erasmus
HOGESCHOOL BRUSSEL

Samenvatting theorie Containerization

Server OS

Bachelor Toegepaste Informatica
Graduaat Netwerken

Auteur : Geert Coulommier

Design, Technologie en Gezondheidszorg

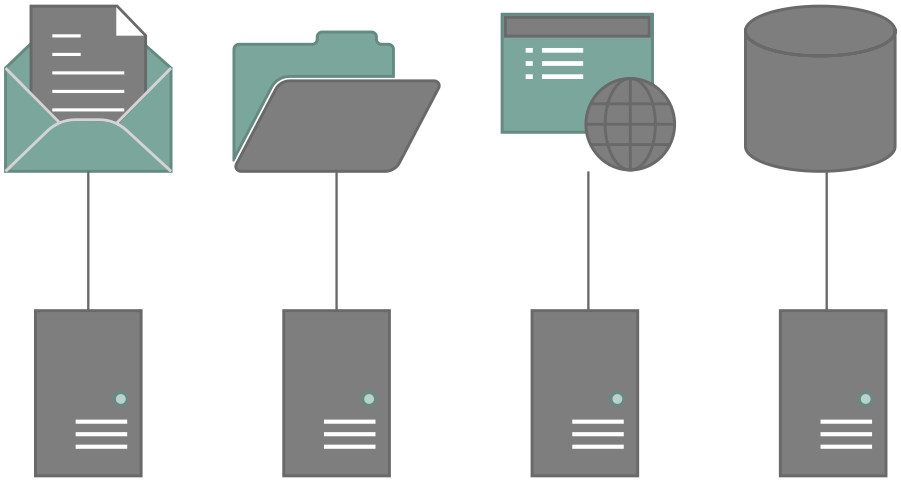
1. Contents

2. Container architecture

- Applications / services are crucial in the operation of organizations
- How to provision them in a secure and efficient way?
  - Should be isolated from each other
  - Should only use the resources it needs
  - Scalable

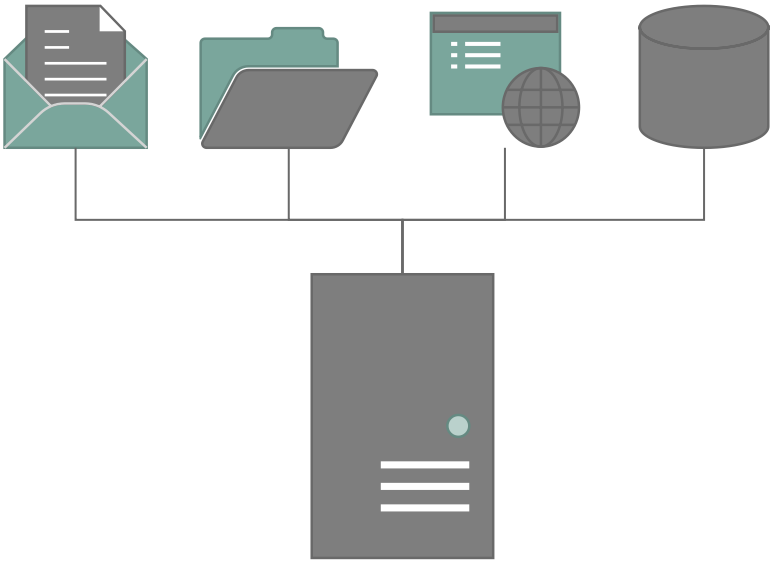
2.1 Before 2000:

- Very often 1 application / service for each hardware server
- To be on the safe side for future upscaling of the application the hardware was very often overprovisioned
- Waste of resources
  - Hardware
  - Support
  - Uptime

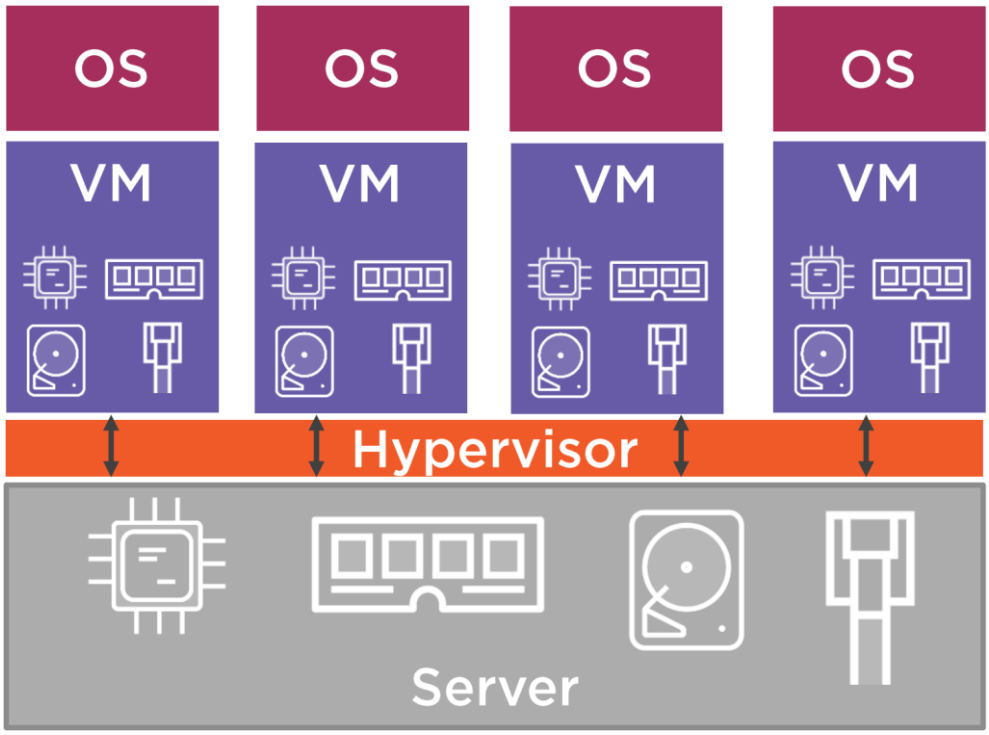


2.2 From 2000 onwards:

- Virtualization gets popular
- Hardware is shared between several Virtual Machines (VM's) through the hypervisor
- Less waste of resources



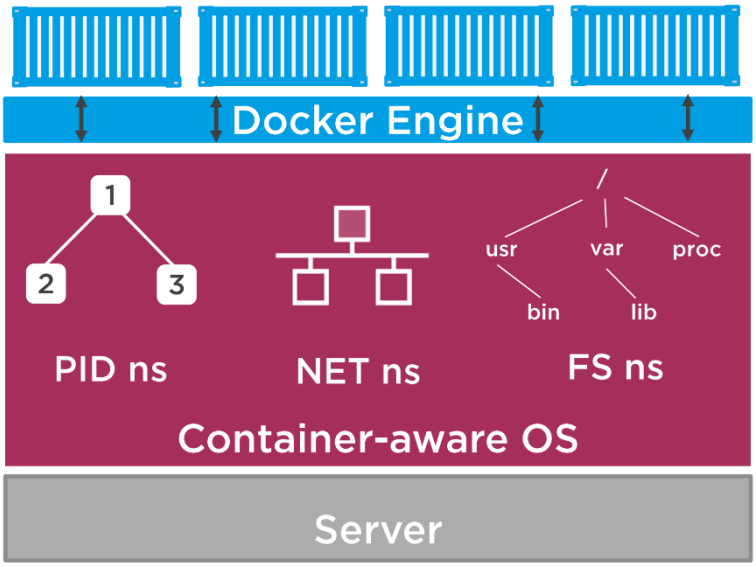
Every app gets its own VM with its own virtual hardware and virtual Operating System
  - Which requires resources for every VM
  - Often the same OS in different VM's



https://app.pluralsight.com/library/courses/docker-getting-started/exercise-files

2.3 Containers

- Apps are isolated from one another in separated containers, with for each container separate namespaces:
  - filesystem
  - process tree
  - users and groups tree
  - network stack



https://app.pluralsight.com/library/courses/docker-getting-started/exercise-files

- Each container acts like a complete separate OS
- Resources used by containers are limited by control groups (cgroups)
- Containers share the same OS kernel from the host computer
- Only one OS kernel necessary, so less resources needed
- As containers use the kernel for the host, containers need the same kernel as the host
  - Linux containers on Linux hosts
  - Windows containers on Windows hosts
  - VM's can be used as host for mixed solutions
    - Linux container on Windows host with a Linux VM in Hyper-V
- One application per container makes them
  - Scalable
  - Self-healing
  - Portable
  - Lightweight and fast
  - Efficient in resource usage
- Containers have existed in the Linux world for decades

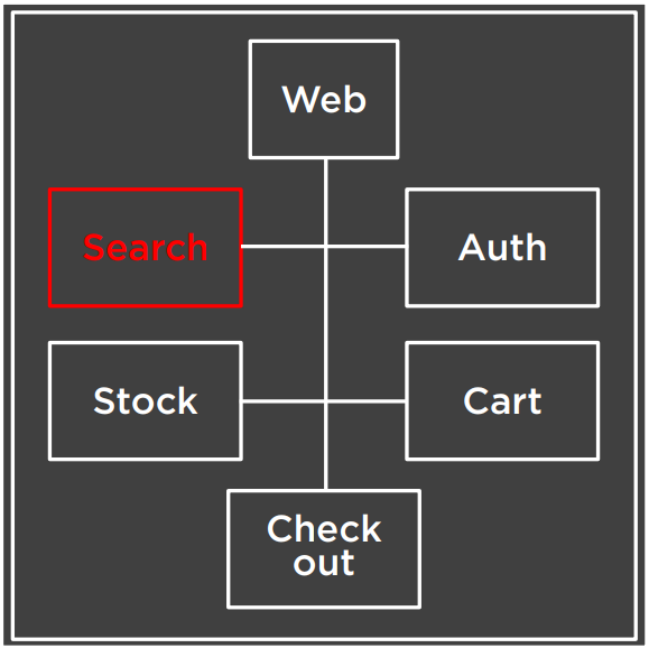
- e.g. Google's search engine

2.4

- Docker made the controlling of namespaces and cgroups easy by bundling them in easy manageable containers
- Controls individual containers, in essence starting and stopping
- De facto standard in containerization
  - Mainly in Linux, but also for Windows and Mac
  - Platform independent:
    - cloud, on-prem, hybrid, Linux, Mac, Windows: commands are the same

3. Microservices and Cloud-Native

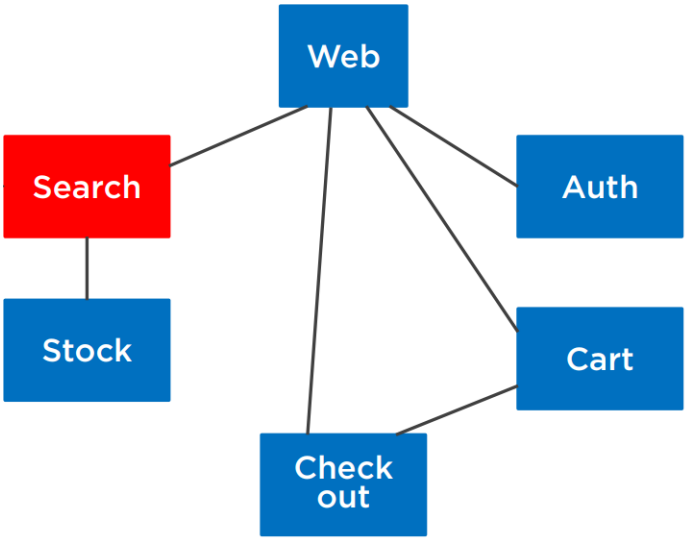
- Monolithic app architecture
  - Large applications with all functionality integrated in one big binary
  - The whole entity had to be taken down to fix/upgrade a single item (e.g. search)



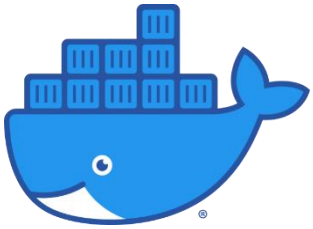
https://app.pluralsight.com/library/courses/docker-kubernetes-big-picture/exercise-files

3.1 Microservices:

- All the separate components of an application are split up in separate components
- When one component needs to get fixed/updated, only that component needs to be taken down and replaced



https://app.pluralsight.com/library/courses/docker-kubernetes-big-picture/exercise-files



- Containers are used as microservices
- One app/process per container
  - Scalable
  - Self-healing
  - Portable
  - Efficient in resource usage
- Docker itself has also been written in a Microservices architecture
  - Split up in different components that can be managed and changed separately from one another
  - Client-Server model:
    - CLI Client (docker)
      - Allows interacting with the containers, like starting and stopping commands
      - Connects to the daemon (local or re mote) for actual operations
    - Daemon (dockerd)
      - Provides API for the client to interact with
      - Manages the actual containers
- This architecture allows to upgrade the client without having to take down running containers

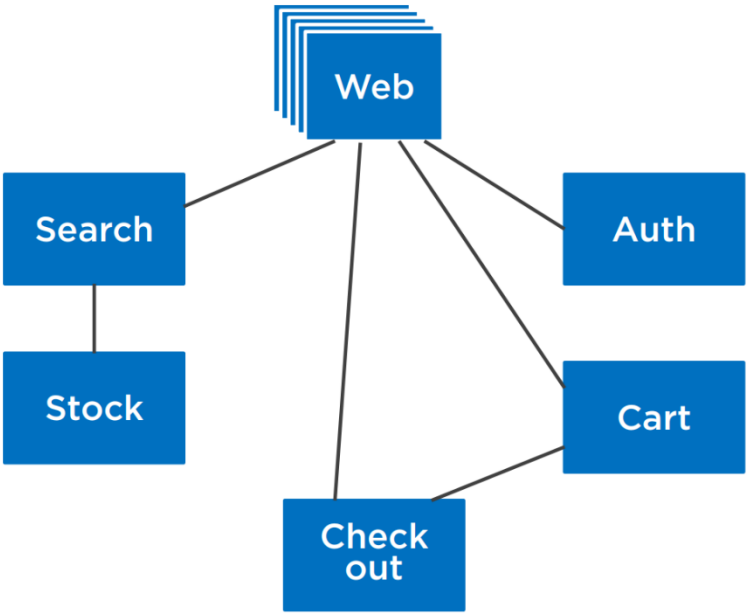
3.2 Cloud-Native

- Optimize microservices with containers for the cloud and you get Cloud-Native apps
- "Cloud native computing uses an open source software stack to be:
  - Containerized. Each part (applications, processes, etc) is packaged in its own container. This facilitates reproducibility, transparency, and resource isolation.
  - Dynamically orchestrated. Containers are actively scheduled and managed to optimize resource utilization.
  - Microservices-oriented. Applications are segmented into microservices. This significantly increases the overall agility and maintainability of applications."

https://www.cncf.io/about/faq/

4. Orchestration

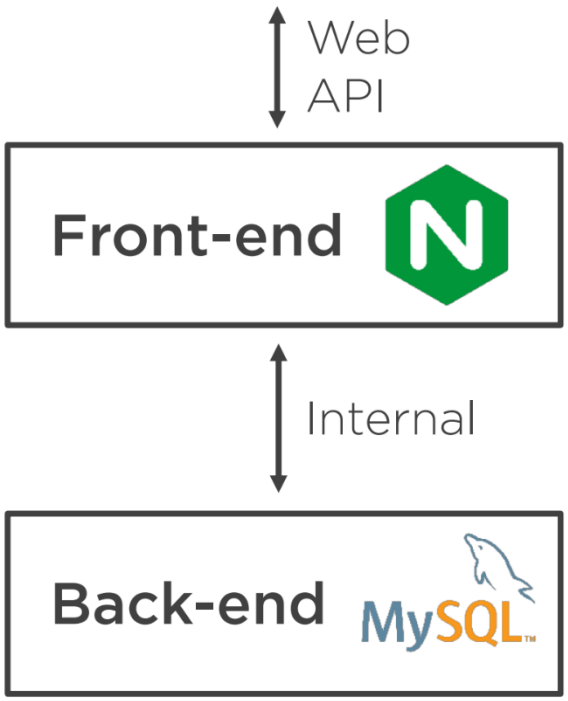
- Microservices architecture can become complex…
  - Single app per container often requires many containers for the full stack
  - Often multiple instances of the same container
  - Containers need to communicate with one-another in often complex layout



https://app.pluralsight.com/library/courses/docker-kubernetes-big-picture/exercise-files

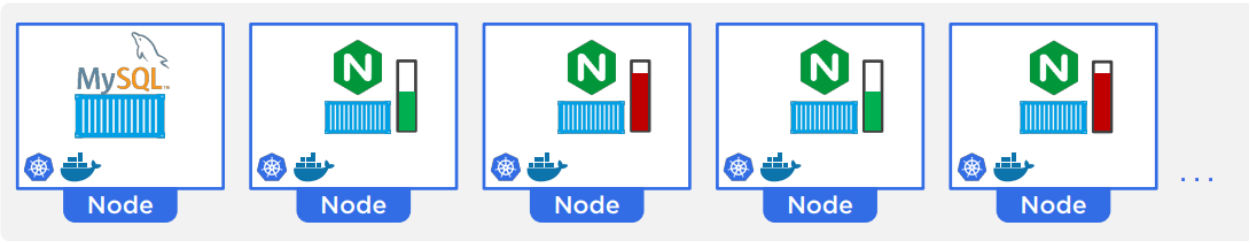
- But also… microservices architecture can become complex
  - Containers need to be dynamically managed according to load and failures
  - Containers are often spread-out over multiple hosts
    - On-premises (on-prem), cloud or hybrid
- Orchestration will automate the task related to the management of container stacks
  - Comparison: orchestra: every musician is like a container, where the conductor of the orchestra starts and stops different groups, sets the tempo, and as such manages the stack

- For instance, consider a stack with NginX as web front-end and MySQL as backend. For load balancing, 2 instances of the NginX container are desired under normal load.



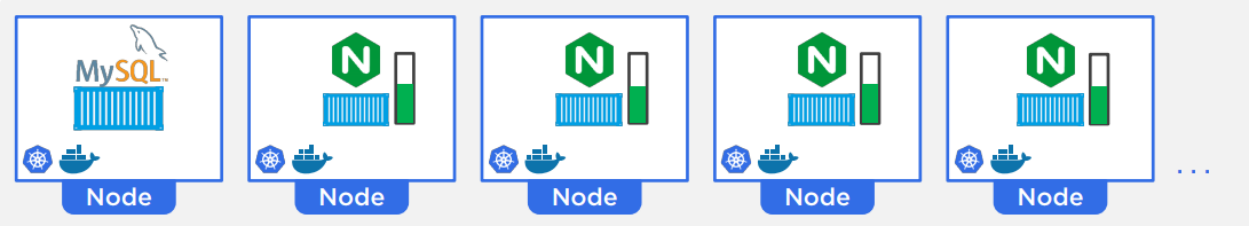
https://app.pluralsight.com/library/courses/docker-kubernetes-big-picture/exercise-files

- The orchestrator will automatically add 2 more instances of NginX (2 and 4) if the 2 existing ones (3 and 5) get too much load.



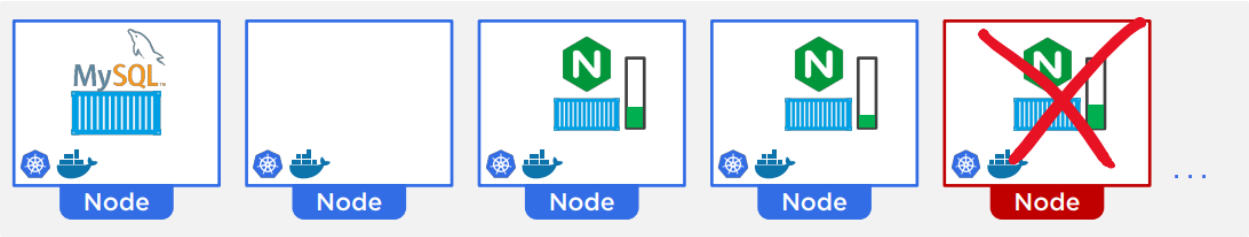
https://app.pluralsight.com/library/courses/docker-kubernetes-big-picture/exercise-files

- The load will automatically be balanced among all instances.



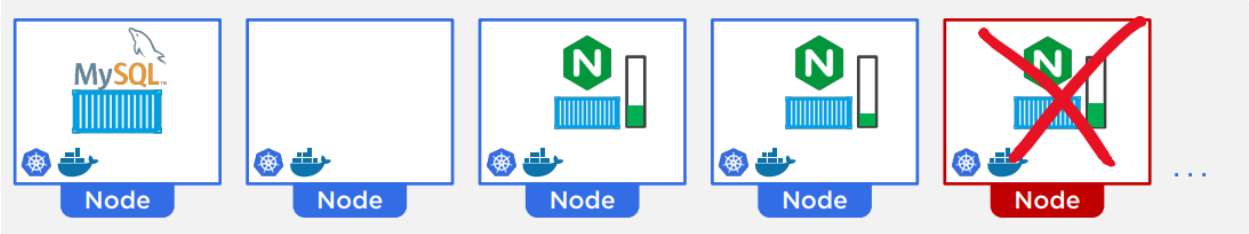
https://app.pluralsight.com/library/courses/docker-kubernetes-big-picture/exercise-files

- And when the load is back to normal, excessive instances will be removed until the desired state (2 NginX, 1 MySQL) is reached.



https://app.pluralsight.com/library/courses/docker-kubernetes-big-picture/exercise-files

- Also, if an instance were to fail, it will immediately be picked up by the orchestrator, and a replacement instance will be started up.



https://app.pluralsight.com/library/courses/docker-kubernetes-big-picture/exercise-files

- Docker Swarm: Docker's own orchestration
  - Easy to use
  - All basic functionality included
- Kubernetes (K8s): Google's orchestration
  - Large amount of functionality
  - Becoming the de facto standard
  - Integrated in all major Cloud Services
  - Integrated in many Server Class Operating Systems
  - Integrated in Docker

5. Docker concepts

5.1 Docker Images

- Basis for every container, 'template' for a container
- Read-only -> immutable
- Build-time construct
- "stopped" container
- Is run with the `docker run` command to start a container
- Can also be pulled from a registry without immediately starting a container with the `docker pull` command
- Contains several stacked layers to build a unified filesystem, and a json manifest file, describing the image and how the layers should be combined together
  - e.g. bottom layer for the OS files, next layer for the app files, third layer for updates
  - Layers are locally stored
    - in Linux: /var/lib/docker/[storage_driver_name]/diff
    - In Windows: C:\ProgramData\Docker\Windows Filter
  - An extra writable layer is added when a container is created from the image
- Are stored in registries
- Needs to have a copy in the local registry on the host
- If a local copy is not available, it will be automatically downloaded ('pulled') from an image registry to the local registry

5.1.1 Docker registries

- Docker hub (hub.docker.com)
  - Default
  - Contains thousands of images for applications
  - Official images: maintained by the developer of the app
    - Should be stable, up-to-date, tested and well documented
    - Don't need a separate namespace
    - e.g. nginx
  - Unofficial images: not from the official developer of the app
    - Require a separate namespace
    - e.g. nginxdemos/hello
  - Many provide different versions, with the latest version usually being the most up-to-date stable version
- Other public registries, like Google, Amazon, Microsoft…
  - Private registries, e.g. provided and maintained by your own organisation with privately created images

5.1.2 Image naming syntax: **registry/repository:tag**

- e.g. `docker.io/ubuntu:latest`
- **registry**: name of the registry
  - Default: docker.io (docker hub)
  - If the registry is docker hub (docker.io) it does not need to be mentioned (default)
- **repository**: separate space in a registry
- **tag**: name of the actual image in the repository
  - If the tag is `latest`, is does not need to be mentioned (default)
    - `latest` is tagged as such manually by the repository maintainer
    - Usually refers to latest stable
    - Can be older than other available images, like edge (development unstable version)

5.1.3 Linux or Windows Images

- Images are built for a specific kernel
- Windows images are a lot bigger, but necessary for apps that need a Windows kernel

Image	**SIZE**
Windows image for Powershell	5.35GB
Linux image for Powershell	339MB

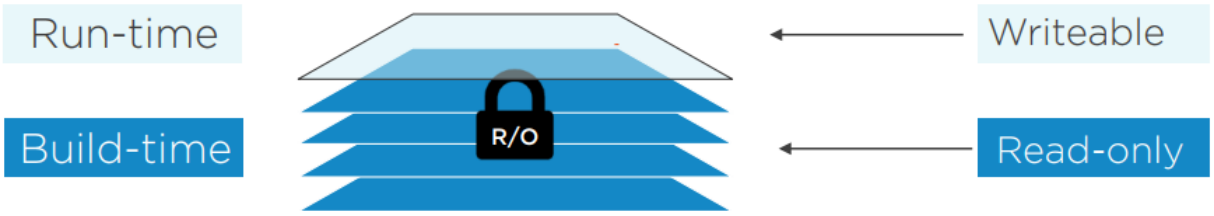
- Base images on which to build an app image are focused on being very lightweight:
  - Linux: alpine
  - Windows: nano server

5.2 Docker containers

- Based on image
- Runtime construct: "running" instance of image
- Multiple instances of same image possible
- Are meant to be ephemeral: only used for set period of time. If changes need to be made, they are done in the image, the container is stopped and replaced with a new container based on the new image.
- When stopping a container, the container is not removed but exited
  - Data persists in the container

- Created to be lightweight
  - Usually only one process running
    - ..in Linux containers. The Windows kernel needs more processes, therefor Windows containers too.
    - e.g. Try docker run –it ubuntu bash for an interactive ubuntu docker and then try vi, it will not work because vi is not available in this image (lightweight). Run top to see only bash and top (forked from bash) running.
    - When exiting the terminal, the bash process will be stopped, the only running process in the container. Therefor, the container itself will also stop.
    - To exit an interactive container without stopping, do [CTRL+P+Q]

5.3 Docker images vs containers



https://app.pluralsight.com/library/courses/docker-deep-dive-update/exercise-files

5.4 Building Docker images

- Steps are very similar to building an app outside of a container environment
- Steps to create an image are defined in a `Dockerfile`
  - **`FROM`**: specifies the image this new image should be based on
  - **`WORKDIR`**: specifies the directory in your image filesystem in which all subsequent actions should be taken
  - **`COPY`**: copies files from the host to the image, e.g. your application files
  - **`EXPOSE`**: documents which ports the application uses
  - **`RUN`**: run a command, e.g. to build the application based upon the imagefiles copied in the previous COPY parameter
- Additional metadata about how to run the container based on this image can also be added to the Dockerfile
  - **`CMD`**: default process to run in the container
- The actual image can be built from the `Dockerfile` with the command…
- `docker image build -t [image_name:tag_name] [path_to_dir_with_all_necessary_files]`
- e.g. `docker image build -t testapplication:1.0 /home/user/testapp`
- This image can be used to start a container
- Building small images
  - Use small base images (FROM)
  - Use multi-stage builds

- The first stage builds the app with the build tools in a temporary image
- Second stage copies the built app over to the final image without the build tools

• Docker concepts

5.5 Persisting data: volumes

- More info: https://docs.docker.com/storage/volumes/
- Containers are meant to be ephemeral
- Where to keep data after stopping/removing the container?
- Volumes
  - allow mapping folders from inside the container to the host
  - container is stopped or removed: data remains
  - can be shared among different containers
  - Can be on a remote host, cloud storage, SAN, NAS…
  - Will by default be created on the host:
    - Linux: /var/lib/docker/volumes

6. Docker installation

- All the different procedures for installing Docker in different environments are available at https://docs.docker.com
- **Docker Desktop**
  - Ideal for testing and development
  - Windows
    - Requires the Containers and Hyper-V features
      - VirtualBox will not work anymore
    - Also possible through Chocolatey
    - Can run from WSL2 too to for linux container support
  - MAC OS
- **Docker in Windows Server**
  - Ideal for production environments
  - Through Powershell
  - Install-Module -Name DockerMsftProvider -Repository PSGallery –Force
  - Install-Package -Name docker -ProviderName DockerMsftProvider
- **Docker in Linux**
  - Suitable for production, development and testing environments
  - Procedure depends on Linux distribution
  - https://docs.docker.com
  - If available, install using the repositories

7. Docker commands

docker

- Displays a quick overview of available docker options and parameters.

docker version

- Displays the installed version of docker.

docker info

- Displays general information of your docker installation.

docker [image] pull (image_name)

- Pulls the latest an image from the registry based on (image_name).

docker [image] pull (image_name):(tag)

- Pulls an image from the registry based on (image_name), with an optional tag referring to version of the image

docker [container] run (image_name)

- Runs a container based on (`image_name`). If the image for the container is not present, it will be downloaded (pulled) from the registry.
- `-d` option: detached: container runs in the background
- `-i -t` option: interactive and TTY -pseudoterminal: gives a cli TTY into the container
- `-p (port_host:port_container)` option: maps the internal `(port_container)` to the (`port_host`), making it available from the host
- `-e (environment variable)` option: sets environment variable
- `--name (container_name)` option: names the newly started container with name `(container_name)`
- `--mount source=(volume_name),target=(path_to_targetfolder)` option: mounts the volume on the host with name `(_volume_name)` to the folder inside the container (`path_to_target_folder)`

docker image

- Manage images
- `ls` command: list. List all the images available locally on the host
  - Same as docker images command
- `rm (image_name)` command: remove. Remove image with name image_name.
  - Same as docker rmi (image_name)
- `build -t` (image_name:image_tag)option: build a new image with name image_name and tag image_tag
- `prune` command. Remove all dangling images.
- `prune -a` command. Remove all unused images.
- `$(docker images -q)` = all images. Can be used with rm
  - e.g. `docker image rm $(docker images -q)`: removes all images

docker container

- Manage containers
- `ls` command: list. List all the running containers on the host.
  - Same as `docker ps` command
- `ls -a` command: list all. List all the containers on the host.
- `rm (container_name/id)`: remove. Remove container with name or id `(container_name/id)`
  - Same as `docker rm (container_name/id)`
- `exec (command)` command: execute. Run the command (command)inside the container (must be running).
  - Can be used with the option `-it` to run the command (command) interactively in the container.

- `stop (container_name/id)` command: stops a running container with name or id `(container_name/id)`
- `start (container_name/id)` command: starts a running container with name or id `(container_name/id)`
- `restart (container_name/id)` command: restarts a running container with name or id `(container_name/id)`
- `$(docker container ls -aq)`: = all containers. Can be used with `rm`, `start`, `stop`, `restart`
  - e.g. `docker container stop $(docker container ls -aq)`: stops all containers

- `logs (container_name/id)` command: shows the logs for the container with name or id (`container_name/id`)
- `top (container_name/id)` command: shows the running processes int the container with name or id `(container_name/id)`
- `prune` command: removes all stopped containers

docker volume

- `create (volume_name)` command: creates a volume with name `(volume_name)`
- `ls` command: list all volumes on the host
- `rm (volume_name)` command: removes volume with name (`volume_name`)

docker system

- `prune command`: stops all...
  - all stopped containers
  - all networks not used by at least one container
  - all dangling images
  - all dangling build cache
- `prune -a` command: stops all…
  - all stopped containers
  - all networks not used by at least one container
  - all images without at least one container associated to them
  - all build cache