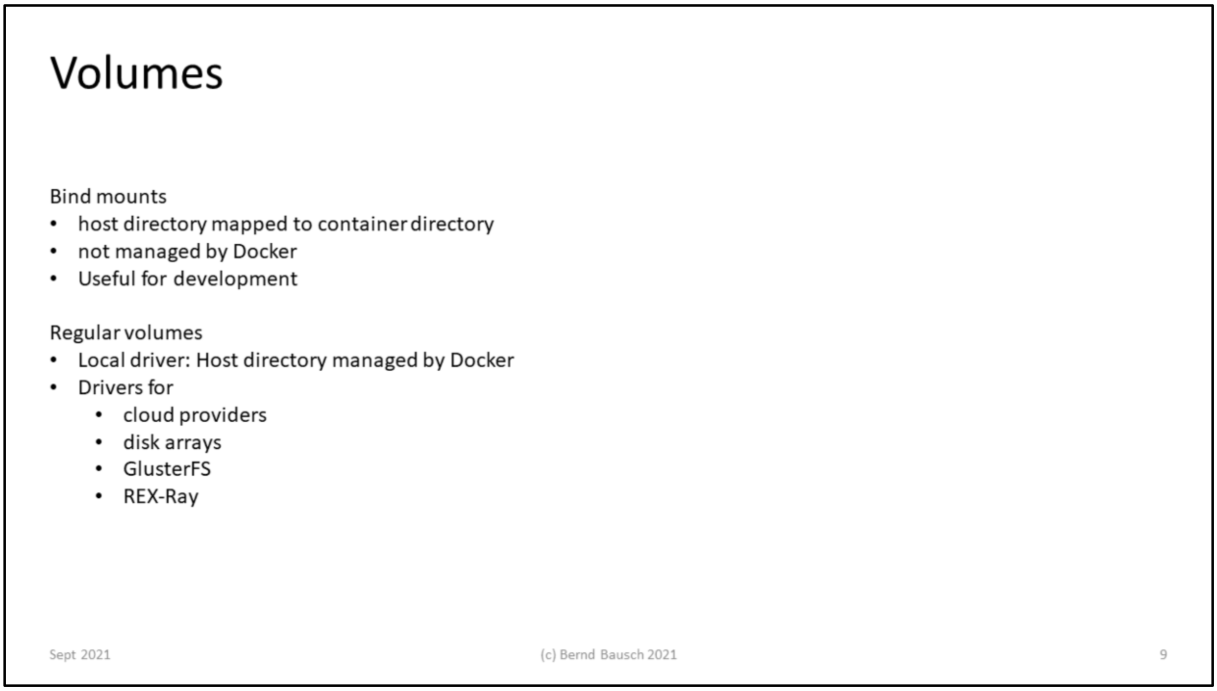
A typical method of configuring Docker containers is via **environment variables**. For example, in the case of WordPress the environment is used to configure the name and access information of the database. Of course, the WordPress container's environment must be consistent with the Database container's environment.

The most convenient way of setting up a dockerized WordPress web site is from the official **wordpress image**, which can be obtained at the Docker Hub.



Connecting Containers

Volumes and Networks



Volumes

Bind mounts
- host directory mapped to container directory
- not managed by Docker
- Useful for development

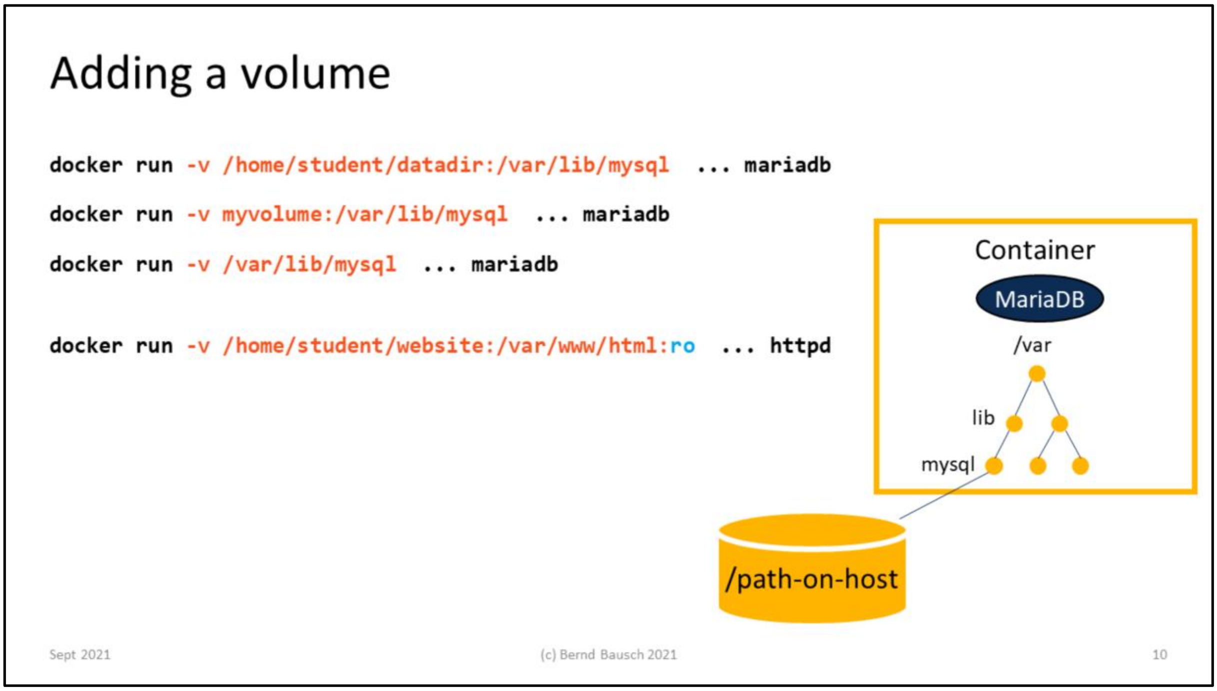
Regular volumes
- Local driver: Host directory managed by Docker
- Drivers for
  - cloud providers
  - disk arrays
  - GlusterFS
  - REX-Ray

Sept 2021 (c) Bernd Bausch 2021 9

External storage is often implemented as a simple directory on the container's host. It can be managed by the user and left alone by Docker, which case is named a **bind mount**. This term comes from the bind feature of the Linux mount system call, which permits mounting existing directories on a different directory (the mount point) in the filesystem.

A volume can also be implemented by a host directory that is managed by Docker.

It is often desirable to centralize volume storage on disk arrays or large-scale storage solutions like GlusterFS. Via its plugin framework, Docker allows the usage of remote storage of various types. When volumes are located on such storage systems, they can be shared between containers that run on different hosts.



Adding a volume

```
docker run -v /home/student/datadir:/var/lib/mysql  ... mariadb
docker run -v myvolume:/var/lib/mysql  ... mariadb
docker run -v /var/lib/mysql  ... mariadb

docker run -v /home/student/website:/var/www/html:ro  ... httpd
```

Sept 2021 (c) Bernd Bausch 2021 10

A volume is a directory outside of the container that is connected to the container's filesystem. A volume is not deleted when the container is deleted.

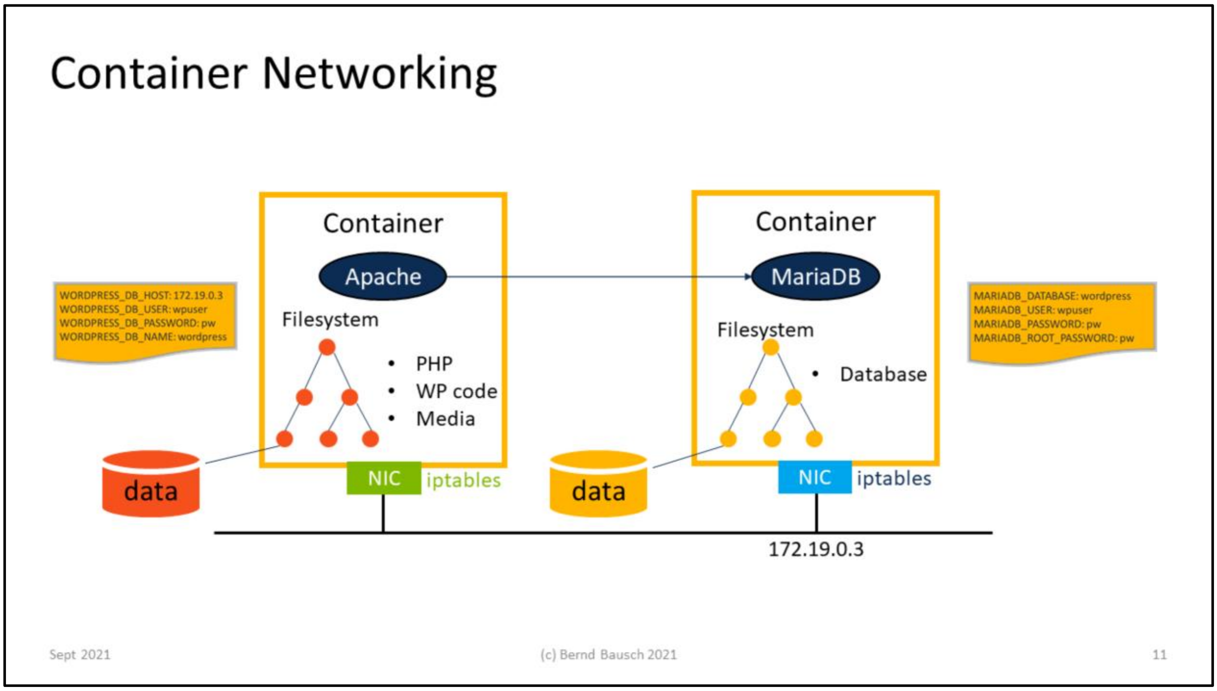
The first command mounts /home/student/datadir to the MariaDB's /var/lib/mysql directory. This means that the database will reside on the student's datadir. When the user specifies the directory that will be mounted, we talk about a **bind mount**. It is not managed by Docker.

The next command mounts a volume named myvolume. If it was not created by the user beforehand, it will be created now. When you create a volume using the docker volume create command, you can specify a driver that takes the volume from a storage provider such as a disk array, NFS, Ceph, GlusterFS etc.

The third command asks Docker to create an anonymous volume, which is normally a directory under /var/lib/docker, and mount it.

The final command shows that volumes can be mounted read-only. Other mount flags are possible.

An alternative to the -v option is --mount. It allows finer-grained control over the mount parameters.



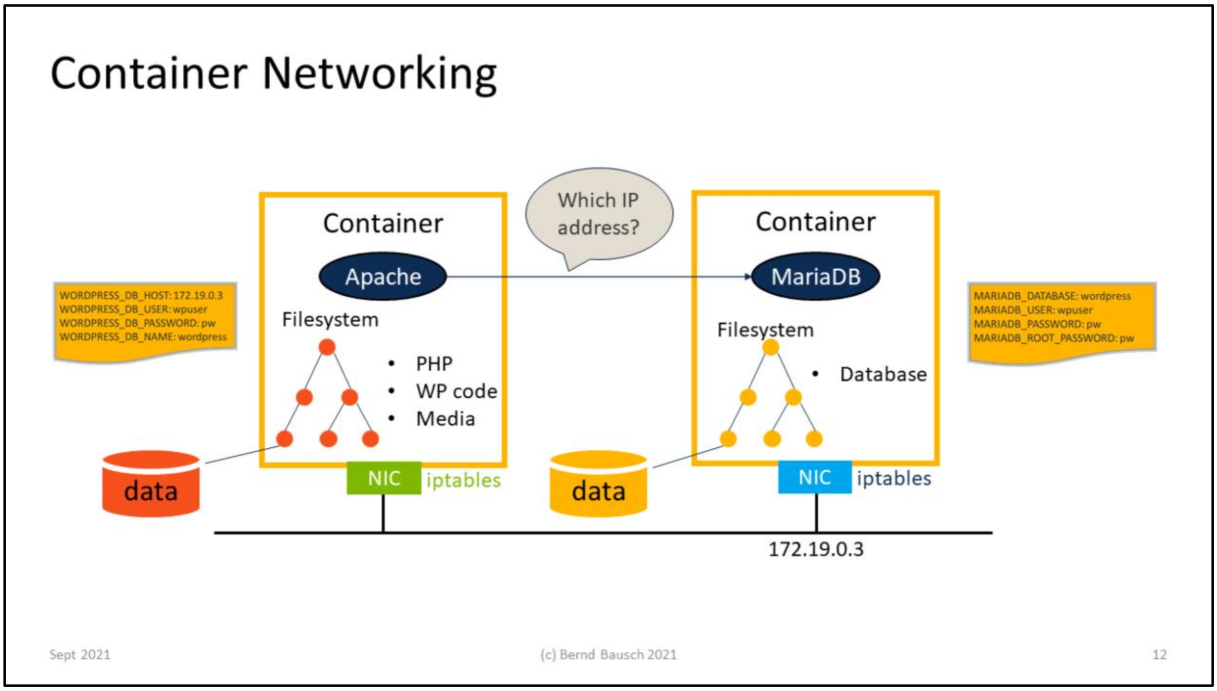
Container Networking

```
WORDPRESS_DB_HOST: 172.19.0.3
WORDPRESS_DB_USER: wpuser
WORDPRESS_DB_PASSWORD: pw
WORDPRESS_DB_NAME: wordpress
```

```
MARIADB_DATABASE: wordpress
MARIADB_USER: wpuser
MARIADB_PASSWORD: pw
MARIADB_ROOT_PASSWORD: pw
```

Sept 2021 (c) Bernd Bausch 2021 11

This is a more complete picture of the WordPress application. The two containers have their own environment, which identifies the database and determines its user name and password.



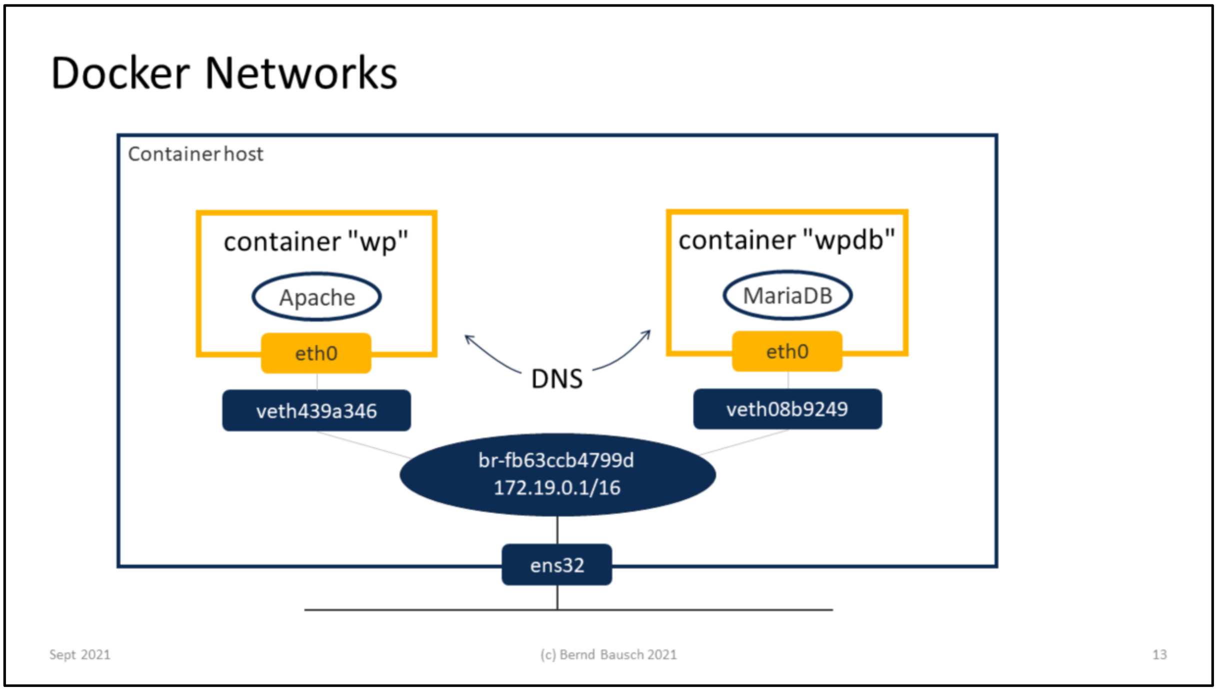
Container Networking

```
WORDPRESS_DB_HOST: 172.19.0.3
WORDPRESS_DB_USER: wpuser
WORDPRESS_DB_PASSWORD: pw
WORDPRESS_DB_NAME: wordpress
```

```
MARIADB_DATABASE: wordpress
MARIADB_USER: wpuser
MARIADB_PASSWORD: pw
MARIADB_ROOT_PASSWORD: pw
```

However, there is a problem. The wordpress container on the left needs to know the IP address of the database container on the right. The IP address is determined by Docker when a container is launched, so that we don't know it beforehand.

We could start the database container, get it's IP address, then start the wordpress container, setting its WORDPRESS_DB_HOST variable to the IP address. This would be a lot of manual work and is not a very elegant solution. Is there a better way to provide connection information to the wordpress container?

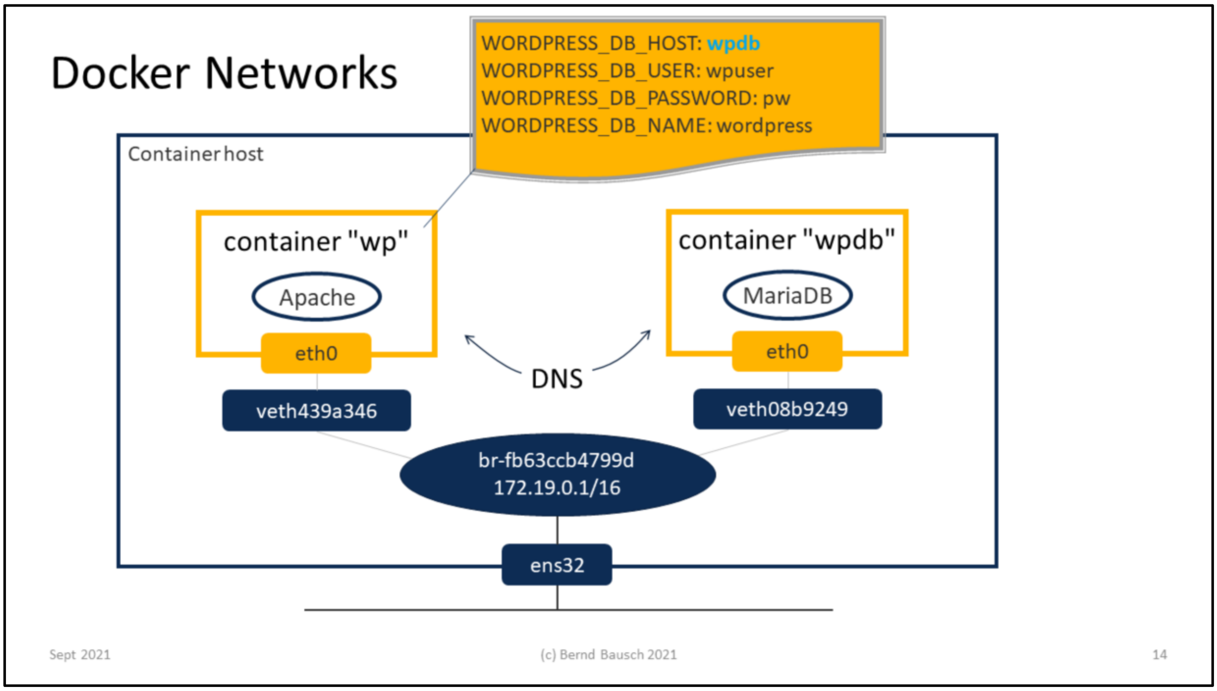


Docker Networks

Sept 2021 (c) Bernd Bausch 2021 13

Containers that cooperate over the network can be attached to their own, private network. This has the advantage that no other containers can see the network traffic. Furthermore, Docker runs a DNS server that allows the containers to refer to each other by name instead of IP address. For example, WordPress configuration in the *wp* container can use *wpdb* as the database host.

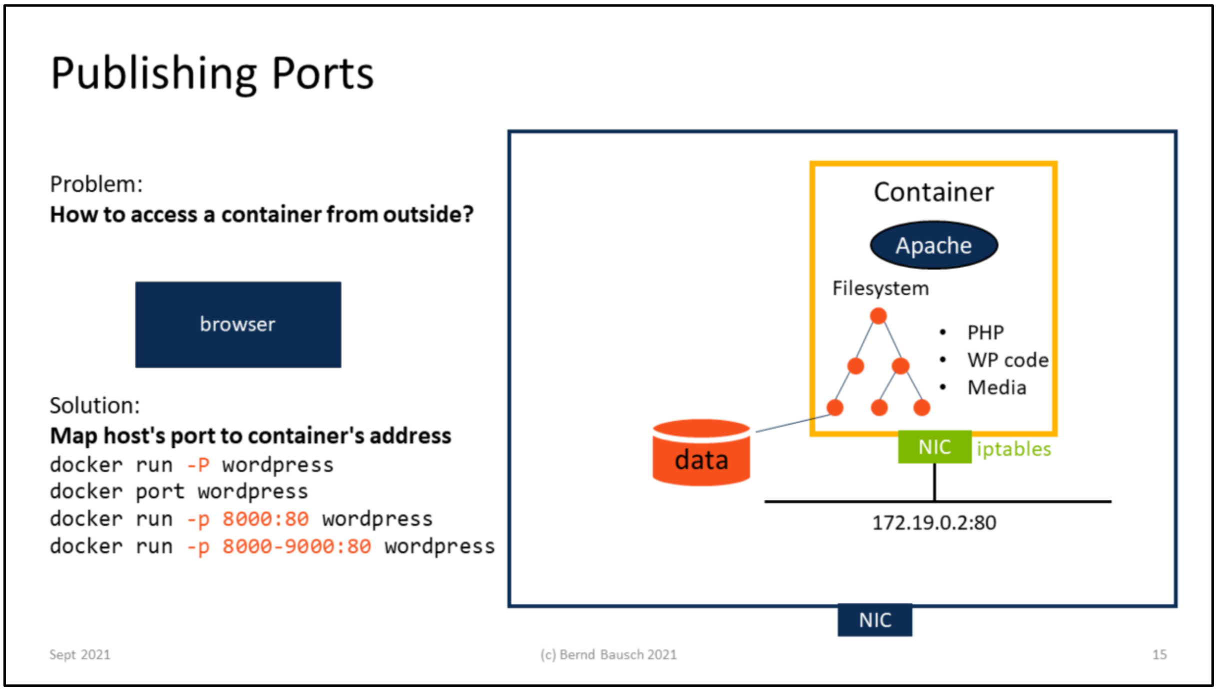
By default, a network is implemented by a Linuxbridge named after the network's Docker ID. Containers are connected to the bridge via so-called **veth pairs**. A veth pair is a set of two network interfaces that are connected via a "patch cable". Any packets that travel through one interface of the veth pair also travel through the other.
Each veth pair connects one container. One end of the veth pair is in the container's network namespace, the other is in the host's namespace and is plugged into the bridge.



Docker Networks

WORDPRESS_DB_HOST: wpdb
WORDPRESS_DB_USER: wpuser
WORDPRESS_DB_PASSWORD: pw
WORDPRESS_DB_NAME: wordpress

Sept 2021 (c) Bernd Bausch 2021 14



Publishing Ports

Problem:
How to access a container from outside?

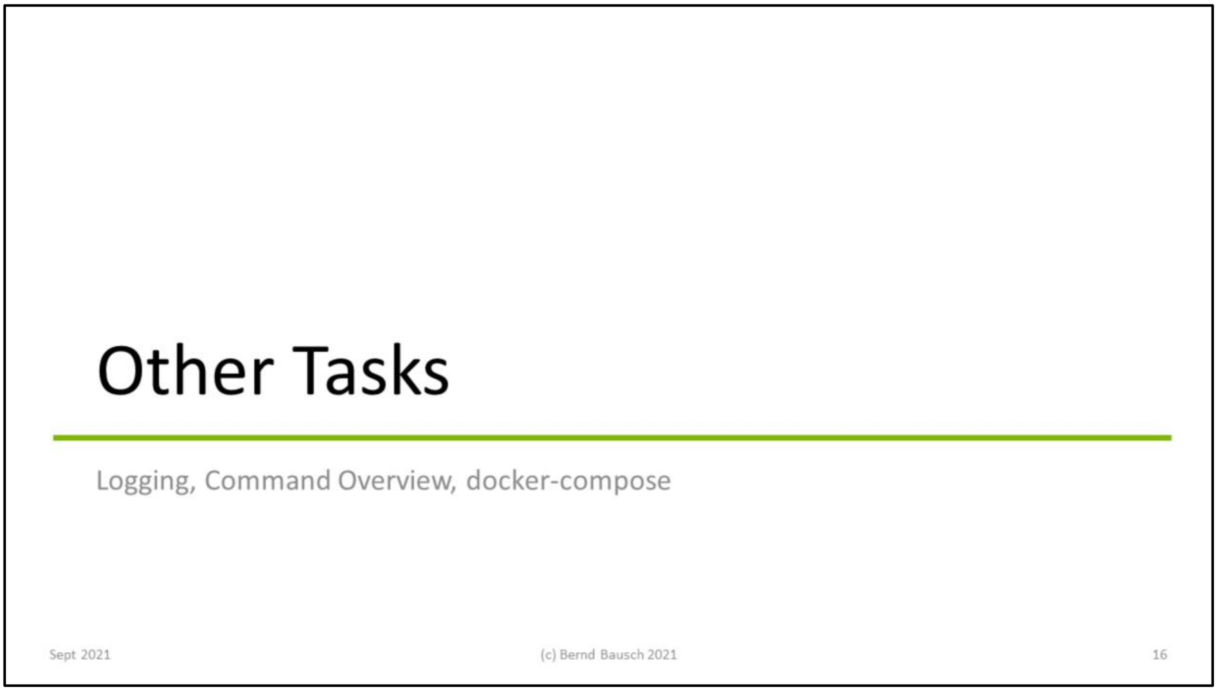
Solution:
Map host's port to container's address

```
docker run -P wordpress
docker port wordpress
docker run -p 8000:80 wordpress
docker run -p 8000-9000:80 wordpress
```

Sept 2021 (c) Bernd Bausch 2021 15

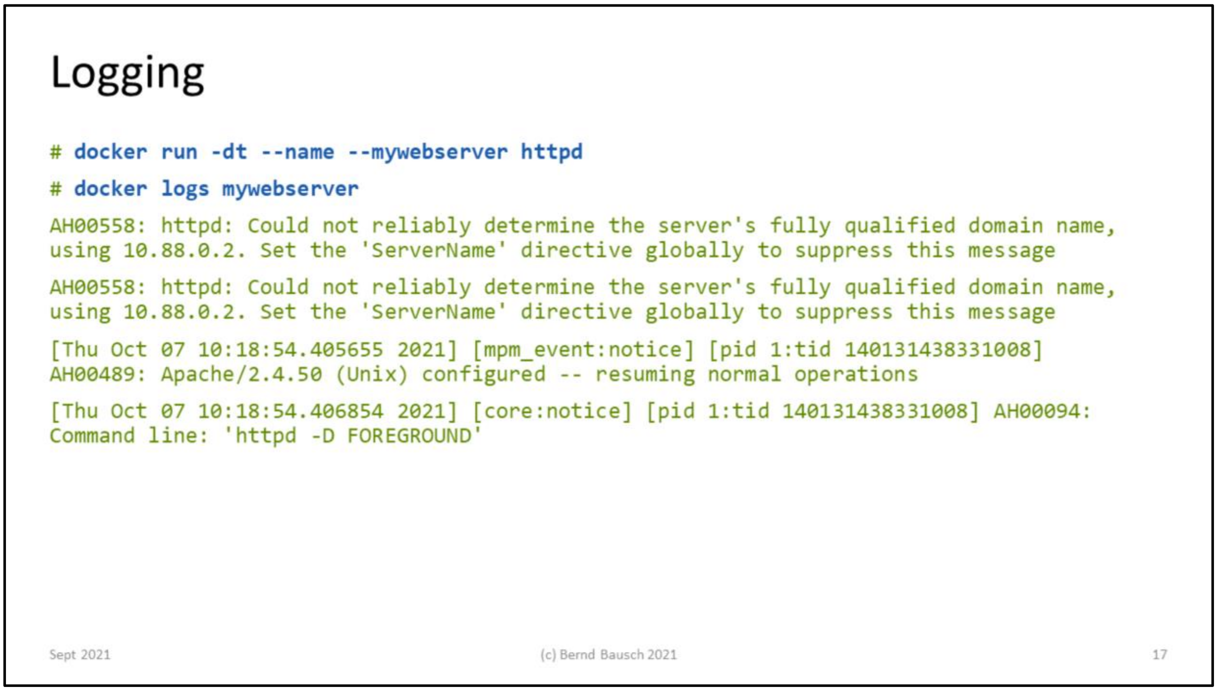
Not all containers need to be accessed from outside, but some of them are "customer-facing". Since Docker associates a non-routable IP address with the container, it can’t be reached from outside directly.

The solution is port-mapping. You can map the port on which the containerized application listens to a port on the host. Docker runs a `docker-proxy` process that performs the connection. This is called **publishing** the port. You can ask Docker to select a random port on the host (upper-case -P option) and use docker port to check which port was chosen. You can also specify a port (lower-case -p), or a port *range* from which Docker will select the port; this is useful, for example, for launching several identical containers using the same command. Finally, the -p option can be repeated to generate several mappings.



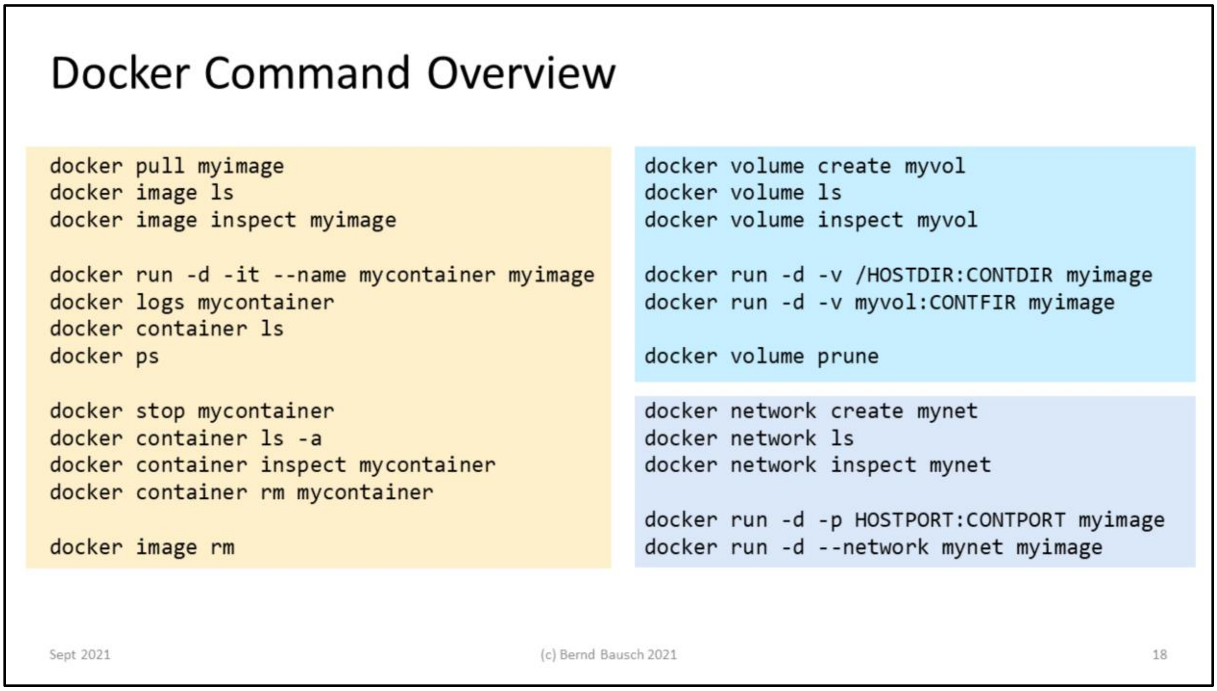
Other Tasks

Logging, Command Overview, docker-compose



Logging

```
# docker run -dt --name --mywebserver httpd
# docker logs mywebserver
AH00558: httpd: Could not reliably determine the server's fully qualified domain name,
using 10.88.0.2. Set the 'ServerName' directive globally to suppress this message
AH00558: httpd: Could not reliably determine the server's fully qualified domain name,
using 10.88.0.2. Set the 'ServerName' directive globally to suppress this message
[Thu Oct 07 10:18:54.405655 2021] [mpm_event:notice] [pid 1:tid 140131438331008]
AH00489: Apache/2.4.50 (Unix) configured -- resuming normal operations
[Thu Oct 07 10:18:54.406854 2021] [core:notice] [pid 1:tid 140131438331008] AH00094:
Command line: 'httpd -D FOREGROUND'
```



Docker Command Overview

```
docker pull myimage
docker image ls
docker image inspect myimage

docker run -d -it --name mycontainer myimage
docker logs mycontainer
docker container ls
docker ps

docker stop mycontainer
docker container ls -a
docker container inspect mycontainer
docker container rm mycontainer

docker image rm
```

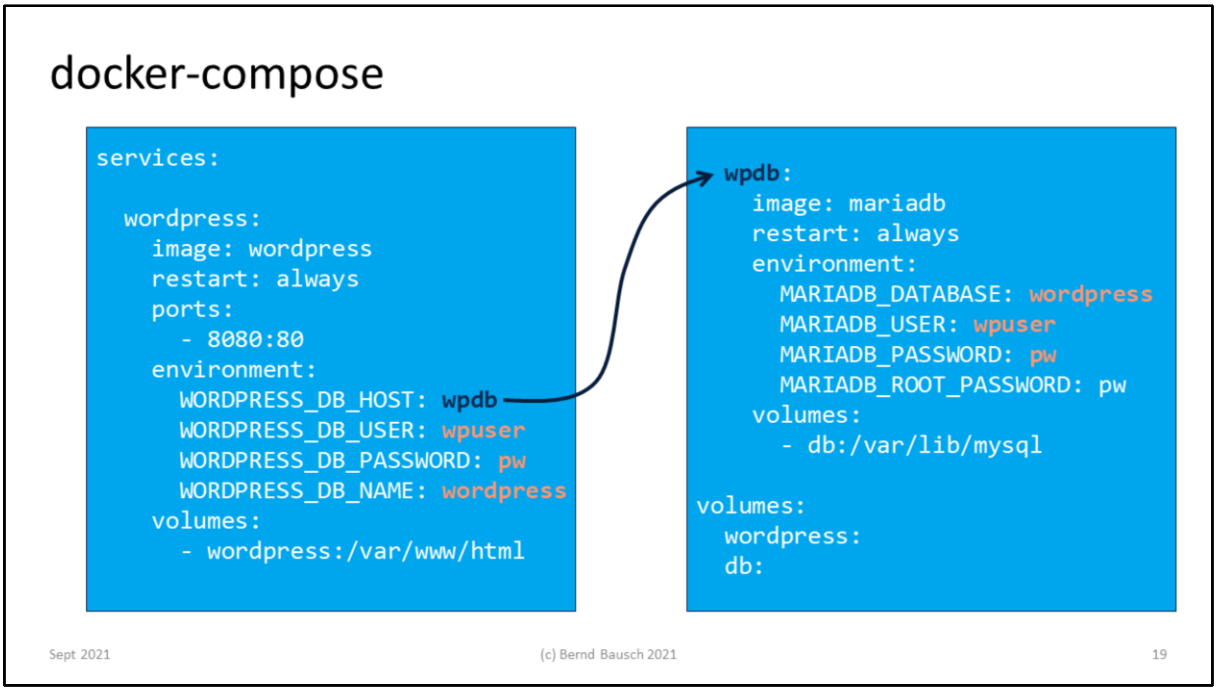
```
docker volume create myvol
docker volume ls
docker volume inspect myvol

docker run -d -v /HOSTDIR:CONTDIR myimage
docker run -d -v myvol:CONTFIR myimage

docker volume prune
```

```
docker network create mynet
docker network ls
docker network inspect mynet

docker run -d -p HOSTPORT:CONTPORT myimage
docker run -d --network mynet myimage
```



docker-compose

```
services:

  wordpress:
    image: wordpress
    restart: always
    ports:
      - 8080:80
    environment:
      WORDPRESS_DB_HOST: wpdb
      WORDPRESS_DB_USER: wpuser
      WORDPRESS_DB_PASSWORD: pw
      WORDPRESS_DB_NAME: wordpress
    volumes:
      - wordpress:/var/www/html
```

```
  wpdb:
    image: mariadb
    restart: always
    environment:
      MARIADB_DATABASE: wordpress
      MARIADB_USER: wpuser
      MARIADB_PASSWORD: pw
      MARIADB_ROOT_PASSWORD: pw
    volumes:
      - db:/var/lib/mysql

volumes:
  wordpress:
  db:
```

Sept 2021 (c) Bernd Bausch 2021 19

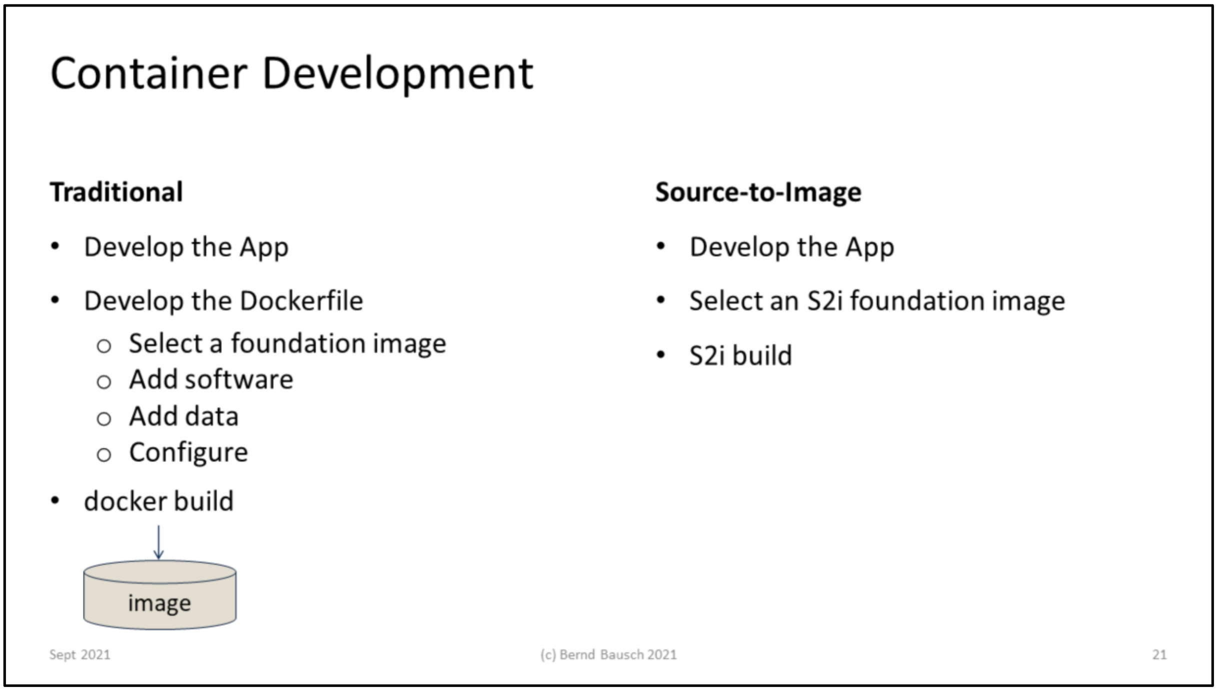
docker-compose is a tool that manages multi-container applications. Rather than running several commands to create containers, volumes, networks and other resources, the user describes the application's properties in a YAML file and uses docker compose commands to start, stop and manage the containers.

The depicted docker compose file is almost verbatim from the wordpress page in docker hub. It describes two **services** consisting of a wordpress and a mariadb container, respectively. The wordpress container references the database container by name.

Container
Development and Deployment

Building and Pushing Images

Sept 2021 (c) Bernd Bausch 2021 20



Container Development

Traditional

- Develop the App
- Develop the Dockerfile
  - Select a foundation image
  - Add software
  - Add data
  - Configure
- docker build

image

Source-to-Image

- Develop the App
- Select an S2i foundation image
- S2i build

Sept 2021 (c) Bernd Bausch 2021 21

The list on the left illustrates a possible development workflow for containerized applications. The application is developed in a separate environment, then packaged as an image that can be deployed via the Docker Hub or another registry.

The `docker build` command is used to create the image. It uses a script named *Dockerfile* that contains Docker-specific instructions. The foundation image contains all or part of the fundamental software needed to run the application, such as a set of commands, libraries, and filesystem trees for configuration files, temporary data and application data.

Rather than developing the app separately and packaging it into an image when it is complete, application development can be done with a container. For example, the developer could launch a container from the foundation image and use a bind mount for hosting the application code. Since the bind mount is accessible from both the host and the container, the application can be developed using tools available on the host and tested by running it inside the container. The developer can build an image at each intermediate application development step and test the result using a container.

Writing Dockerfiles can become a non-trivial task. Red Hat's source-to-image technology enables application development and deployment without Dockerfile. This works by including scripts in the S2i-enabled image that perform tasks equivalent to Dockerfile instructions. All the developer has to do is provide the application source code, select the appropriate S2i image and launch the S2i build command. S2i is integrated in OpenShift, but a slightly different variety is also available standalone.

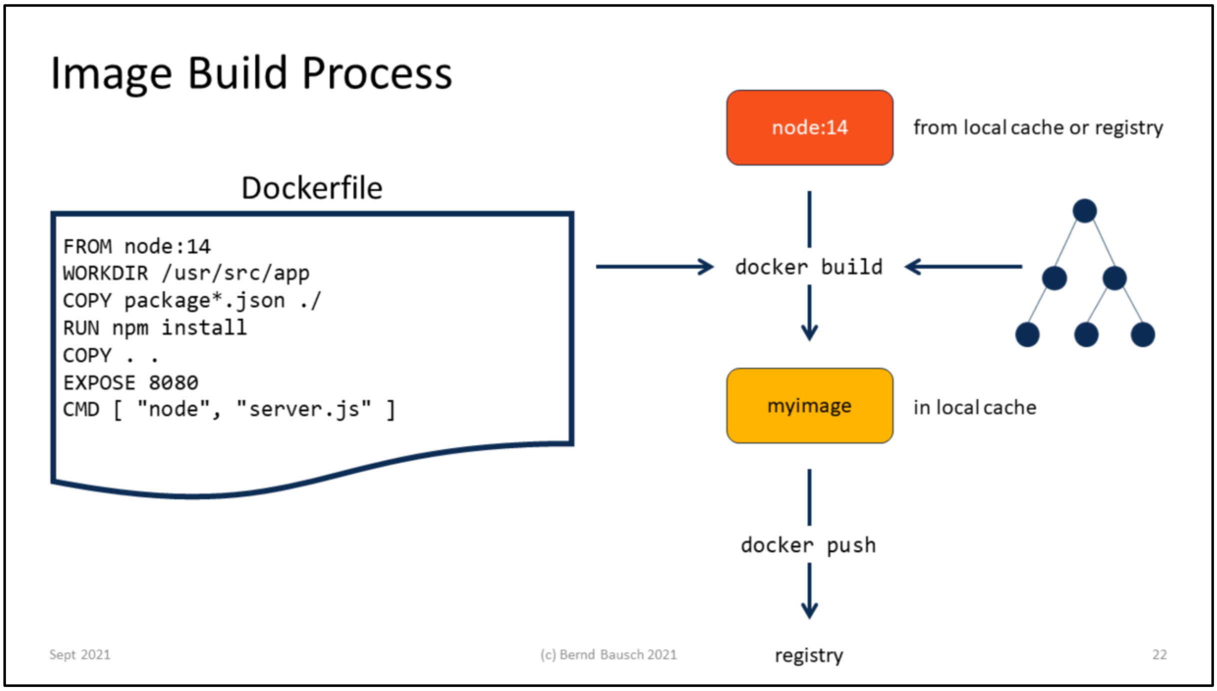


Image Build Process

Dockerfile

```
FROM node:14
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 8080
CMD [ "node", "server.js" ]
```

node:14

from local cache or registry

docker build

myimage

in local cache

docker push

registry

Sept 2021 (c) Bernd Bausch 2021 22

The Dockerfile on the left first executes the FROM directive, which downloads the *node* image version 14 from Docker Hub (or gets it from the local container host's image cache if it was downloaded before).

The remaining instructions create the application image by adding modifications to the foundation. This example declares the application's working directory in the container, adds the nodejs-specific package description (the first COPY command), runs the `npm install` command, which uses `package.json` to install required nodejs modules, copies the application code into the image, and declares the network port and the command that will be run when a container is launched from the image.

Note that Dockerfile directives are executed from top to bottom, like a script. A Dockerfile is not a description of an image, but a list of the steps that create the image.

The EXPOSE directive includes the network port on which the application listens in the image's metadata.

It is easy to confuse the RUN and CMD directives. RUN commands are executed at image build time. CMD commands are executed when a container is launched from the image.

To generate the image, the developer uses `docker build`. This command processes the Dockerfile, creates the image and stores it in the local image cache. The `docker push` command then deploys the image to a registry.

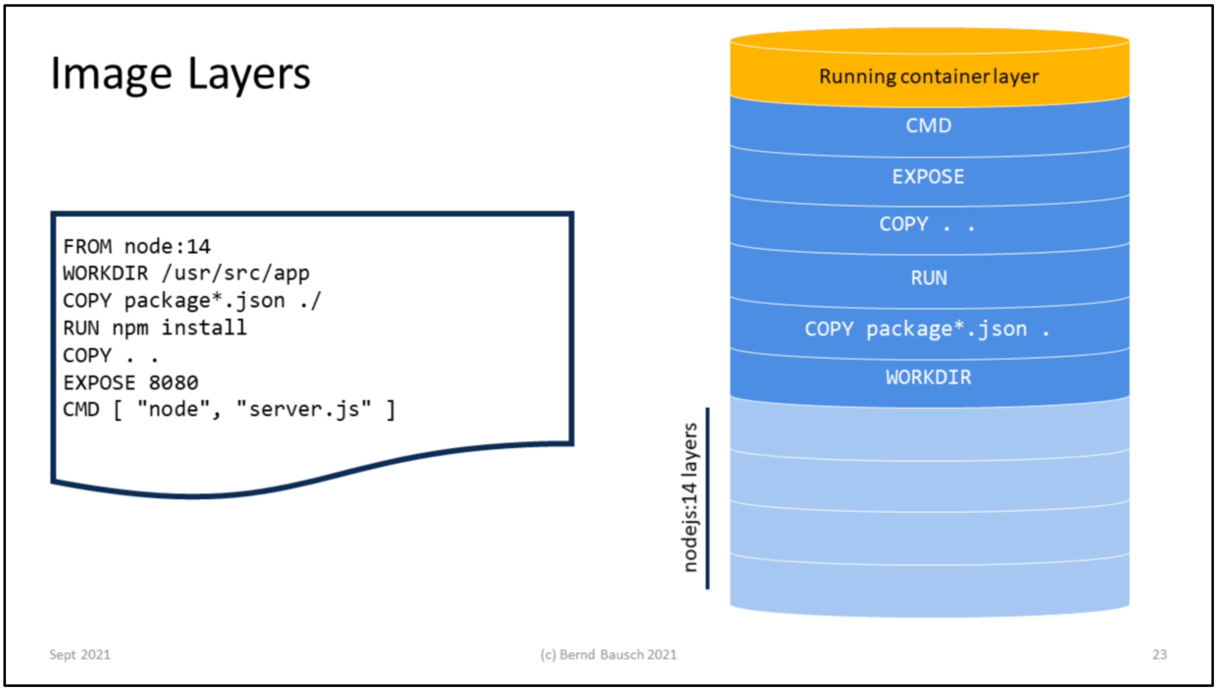


Image Layers

```
FROM node:14
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 8080
CMD [ "node", "server.js" ]
```

Sept 2021 (c) Bernd Bausch 2021 23

Docker images are layered. The node image that serves as the starting point for the build consists of several layers already. Each layer is an archive of files that is downloaded, then unpacked when an image is retrieved from a registry. This is done in parallel.

Each directive in the Dockerfile adds another layer. The COPY and RUN directives generate files that are packaged in an archive. The WORKDIR, EXPOSE and CMD directives only generate image metadata and don't add files to the image, but they also generate layers.

The build process is optimized. Only those layers that require changes are created. For example, the code of the node application shown here resides in the layer created by the second COPY command. All earlier layers are unchanged when developing the application; in particular, the `npm install` command, which can take some time, only needs to be executed at the first build. Thus, a typical cycle could be

1. Write initial application code
2. `docker build` (takes some time due to `npm install`)
3. Test by launching container from image
4. modify application code
5. `docker build` (quick because there is no need for `npm install`)
6. Test by launching container from image

... repeat steps 4 go 6 until application is final

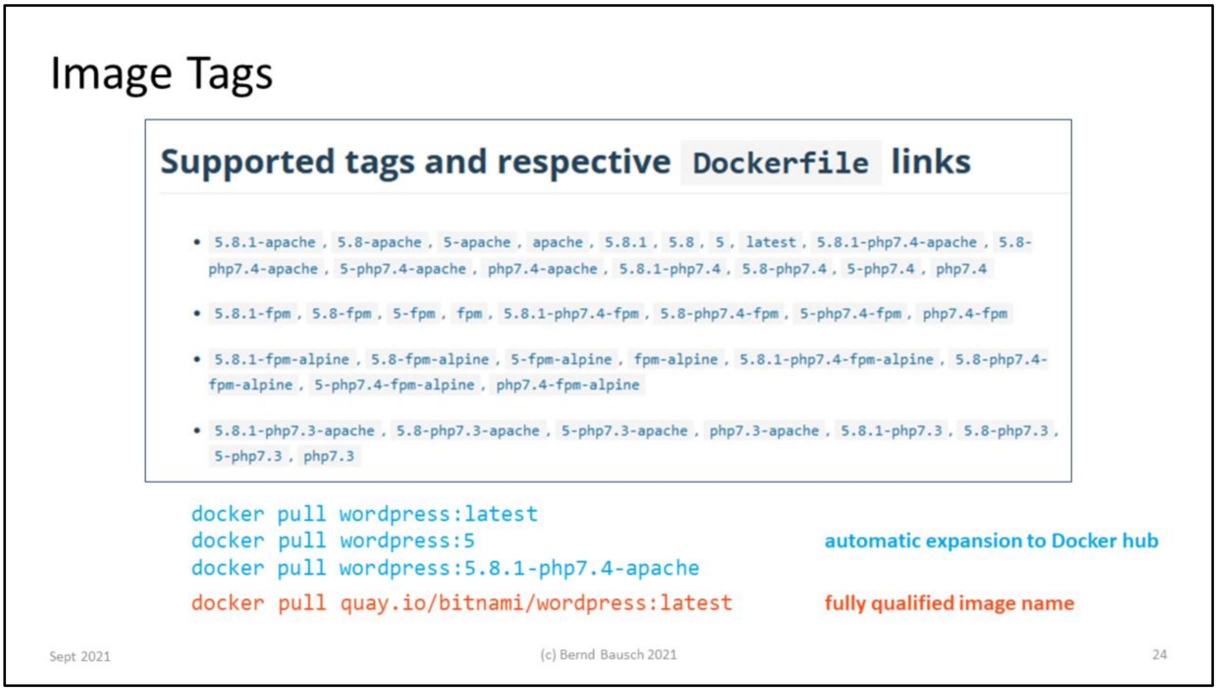


Image Tags

Supported tags and respective `Dockerfile` links

- `5.8.1-apache`, `5.8-apache`, `5-apache`, `apache`, `5.8.1`, `5.8`, `5`, `latest`, `5.8.1-php7.4-apache`, `5.8-php7.4-apache`, `5-php7.4-apache`, `php7.4-apache`, `5.8.1-php7.4`, `5.8-php7.4`, `5-php7.4`, `php7.4`
- `5.8.1-fpm`, `5.8-fpm`, `5-fpm`, `fpm`, `5.8.1-php7.4-fpm`, `5.8-php7.4-fpm`, `5-php7.4-fpm`, `php7.4-fpm`
- `5.8.1-fpm-alpine`, `5.8-fpm-alpine`, `5-fpm-alpine`, `fpm-alpine`, `5.8.1-php7.4-fpm-alpine`, `5.8-php7.4-fpm-alpine`, `5-php7.4-fpm-alpine`, `php7.4-fpm-alpine`
- `5.8.1-php7.3-apache`, `5.8-php7.3-apache`, `5-php7.3-apache`, `php7.3-apache`, `5.8.1-php7.3`, `5.8-php7.3`, `5-php7.3`, `php7.3`

```
docker pull wordpress:latest
docker pull wordpress:5
docker pull wordpress:5.8.1-php7.4-apache
```

automatic expansion to Docker hub

```
docker pull quay.io/bitnami/wordpress:latest
```

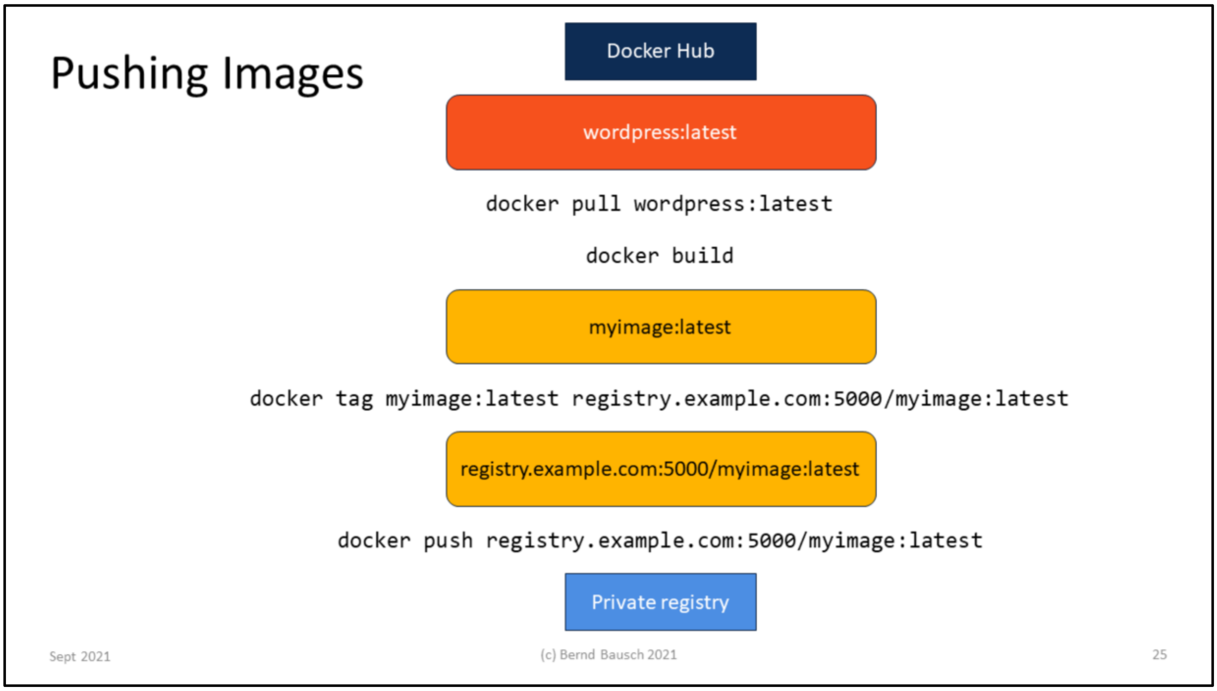
fully qualified image name

Sept 2021 (c) Bernd Bausch 2021 24

A **tag** is the actual image name. It is commonly used to include the image version in its name. The slide shows the first bullets of the tag section on the WordPress Docker hub page. Each bullet stands for one image version. The same version has several tags. For example, the tags in the first bullet refer to an image with Wordpress 5.8.1 built on Apache and PHP 7.4.

You can get the latest version with `docker pull wordpress:latest` or `docker pull wordpress:5`, or `docker pull wordpress:5.8.1-php7.4-apache`.

Commands that pull an image, i.e. `docker pull` and `docker run`, can refer to the image by its tag only, e.g `wordpress:latest`. In this case, the correct image is downloaded from the Docker Hub. However, the Docker Hub is not the only registry in existence; there are several public registries, and you may have your own private registries. To get images from registries other than the Docker hub, you need to specify the fully qualified image name consisting of the registry's URL and the tag.



Pushing Images

Docker Hub

wordpress:latest

```
docker pull wordpress:latest
```

```
docker build
```

myimage:latest

```
docker tag myimage:latest registry.example.com:5000/myimage:latest
```

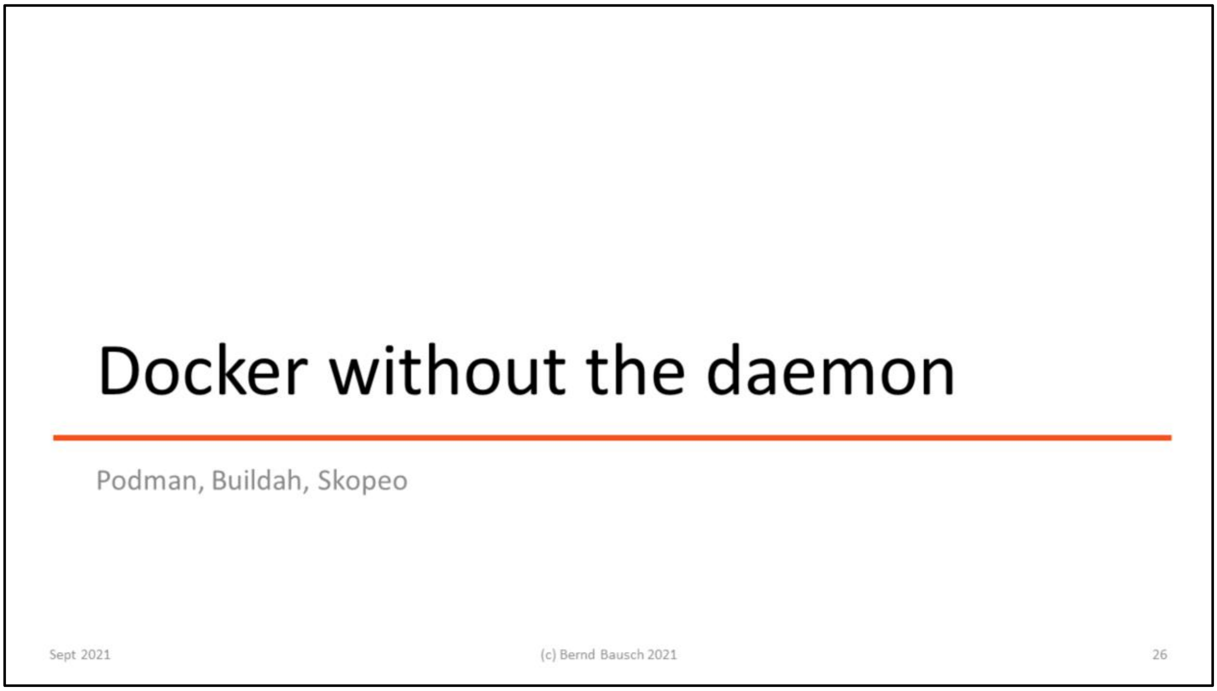
registry.example.com:5000/myimage:latest

```
docker push registry.example.com:5000/myimage:latest
```

Private registry

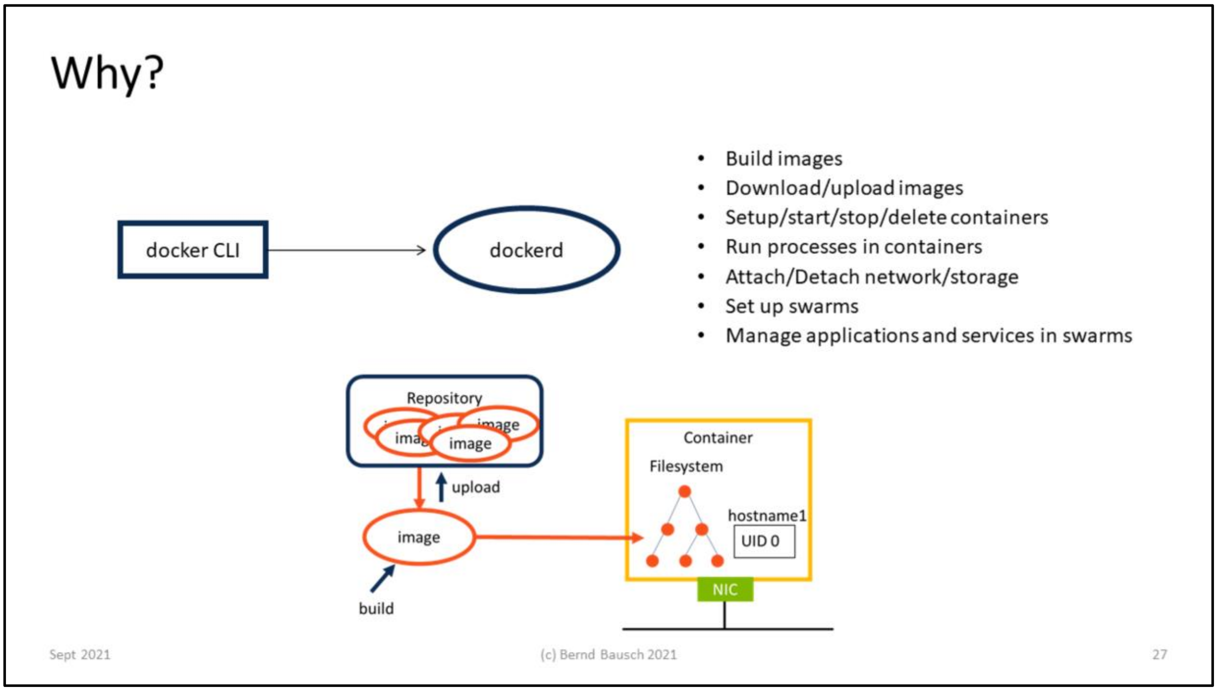
Sept 2021 (c) Bernd Bausch 2021 25

The `docker push` command uploads an image to a registry. If only the image tag without the registry's URL is specified, the image goes to the Docker hub. A fully qualified image name is pushed to the registry specified in the name. However, `docker push` must find this name in the local image cache. Therefore, before you can upload an image to a registry other than the Docker hub, you need to give it a tag that includes the destination registry's URL.



Docker without the daemon

Podman, Buildah, Skopeo

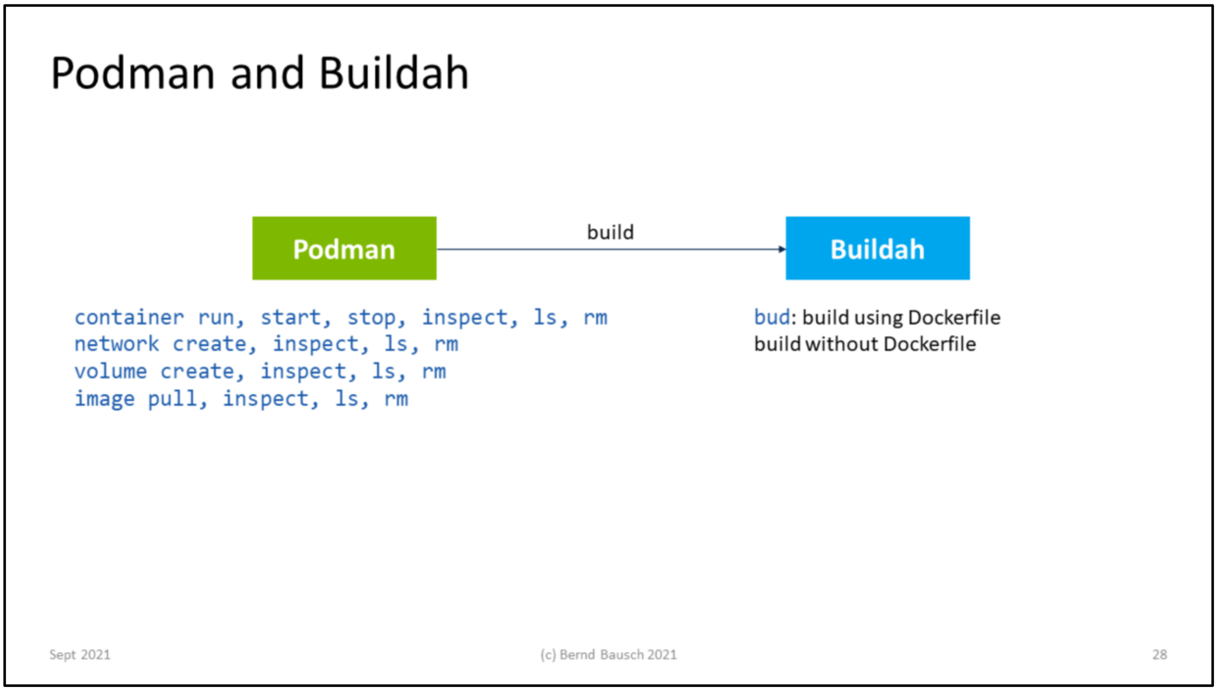


Why?

- Build images
- Download/upload images
- Setup/start/stop/delete containers
- Run processes in containers
- Attach/Detach network/storage
- Set up swarms
- Manage applications and services in swarms

Sept 2021 (c) Bernd Bausch 2021 27

Docker has a rich feature set that may be overkill. For example, to develop an application and create an image from it, docker-compose and swarms may not be needed. A daemon running in the background is another moving part which may cause trouble. Finally, rootless Docker is not a mature and widely available technology at the time this course is written.

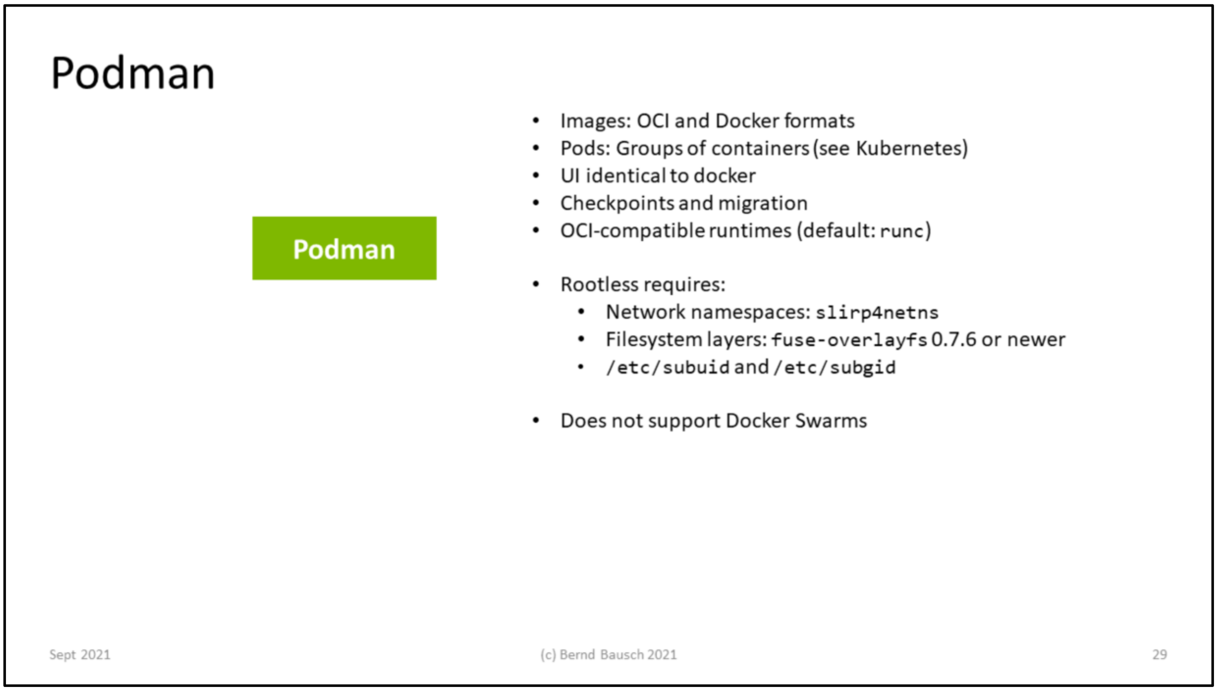


Podman and Buildah

Podman → build → Buildah

```
container run, start, stop, inspect, ls, rm
network create, inspect, ls, rm
volume create, inspect, ls, rm
image pull, inspect, ls, rm
```

bud: build using Dockerfile
build without Dockerfile



Podman

Podman

- Images: OCI and Docker formats
- Pods: Groups of containers (see Kubernetes)
- UI identical to docker
- Checkpoints and migration
- OCI-compatible runtimes (default: `runc`)

- Rootless requires:
  - Network namespaces: `slirp4netns`
  - Filesystem layers: `fuse-overlayfs` 0.7.6 or newer
  - `/etc/subuid` and `/etc/subgid`

- Does not support Docker Swarms

Sept 2021 (c) Bernd Bausch 2021 29

A few noteworthy features of Podman (not a complete list):

- It can launch containers from Docker but also OCI-compatible images
- The name Podman indicates that it can manage pods, i.e. groups of containers. The pod concept is from Kubernetes.
- It supports most (all?) non-swarm commands of the docker client
- It allows checkpointing running containers. This feature can be used for copying a container to a different host (migration).
- Any OCI-compatible runtime can be used. The default is runc.
- Non-privileged containers are supported. This is also known as rootless mode. A few prerequisites exist: To enable user-manageable network namespaces and filesystem layers, slirp4netns and the fuse-overlayfs filesystem must be available. This is the case in Centos/RHEL 8. The administrator must configure user and group ID mappings in /etc/subuid and /etc/subgid.