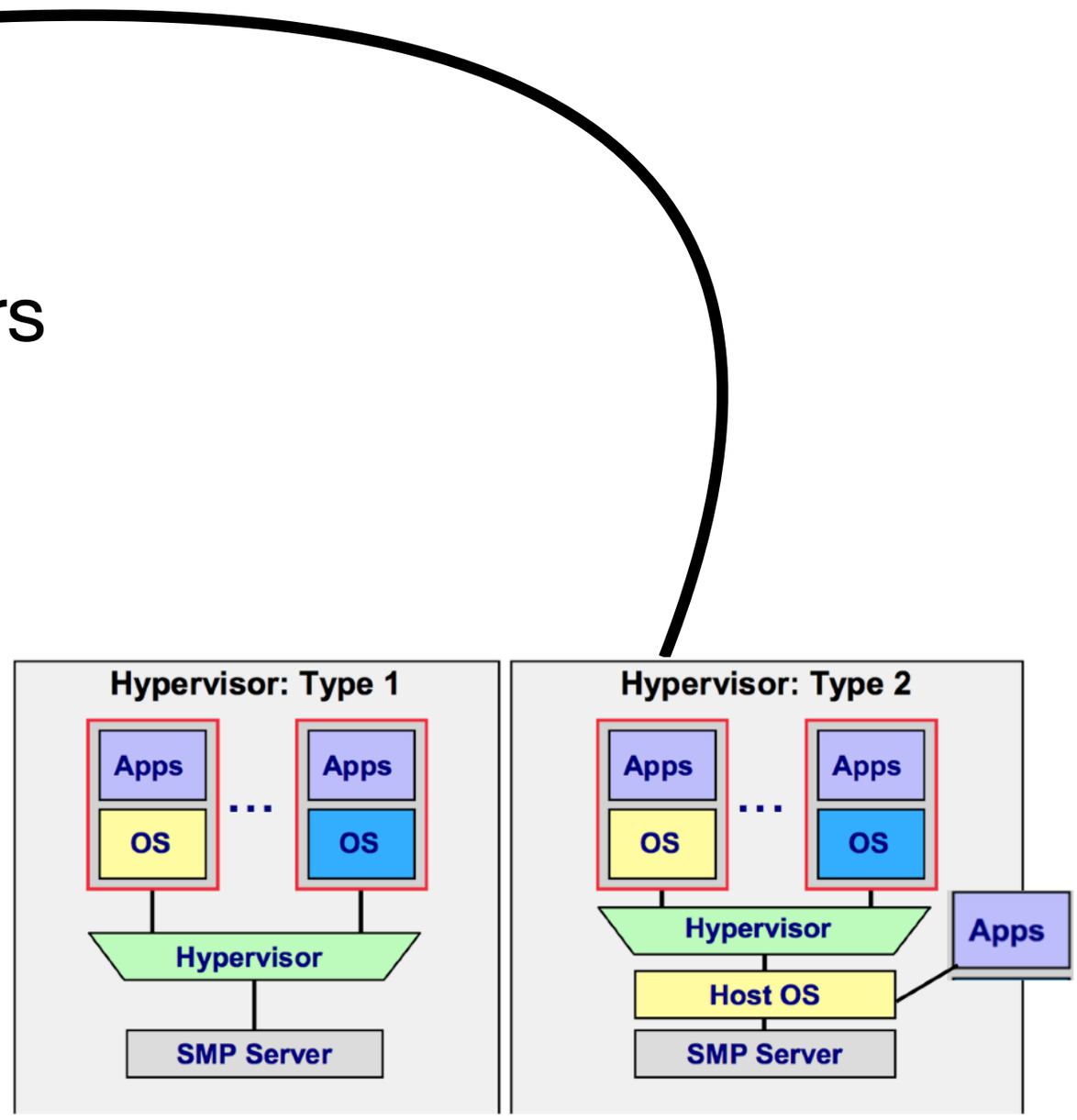
DHBW Informatik Wahlpflicht
Cloud Computing
Chapter 3 PaaS Containers

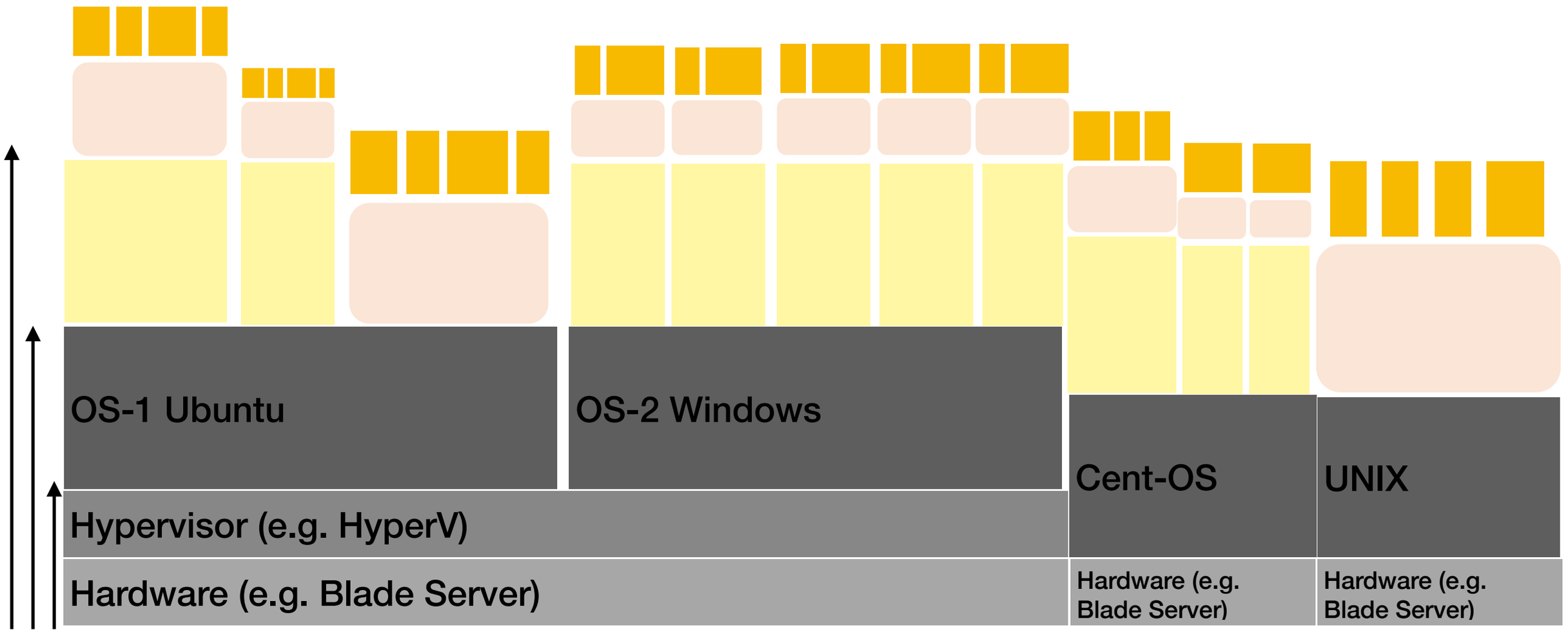
Juergen Schneider

Agenda

- Docker based Linux Containers
- Kubernetes



Work Isolation



Legend:

OS based Container	App Container e.g. Node.JS . JAVA VM, Tomcat	App Code Single Tenant

Workload: IT Applications using compute resources (e.g. CPU Memory, Storage, Network, OS)
Isolation QoS

- Resource Isolation : sharing of IT resources is fair
- Workload Isolation : my workload (including data in memory) is secure
- Manageability : how to set up and control the environment

- Classical IT : Single tenant : tenant owns the (used) IT completely (but runs many different workload)
- Cloud : Multiple tenant : sharing applies on IT resources

What are Containers?

- A group of processes run in isolation
  - Similar to VMs but managed at the process level (share the same kernel)

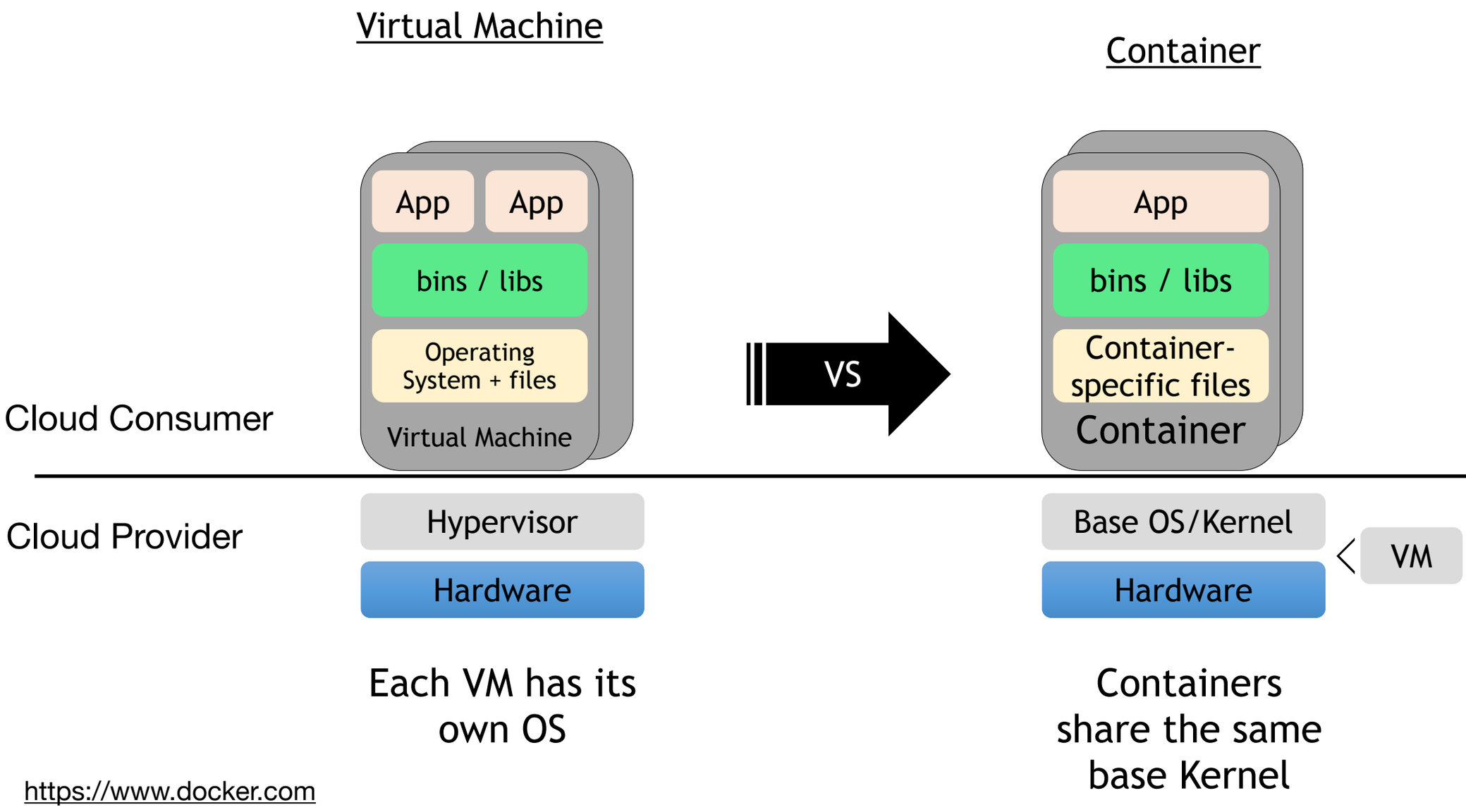
- Each container has its own set of "namespaces" (isolated view)
  - **PIDs -** process IDs
  - **USER -** user and group IDs
  - **UTS -** hostname and domain name
  - **MNT -** mount points
  - **NET -** Network devices, stacks, ports
  - **IPC -** inter-process communications, message queues
  - **cgroups -** controls limits and monitoring of resources

- Are container something new ?
  - No, Linux Namespace(2006), cgroups (2007), LXC(2008), CF Warden(2011) Docker(2013)

- https://www.youtube.com/watch?v=sK5i-N34im8
- FreeBSD Container (Jails) (1999) https://wiki.freebsd.org/Containers
- Windows native Container (Silos) https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/

- Official DB for Security Issues (Nist) https://nvd.nist.gov/

VM vs Container



https://www.docker.com

https://www.redhat.com/en/blog/limits-compatibility-and-supportability-containers

https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction/#h.dqlu6589ootw

Why Containers?

- Fast startup time - only takes milliseconds to:
  - Lay-down the container's filesystem
  - Setup the networks, mounts, ...
  - Start the process

- Better resource utilization
  - Can fit far more containers than VMs into a host

- Workload Files (e.g. Docker Images) are smaller and incremental

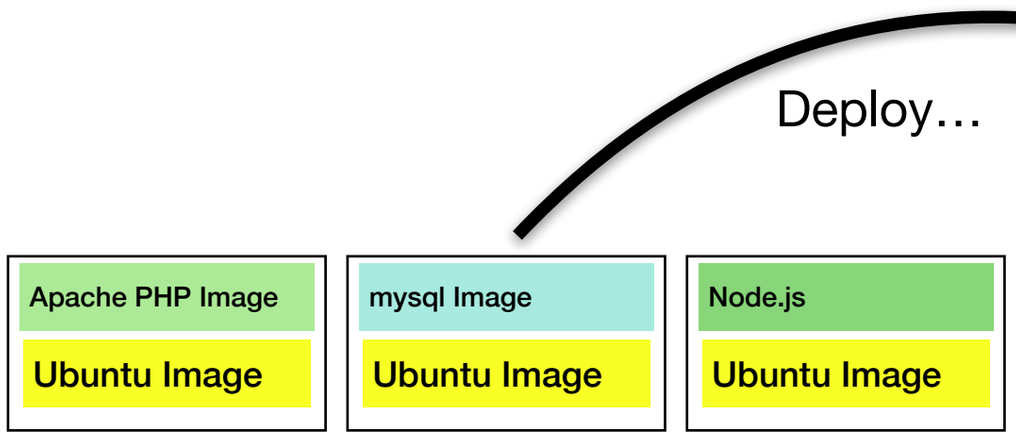
- Useful in the emerging Micro Services Implementations

What is Docker again?

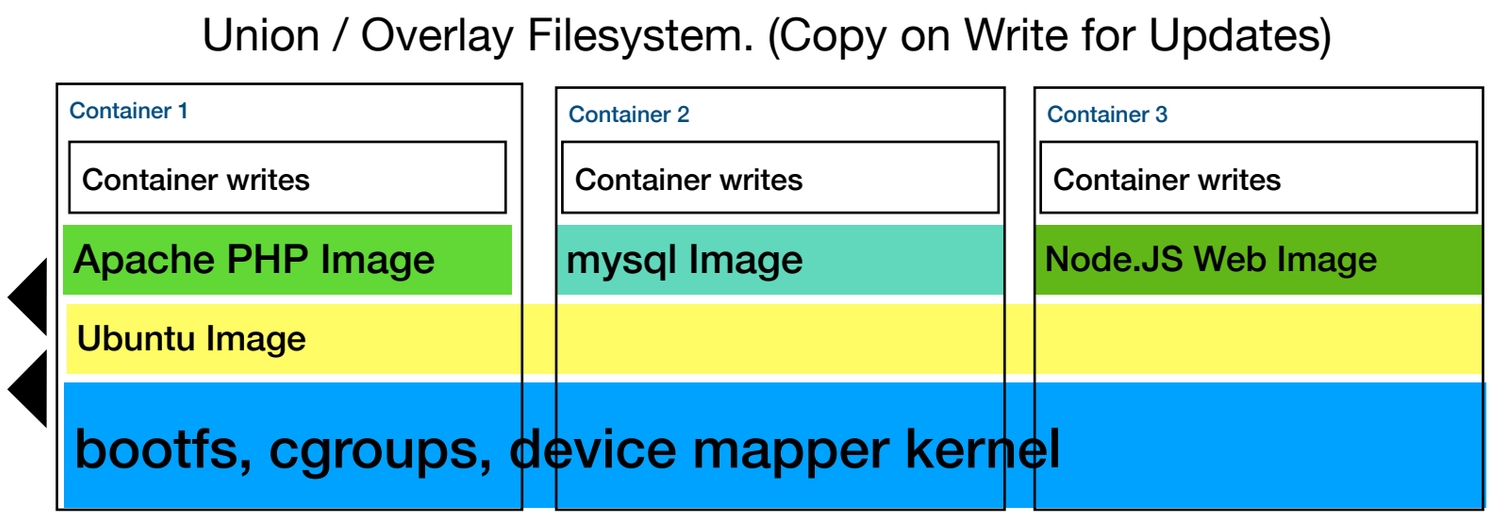
https://www.docker.com/

- Tooling to manage containers
  - Containers are not new (but will be further developed, OCI Open Container Initiative)
  - Docker **made them easy to use**

- Docker creates and manages the lifecycle of containers
  - Setup filesystem based on Docker Images Layers and often the Linux Overlay/Union Filesystem Driver)
  - CRUD container
    - Setup networks
    - Setup volumes / mounts
    - Create: start new process telling OS to run it in isolation



https://docs.docker.com/storage/storagedriver/
https://docs.docker.com/storage/storagedriver/overlayfs-driver/

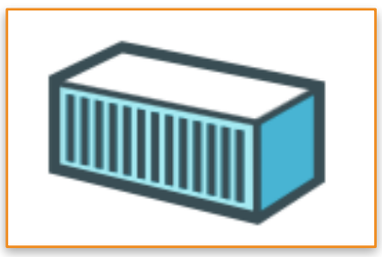


4 Images on top of the kernel (blue) , Ubuntu FS Image shared among the three containers
(Copy on Write for updates on the shared FS !!)

Docker Basics – Terminology

Image

- A read-only snapshot of a container stored in Docker Hub or any other registry to be used as a template for building containers



Container

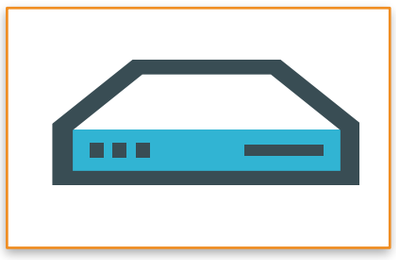
- The standard unit in which the application service resides and will be executed

A docker file

- A set of command used by Docker to build a new image (potentially based on a image from a public repository)

Docker Hub/Registry

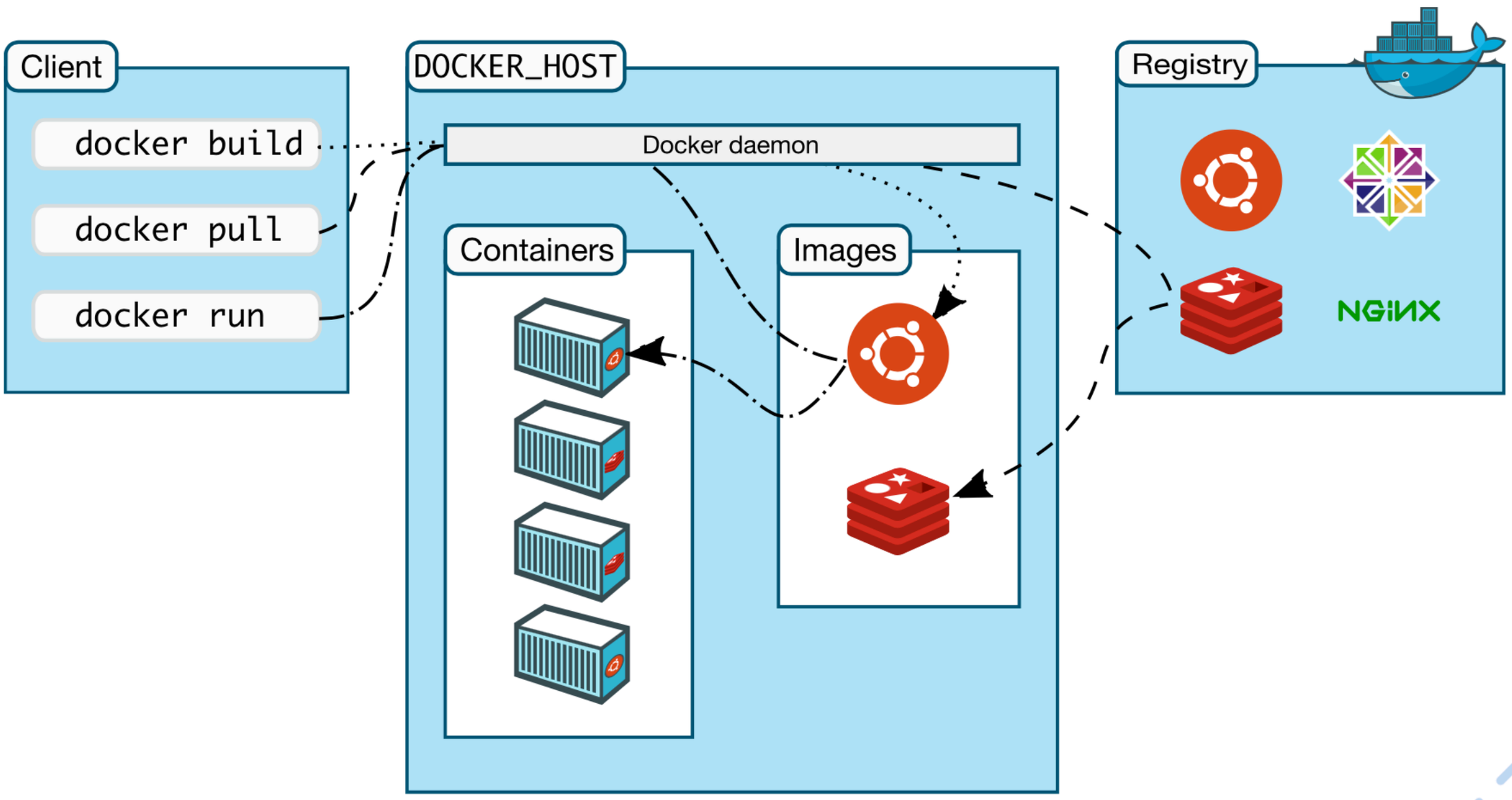
- Available in SaaS or Enterprise to deploy anywhere you choose
- Stores, distributes and shares container images



Docker Engine

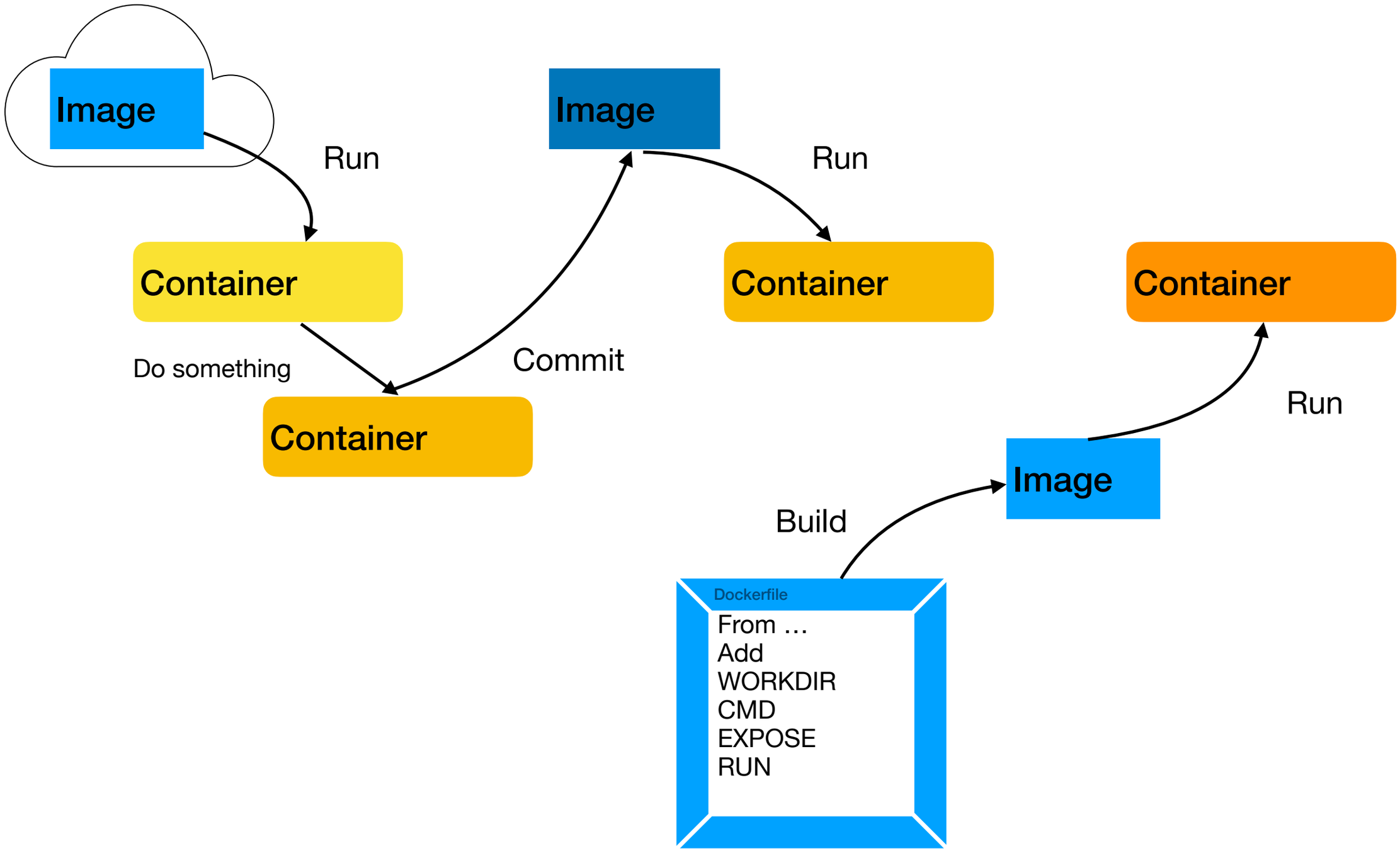
- A program that creates, ships and runs application containers
- Runs on any physical and virtual machine or server locally, in private or public cloud (Docker Host)
- Client communicates with Engine to execute commands

Architecture Overview Docker



Reference: https://docs.docker.com/engine/docker-overview/#the-underlying-technology

Image, Container, Dockerfile Relations



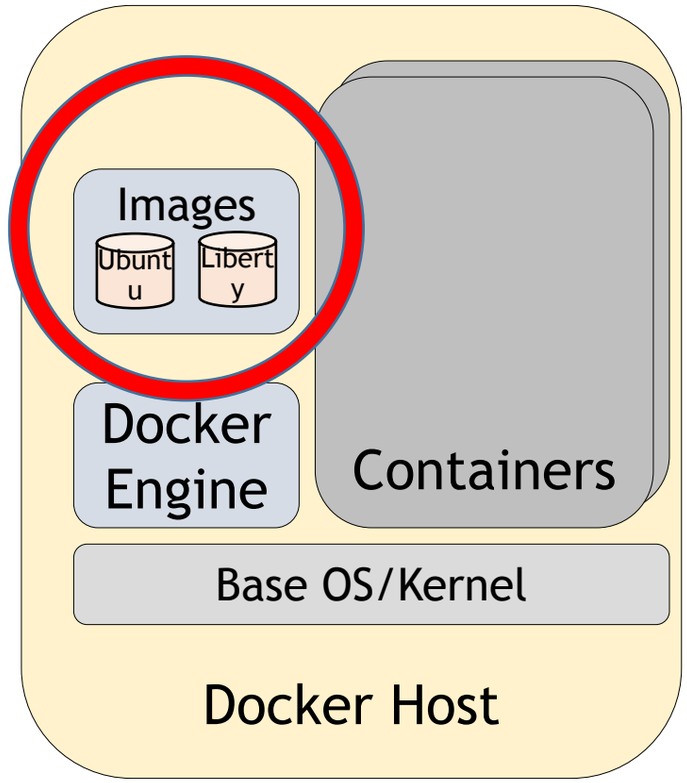
https://docs.docker.com/engine/reference/commandline/docker/

Examples

- *$ docker run -t -i node /bin/bash*
  - Fetches the public image containing node.js (named node) on top of a Debian Linux
  - -t allocates a pseudo tty, -i gives an inactive mode and the /bin/bash is the start command opening a bash
  - cat /etc/os-release (show that you are really working in a linux world (a Container in a VM running Linux on top of a Mac/Windows)
    - bwcloud : Host Linux = Ubuntu LTS 20.04
  - We start nodes (via the node command and enter some JavaScript Statements.
  - We can add a new file to this container
    - mkdir (new directory) and there cat > new.file (Return).. text (Return) <ctrl-c>
    - Exit this container
    - Docker container ls -a shows all containers (a container can be active or exited)
    - Snapshot can be built from both..
    - An exited container can be restarted with docker start -a -i <container#> (it takes the existing start command)
- Commit the changes is done by
  - $ docker commit -a Juergen.Schneider <container#> <name-of rep:tag>
    - -a is the author and name-of rep:tag the repository and tag (which is a the image name)
    - A image is named with the repository name and a tag (if missing it is :latest). Therefore a repository can have more than one differently tagged images.
  - The result can be reviewed via *$ docker image ls*
- docker run -t -i <name-of rep:tag>
  - The daemon remembers that the start command was /bin/bash and we are back in in this image you see the updates
  - The command docker run -t -i node /bin/bash would show the previous version

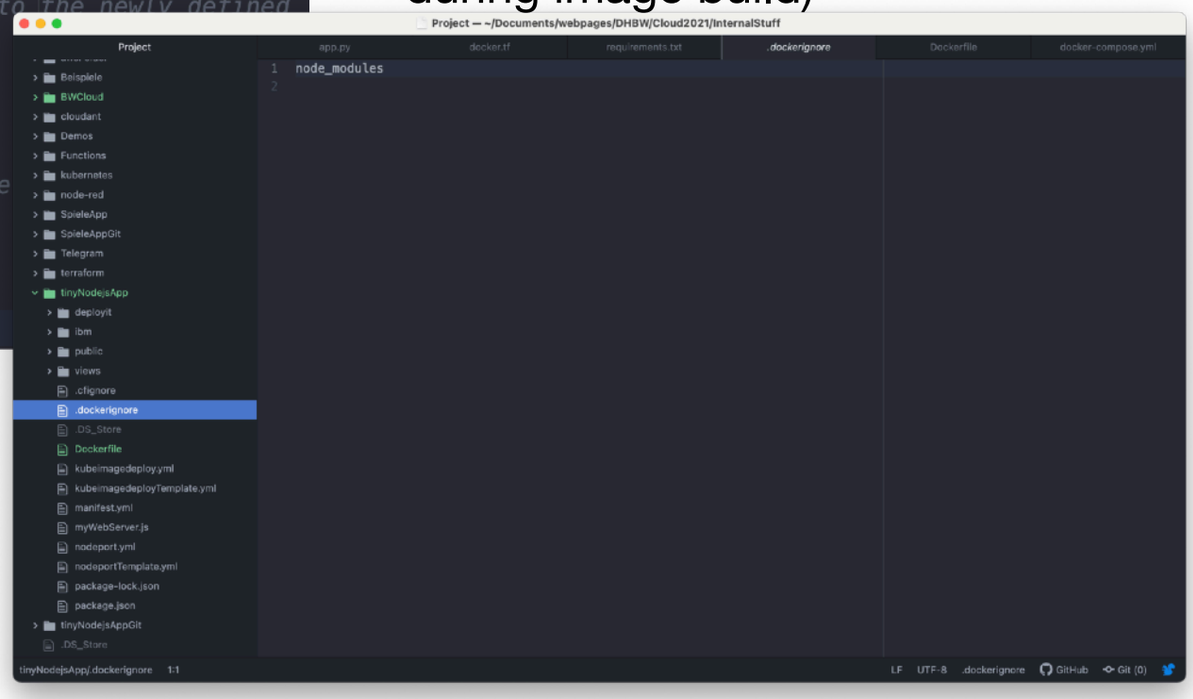
Docker Images

- Tar files containing a container filesystem + metadata
- For sharing and redistribution
  - Global/public registry for sharing: DockerHub
- Similar, in concept, to a VM image
- Docker can build images automatically reading the instructions from a dockerfile, a text file that contains all the commands, in order, needed to build a given image.
- https://docs.docker.com/v17.09/engine/userguide/eng-image/dockerfile_best-practices/#use-multi-stage-builds
-




```
# specify the node base image with your desired version node:<version>
FROM node
# Define the Work Directory including all the node.js stuff we need
#
WORKDIR /work
#
# we copy our current directory (from which the build command is executed) into the newly defined
# work Directory
COPY . ./
#
#
# This is the command which will be executed when the image is deployed in the
RUN npm install
EXPOSE 6002
CMD ["node","myWebServer.js"]
```

.dockerignore (to ignore these files during image build)



Dockerfile Instruction

- The dockerfile itself is named Dockerfile (no extension)
- You typically put this dockerfile into your working directory (the directory from which you mostly build your image)
- Important Dockerfile instruction
  - https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
  - https://docs.docker.com/engine/reference/builder/
  - FROM
    - The FROM instruction initializes a new build stage and sets the base Image for subsequent instructions. As such, a valid Dockerfile must start with a FROM instruction. The image can be any valid image – it is especially easy to start by pulling an image from a external (public) Repositories.
  - RUN
    - The RUN instruction will execute any commands (build an app) in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile.
  - CMD (ENTRYPOINT)
    - Is used to define the command which is executed when the container is started. Think about starting our Node.js or simple invoke a bash via /bin/bash. (CMD and Entrypoint are very similar, (CMD more run a Command, Entrypoint more run a Executable)both could be used in our cases)
  - EXPOSE
    - The Expose instruction informs Docker that the container listens on the specified network ports at runtime. The instruction does not actually publish the port. It functions as a type of documentation between the person who builds the image and the person who runs the container, about which ports are intended to be published. To actually publish the port when running the container, use the -p flag on the docker RUN.

Dockerfile Instruction (continued)

- COPY / ADD
  - The COPY instruction will copy new files from <src> and add them to the container's filesystem at path <dest>. ADD can also copy data from a URL and a compressed file
- WORKDIR
  - The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the DOCKERFILE. You are directed to the working directory when the container has been started and each command referring to an image directory can be marked as ./
- ENV
  - Set Environment variables in that container
- The command to build an image out of a dockerfile (in the directory you enter this command)
  - **docker build -t name-of-repository:tag .**
    - Consider the ‘.’

Image Layout

- https://github.com/moby/moby/blob/master/image/spec/v1.md
- A *Image* is an **ordered collection (layers) of filesystem changes**
- Layer
  - Images are composed of *layers (stacked images)*. Each layer (a single image) consist basically out of
    - The metadata for the layer as JSON describing
      - the image creation date,
      - author,
      - the ID of its parent image
      - The Command to start a container using this image
      - execution/runtime configuration like its entry point, default arguments, CPU/memory shares, networking, and volumes.
  - The filesystem changes described by a layer, relative to its parent layer (image).
  - other stuff

Layer of node image

```
"RootFS": {
    "Type": "layers",
    "Layers": [
        "sha256:ec09eb83ea031896df916feb3a61cefba9facf449c8a55d88667927538dca2b4",
        "sha256:053a1f71007ec7293dceb3f229446da4c36bad0d27116ab5324e1c7b61071eae",
        "sha256:a90e3914fb92b189ed9bbe543c4e4cc5be5bd3e7d221ef585405dd35b3e4db43",
        "sha256:5ab567b9150b948de4708a880cad7026665e7e0a227dea526cc3839eca4b2515",
        "sha256:d4514f8b2aac13e66dfc8b6e15aa2feb0d4ff942a192d9481f09b45e681ceb40",
        "sha256:7efd70d0bc369b94d8c5810e20609c9e4eadc0fdaa4deabc6df52174c73104a6",
        "sha256:bf55b14248a70f89f31efca6dbcb1de3b4376c82d1c5906f754060bd4f18f390",
        "sha256:d25cfc53cd3feda742c04ecc7b07e26a8ec1769f572bf05c218edfa2d4910349",
        "sha256:a229f6e7a6155e80a80ee2d3615fe482bd85adfe4f090e152e4d296ef81ffc6d"
    ]
},
```

Layer of node image + App

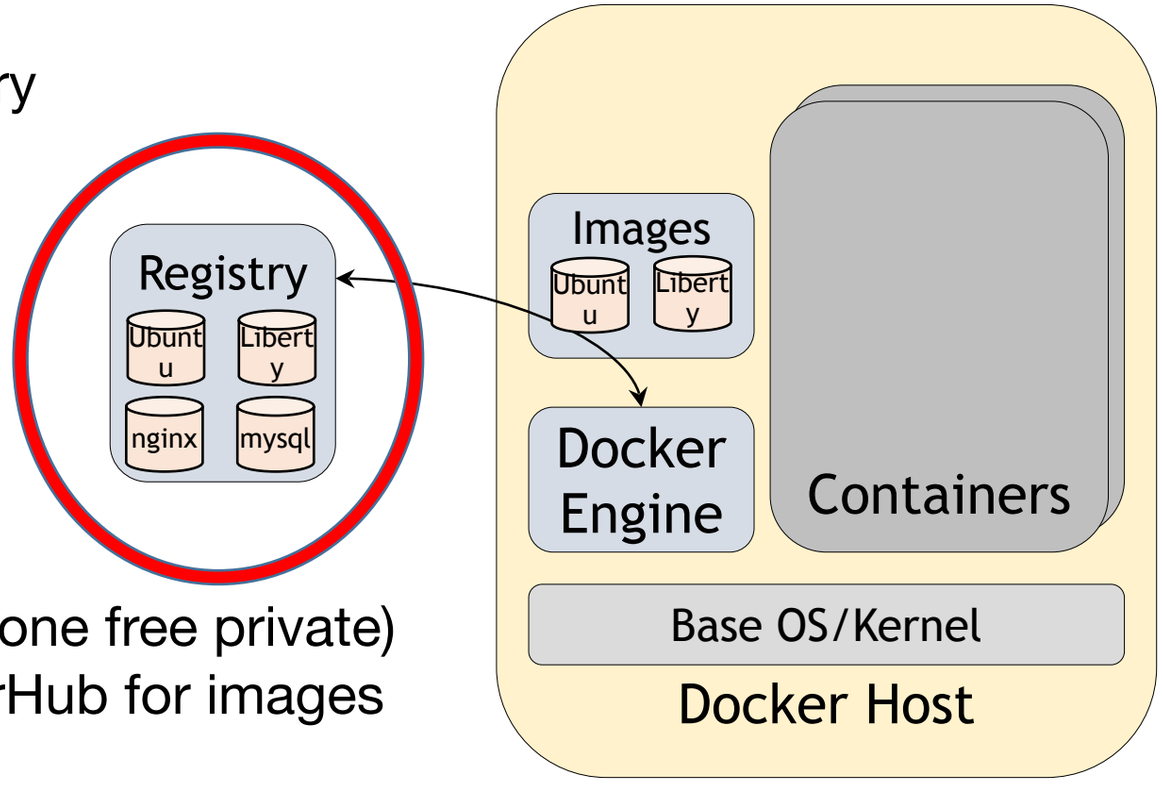
```
"RootFS": {
    "Type": "layers",
    "Layers": [
        "sha256:ec09eb83ea031896df916feb3a61cefba9facf449c8a55d88667927538dca2b4",
        "sha256:053a1f71007ec7293dceb3f229446da4c36bad0d27116ab5324e1c7b61071eae",
        "sha256:a90e3914fb92b189ed9bbe543c4e4cc5be5bd3e7d221ef585405dd35b3e4db43",
        "sha256:5ab567b9150b948de4708a880cad7026665e7e0a227dea526cc3839eca4b2515",
        "sha256:d4514f8b2aac13e66dfc8b6e15aa2feb0d4ff942a192d9481f09b45e681ceb40",
        "sha256:7efd70d0bc369b94d8c5810e20609c9e4eadc0fdaa4deabc6df52174c73104a6",
        "sha256:bf55b14248a70f89f31efca6dbcb1de3b4376c82d1c5906f754060bd4f18f390",
        "sha256:d25cfc53cd3feda742c04ecc7b07e26a8ec1769f572bf05c218edfa2d4910349",
        "sha256:a229f6e7a6155e80a80ee2d3615fe482bd85adfe4f090e152e4d296ef81ffc6d",
        "sha256:4f53ab4c2048a390e71b441445322dc599df35598afda5624cf04259ea506c42",
        "sha256:7bf9d0dd5a56315d1058adfcb6577d06f33630ed94af151aefc53386fa6501af",
        "sha256:9f4c518d93b495962d04334d2caf786ed40926ee89bdd387a18c63de283df4ab"
    ]
},
```

Build an Image of your Node.JS App

- Command to build a Image instructed by a Dockerfile
  - ***$ docker build -t <myapp> .***
    - Built an image named my app using the docker file on the current directory (**.**)
- Command to create a container based on the image above
  - ***$ docker run -d —name <myappcontainer> -p 7000:80 <myapp>***
    - Creates and runs a container using the image <myapp>:latest, port 80 (exposed in dockerfile) to which the node.js server is listening will be mapped to the host as port 7000 (which can be accessed externally via the host IP:7000
    - This (-d) runs in a detached mode (in background)
- Command to execute some Linux commands (to a detached container)
  - **$ docker exec -it <myappcontainer> any command**
- Command to list images
  - $ docker image ls
- Command to list active and exited containers
  - $ docker container ls -a
- Command to restart an exited container
  - $ docker start -a -i container-id
- All commands https://docs.docker.com/reference/cli/docker/

Docker Registry/Repository

- Creating and using images is only part of the story
- Sharing them is the other

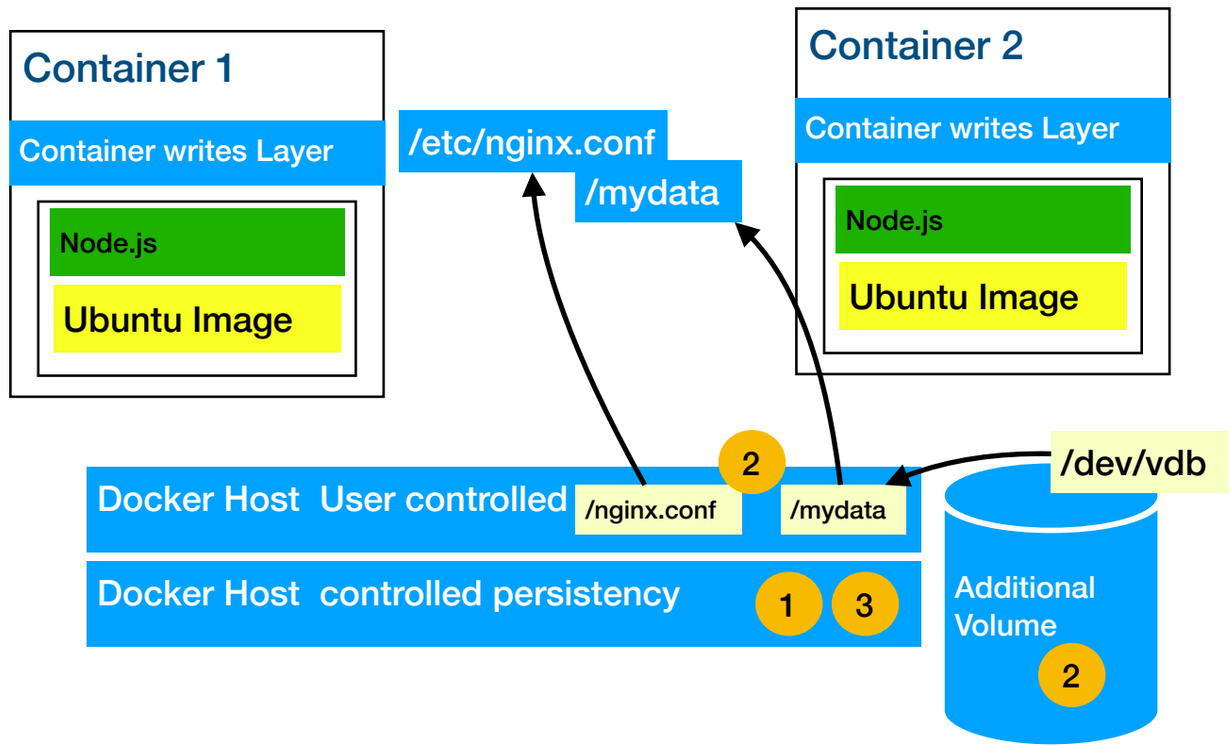


- DockerHub - http://hub.docker.com
  - Public registry of Docker Images
  - Hosted by Docker Inc.
  - Free for public images, pay for private ones (one free private)
  - By default docker engines will look in DockerHub for images
  - Web interface..
- Each Cloud provider has typically also a hosted registry.
- To push a local image to external public/private registry
  - docker login (local login)
  - docker tag imageId registry-domain/<my_namespace>/<my_repository>:<my_tag>
  - docker push registry-domain/<my_namespace>/<my_repository>:<my_tag>

Docker and Data Persistence

Container access to new and changed data

- **Internal**
  - Data as part of the image
- **External**
  - Data outside the Image as part of the host persistence (e.g VM Image of the Docker Host)
  - Additional Volume mounted to the Docker Host
  - External Directories (Storage) and Volumes **can be mounted** into the container



External Storage in Docker (defined as mounts, for the container)

1. As Docker Volume **(fully managed by Docker**)
   - Sharing only among containers within a Docker host
   - It is actually a mounted volume but the source is always in /var/lib/docker/volumes/ docker protected directory
2. As Mount-Bind
   - Like the Docker Volume but source can be anywhere (as the example) and is therefore not protected. (E.g a additional Volume can even be shared among different Docker hosts)
3. tmpfs
   - Like Volume but FS is in Docker Host memory, and could be used by the containers in that host.

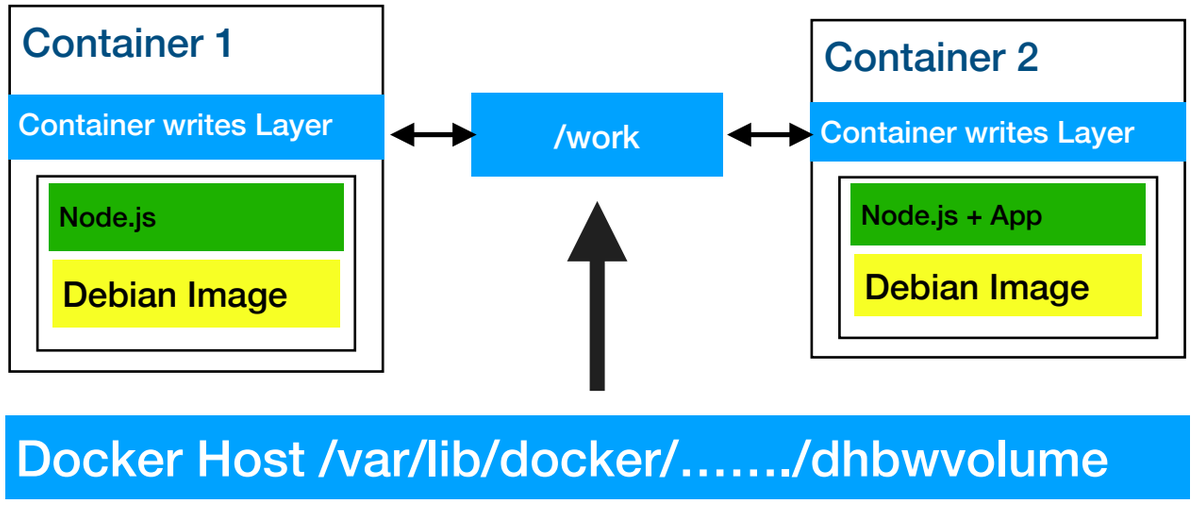
Use Cases

- Sharing data among containers (and as an external Volume sharing data among different hosts)
- Docker image updated data is gone (if not committed), when the container crashes or stops)
- Even more resilience when the real Volume is independent from the host
- Easier changes of configuration w/o the need to build a new image. (E.g. changes in /nginx.conf are available immediately in the containers)

https://docs.docker.com/storage/

Docker external Volume Example

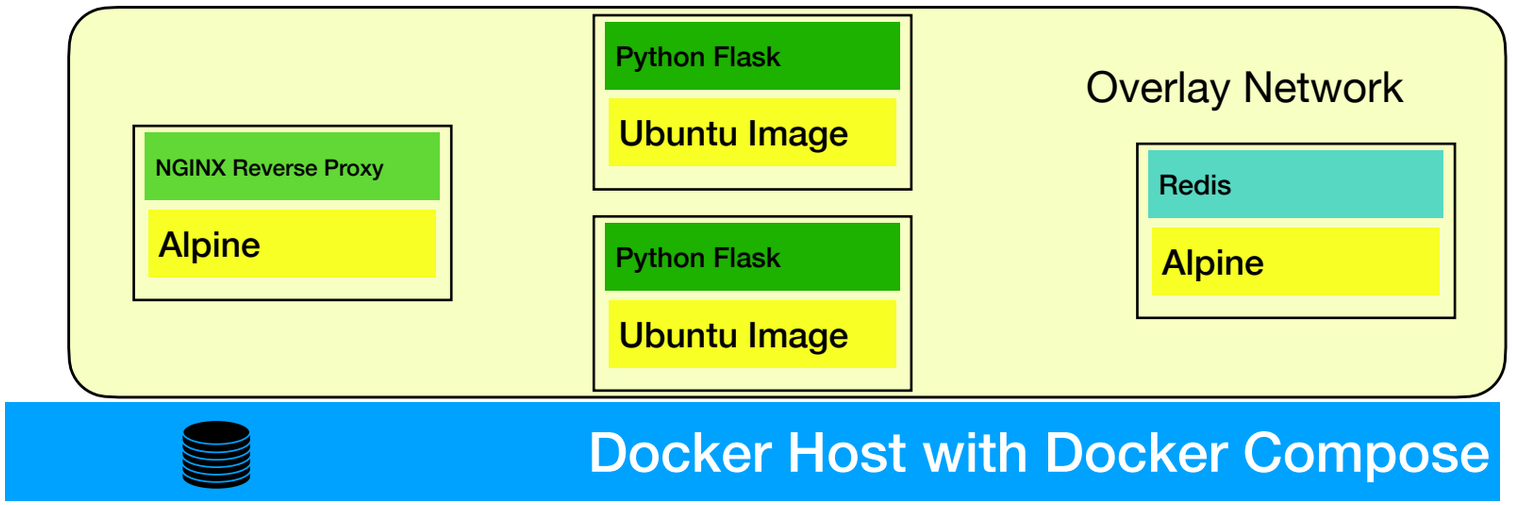
- Example of docker mounts
  - $ docker run -it **--mount type=bind,src=/home/juergen/work/python,dst=/share1** node /bin/bash
    The host directory /home/juergen/.. of the docker host is mounted into the container as /share1
    In case of the Docker Desktop (Mac). The local file system is part of the docker host file system, and therefore can be a source into the container)
- Docker Volume (Demo).. The Volume maps into a internal Docker Host Directory var/lib/docker…/dhbwvolume directory
  - **$ docker volume create dhbwvolume** // create a volume using a specific docker host directory
  - **$ docker run -it -v dhbwvolume:/work** <tinyappimage> /bin/bash // would mount this Docker volume to the container work directory and therefore would already have our data which has been installed in an earlier image build).
    - If the container mount point has data and the volume is empty, the container data is ‘copied’ to the volume (only for docker volumes)
    - If the volume has data and is newly mounted to an container it will hide the container data and show the volume data (true for both)
    - Now we could modify the work directory and with that modify the docker volume.
    - If the container stops and will be deleted the volume data remains
  - **$ docker run -it -v dhbwvolume:/work** node /bin/bash // although the work directory did not exist in the container it is visible now with all the data from the earlier tinyApp
- We can also use the desktop App to see the volume and data and within the container the data as well



Docker Compose

Manage several containers in one Docker host as a Service.

- Definition of all components of a Service in one file (.yaml)
- Start / Stop of all container belonging to a certain service with one command



- The Network of such a service is an internal overlay network (build by docker-compose)
  - each container in that network can join it and reach each other container via specified hostname (e.g. lb, web, db) and port (flask: 5000, redis: 6379)
  - ports can be specified either as internal (part of the overlay) and/or external to the service (e.g. important for the web front, in our case NGINX)
- Image can be built or used during Compose Start-Up
- Startup dependency can be specified
- Each container must be defined individually (e.g. 4 above, therefore 4 hostnames lb, web1, web2, db)… no implicit load balancing (in above example nginx does the load balancing to either web1 or web2)… see Swarm mode later
- External volumes for specific configurations desirable (e.g. nginx.conf)

- https://docs.docker.com/compose/

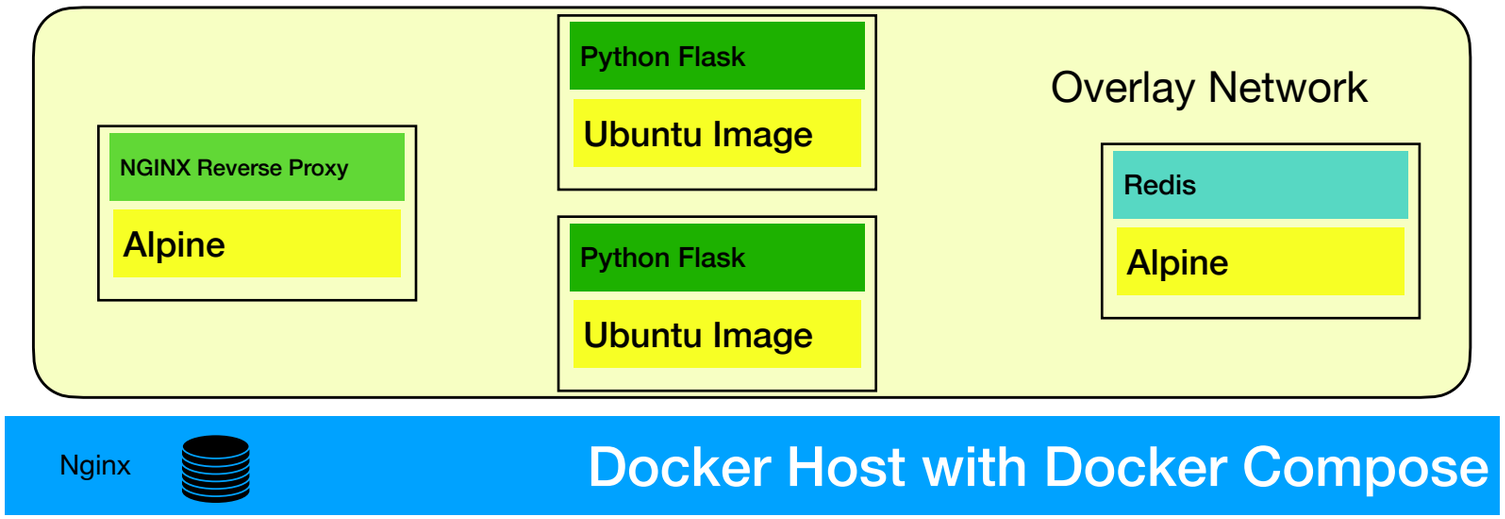
YAML = YAML ain't markup language

Depending on whom you ask, YAML stands for **yet another markup language** or YAML ain't markup language (a recursive acronym), which emphasises that YAML is for data, not documents. YAML is a popular programming language because it is designed to be easy to read and understand.

Docker Compose Configurations

nginx.conf - load-balancing

```
events {
  worker_connections  1024;  ## Default: 1024
}
http {
   upstream myapp1 {
      server web1:5000;  # the internal container port
      server web2:5000;  # the internal container port
    }

    server {
          listen 80;
          location /home {
          proxy_pass http://myapp1;
        }
    }
}
```



Docker-compose.yaml

```
version: "3.9"
services:
  lb:
    image: nginx:latest
    container_name: loadbalancer_nginx
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf # this references the host file nginx.conf to the image /etc/nginx/nginx.conf
    ports:
      - 80:80
      - 443:443
  web1:
    build:
      context: .
      args:
        InstanceName : web1
    container_name: web1  # internal port Nummer ist 5000 dockerfile expose
  web2:
    build:
      context: .
      args:
        InstanceName : web2
    container_name: web2  # internal port Nummer ist 5000 dockerfile expose
  db:
    image: "redis:alpine"
```

https://docs.docker.com/compose/compose-file/compose-file-v3/#configs-configuration-reference

Python Code - Accessing Redis

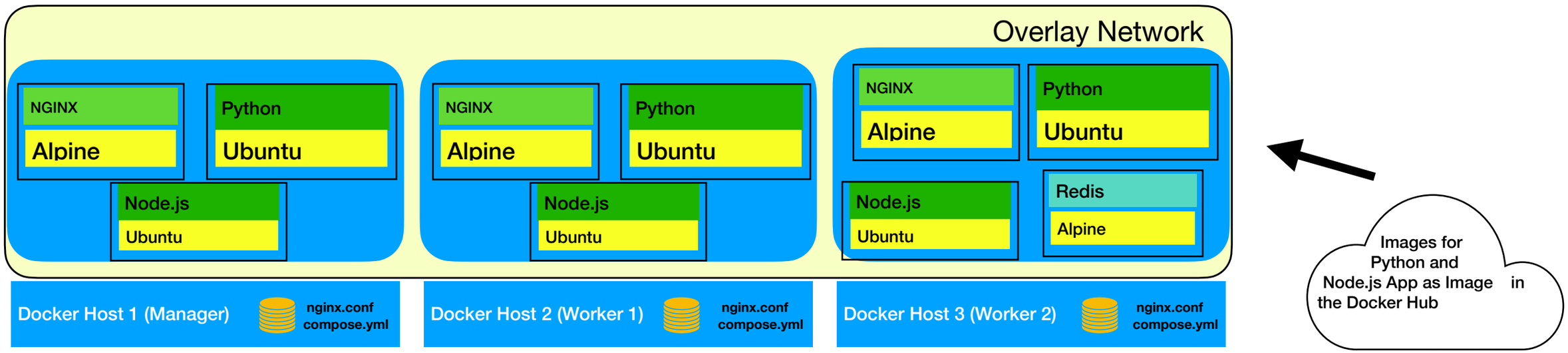
```
import datetime
import redis
import string
import random
import os
from flask import Flask , request

app = Flask(__name__)
cache = redis.Redis(host='backenddb', port=6379)
start = datetime.datetime.now()
```

```
in app folder (mit docker-compose.yml) $ docker compose up
Network overlay -> $ docker network inspect pythonapp_default
```

Docker Swarm

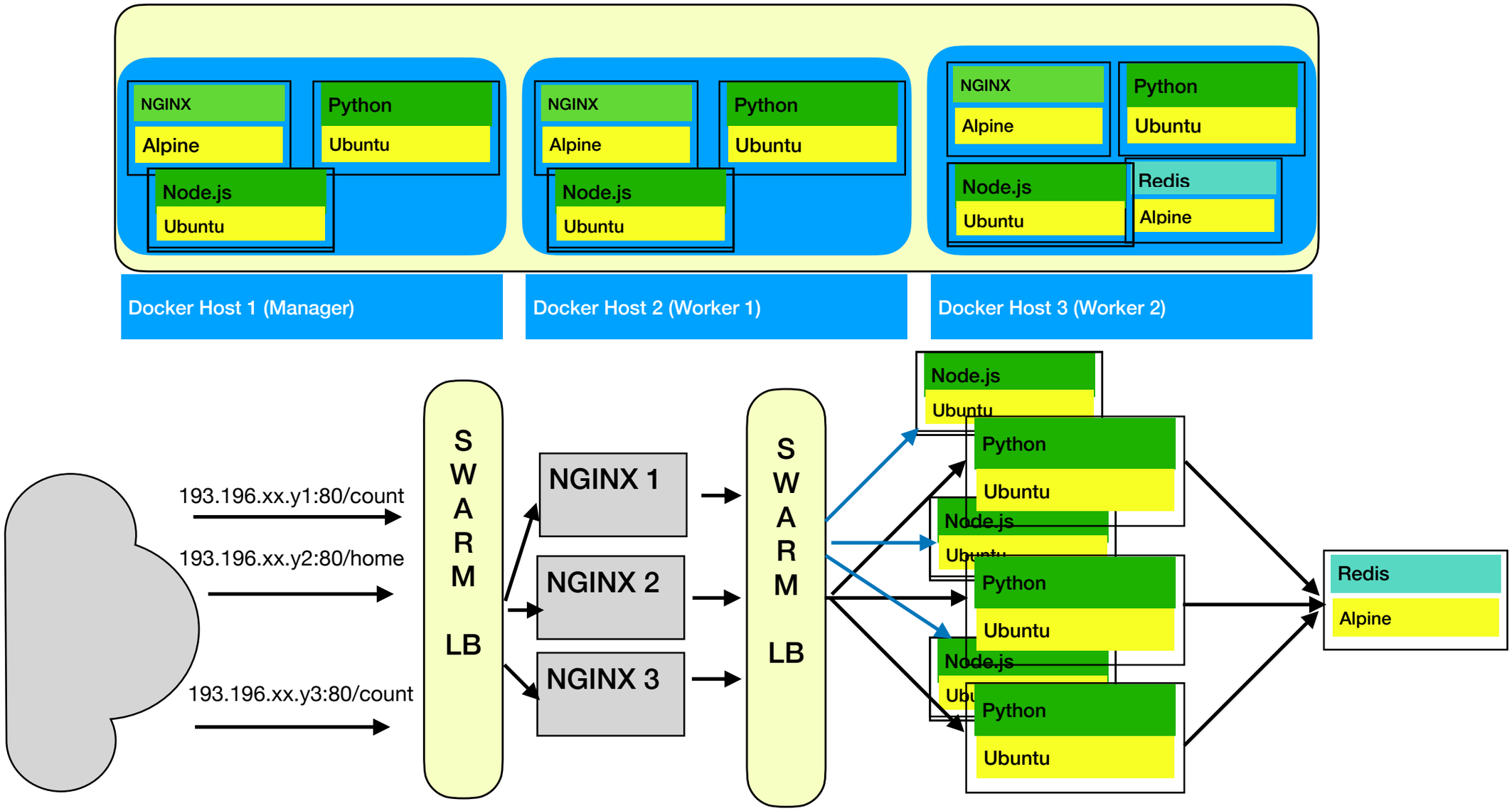
Manage many containers **distributed over more than one Docker host** as a Service (named stack).

- Definition of all components of a stack in one file (.yaml) (the components are named service)
- Start / Stop of the stack



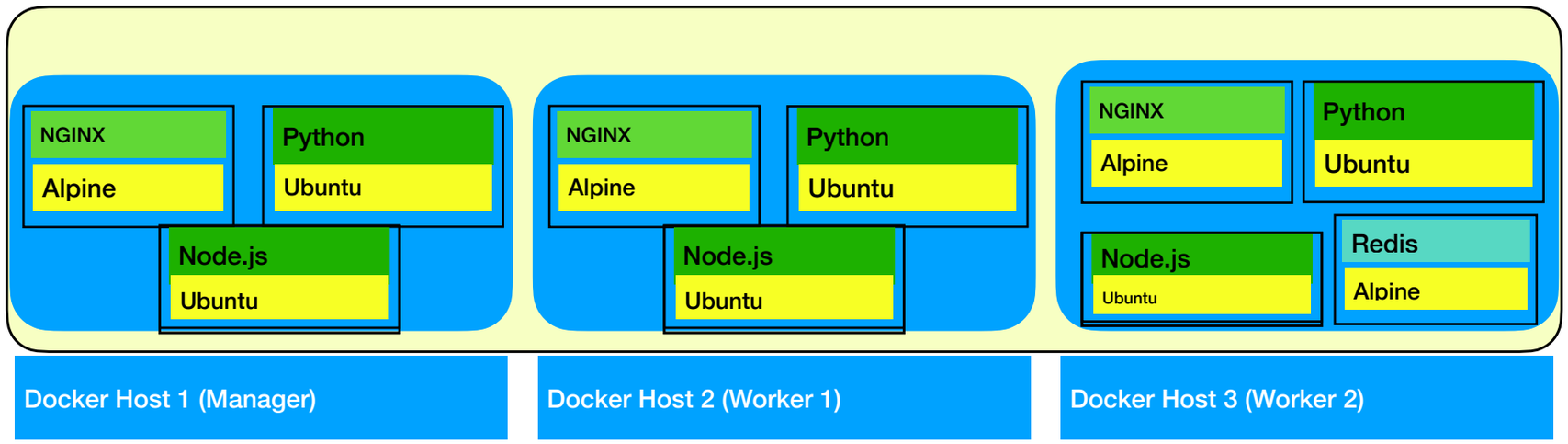
- Docker host acts either as Swarm Manager and or Swarm Worker
  - Worker host the container belonging to a Service, Manager takes the additional burden to administer the Swarm
- The Network of all services in a stack is a internal overlay network (build by Swarm Manager, like Docker Compose)
- No Image build during startup (Image must be ready)
- Startup dependency can be specified
- External volumes for specific configurations desirable (e.g. nginx, Dockerfile, dockerswarm-app.yaml etc)
- Important **Deployment Options** (per service part e.g. nginx, Python, Redis)
  - **Mode** : **Global** = always 1 on each node (independent of the # of nodes) OR **Replicated** with a certain number of services over the Swarm (always the replicated # independent of the # of nodes)
  - **Replicas** : when Mode is replicated, here we specify the # of container per service should be active in the cluster
  - **Constraints :** only on specific nodes (e.g redis)
  - Preferences : spread over a subset of nodes … (e.g equal split over datacenter)
  - Resource capacity constraints (CPU and Memory Limit)
  - **Restart** : when to restart, how often to restart until consider a problem, etc..
  - Rollback : how the service should be rollbacked in case of a failing update.
  - Update : how the service should be updated (parallel, in sequence, when to roll back)
- Many things more…..
- https://docs.docker.com/compose/compose-file/compose-file-v3/

Docker Swarm Load Balancing



- NGINX acts merely as http proxy (application Level LB)
  - Load balancing from external requests to NGINX is done by Docker Swarm
  - Load balancing from NGINX to the python or node.js Service (with 3 instances) is done by Docker Swarm
  - Access to redis is done via overlay network backenddb:port (as in compose, but adding a hostname in the stackyaml)
  - We have 3 external open ports (port 80 on each system). This could be used by an cloud based external Loadbalancer (Cloudflare) and the different docker hosts could live in different physical locations
  - We could also remove the nginx Balancer and use the SWARM Balancer with port 5000 and 80, however than we would loose our ALB capabilities
- https://docs.docker.com/compose/compose-file/compose-file-v3/

Docker Swarm Demo Scenarios



- Case 1 -> Run Service on all three public IP addresses
- Case 2 -> Pause Worker 1
  - See the movement of the service to the remaining nodes to keep the # of replications
  - Consideration :
    - no movement after resuming (worker 1 remains empty), however external access port 80 runs again.
- Case 2 -> Pause Worker 2
  - See the movement of the service, redis will not move and the service becomes unavailable
  - Considerations : after the resume redis comes up again

Service Registry/Discovery = critical Feature for supporting MicroServices

Docker Swarm Commands

at Swarm Manager:

- **docker swarm init** // dann die worker joinen lassen commando wird angezeigt.
- **cd /mydata** // dort liegen alle Daten
- **docker stack deploy -c dockerswarm-app.yml juergenapp** // muss unbedingt juergenapp sein damit nginx config stimmt
- **docker service ps juergenapp_web** // zeigt wie die einzelne Teile (Services) auf den Servern verteilt sind
- **docker stack ps juergenapp** // zeigt alle services einer app und Ihrer Verteilung mit Historie

Docker Container Recap

- Operating Systeme sind in der Lage unterschiedliche Anwendungsprogramme von einander zu isolieren
  - Isolierung im Sinne der Security und im Sinne der Betriebsmittelzuweisung
- Daher ist es sinnvoll das Operating System selbst zu benutzen, da die klassischen Hypervisor (Hyper-V, VMware etc) komplette Operating System virtualisieren.
- Container ist der technische Begriff dieser Linux basierten Isolation. Über diese Container haben die Anwendungen einen isolierten Zugriff auf alle notwenigen Betriebsmittel. Die Illusion alleine auf einer bestimmten Linux Version zu laufen obwohl die Anwendung im Grunde den Host OS Kernel teilt.
- Docker als Firma hat sehr stark auch zur ‘Managebility’ und Nutzung dieser Container beigetragen.
- Die Nutzung der Container gilt dem klassischen Virtualisieren (Hyper-V) als überlegen weil das Aufsetzen dieser Container sehr schnell geht und weil man mehr Anwendungen über Container auf einen ‘Host’ packen kann. (e.g. Kernel Daten nur einmal da)
- Docker Images sind im wesentlichen aufgestapelte Schichten (Layers) von Filesystem Änderungen in Bezug auf die drunterliegende Schicht. (neu, verändert, gelöscht). Damit wird unnötige Redundanz vermieden
- Die Nutzung eines Union Filesystem Drivers erlaubt das effiziente “Mounten” aller notwendigen Layer beim Start eines Containers
- Werden durch die App im Container Daten verändert, werden diese Änderungen in einem neuen Top Layer vermerkt (Write Layer), alle über das Image reingekommenen Layers sind read only. Man kann aber diese aufgelaufenen Änderung wieder als Image Snapshot speichern (Commit) und könnte aus diesem neu entstandenen Image einen neuen Container aufbauen.
- Nach diesem Prinzip entstehen die Docker Images. Von einer zu ladenden Basis werden durch Veränderung neue Layer draufgesetzt die ich dann als ‘read only” layer ‘commite’
- Dieser Veränderungsablauf kann ich über eine Dockerfile automatisieren
- Docker (und andere) bieten sehr viele Standard Apps über fertige Images an (Image:pull) öffentlich an. Solche Hubs können auch per Service für Unternehmungen gegen Gebühr bereitgestellt werden. Man kann selbst eigene Image öffentlich anbieten (Docker Account) und sich ein Lob einheimsen.
- Für jedes dieser Images oder auch den eigenen Images wird per Default ein Vulnerability Scan gemacht und mögliche Risiken angezeigt.

Docker Container Recap

- Das Managen einzelner Container wird bei Anwendung viele Container mühselig. Man möchte alle Container die zur einer mult-tiered App gehören als eine Einheit verwalten. Dafür gibt es Docker Compose.
  - In Docker Compose werden alle Anwendungsteile in eine .yml File (abstrakt und declarative) beschrieben. Welche Images werden benötigt ? Muß ich eigene Images bauen ? Wo stehen die App Konfigurationen und Environment Daten ? Wie wird untereinander und/oder extern kommuniziert. Außerdem können Aktivierungsabhängigkeiten formuliert werden.
  - Docker Swarm ist die logische und notwendige Erweiterung um solche containerized Apps auf unterschiedlichen Host (in unterschiedlichen DCs) zu verteilen. Zu dieser Verteilung gibt es weitere Optionen in der .yml die angibt wieviel und wo im Swarm diese Container (mit den App Teilen) verteilt werden. Bei Ausfall eines Containers oder eines Hosts im Swarm wird versucht durch Umverteilung das gesetzte Ziel wieder zu erreichen.
  - Service Discovery ist der allgemeine Begriff der das “Wiederentdecken” von Apps (die über IP/Port kommunizieren) zu ermöglichen.
- Anwendungen produzieren Daten (e.g. DB) und benötigen Daten (e.g. Konfiguration). Dafür gibt es nun verschiedene Möglichkeiten solche Daten zu persistieren
  - Daten können Teil des Images sein (Lesen) und neue Daten können im ‘writeable’ top Layer geschrieben werden. Die neu geschriebenen Daten sind beim Crash oder Löschen des Containers weg. Außerdem muss ich bei Änderungen der App Logik sowie der Konfiguration immer ein neues Images bauen.
  - Daten können extern zum Container angelegt werden (Lesen und Schreiben). Diese Volumes gehören im Grunde dem Host. Damit würden sie den Container Crash oder das Löschen überleben. Außerdem könnte ein externes Volume auch einen Host Crash überleben.
  - Wichtig: solche externe Volumes können dann auch zum Datenaustausch zwischen den Container genutzt werden
  - Es gibt folgende Ausprägungen eines externen Volumes.
    - Externes Volumes wird vom Docker Host vollständig kontrolliert, nur für Austausch zwischen den Container geeignet. Oder ein TempFS als Teil des Docker Hosts.
    - Externes Volume ist ein Docker Host Mount Point der vom Benutzer kontrolliert werden kann. Damit geeignet schnell App oder Konfiguration Änderungen von extern einzuspielen ohne ein neues Image bauen zu müssen.

What is Kubernetes?

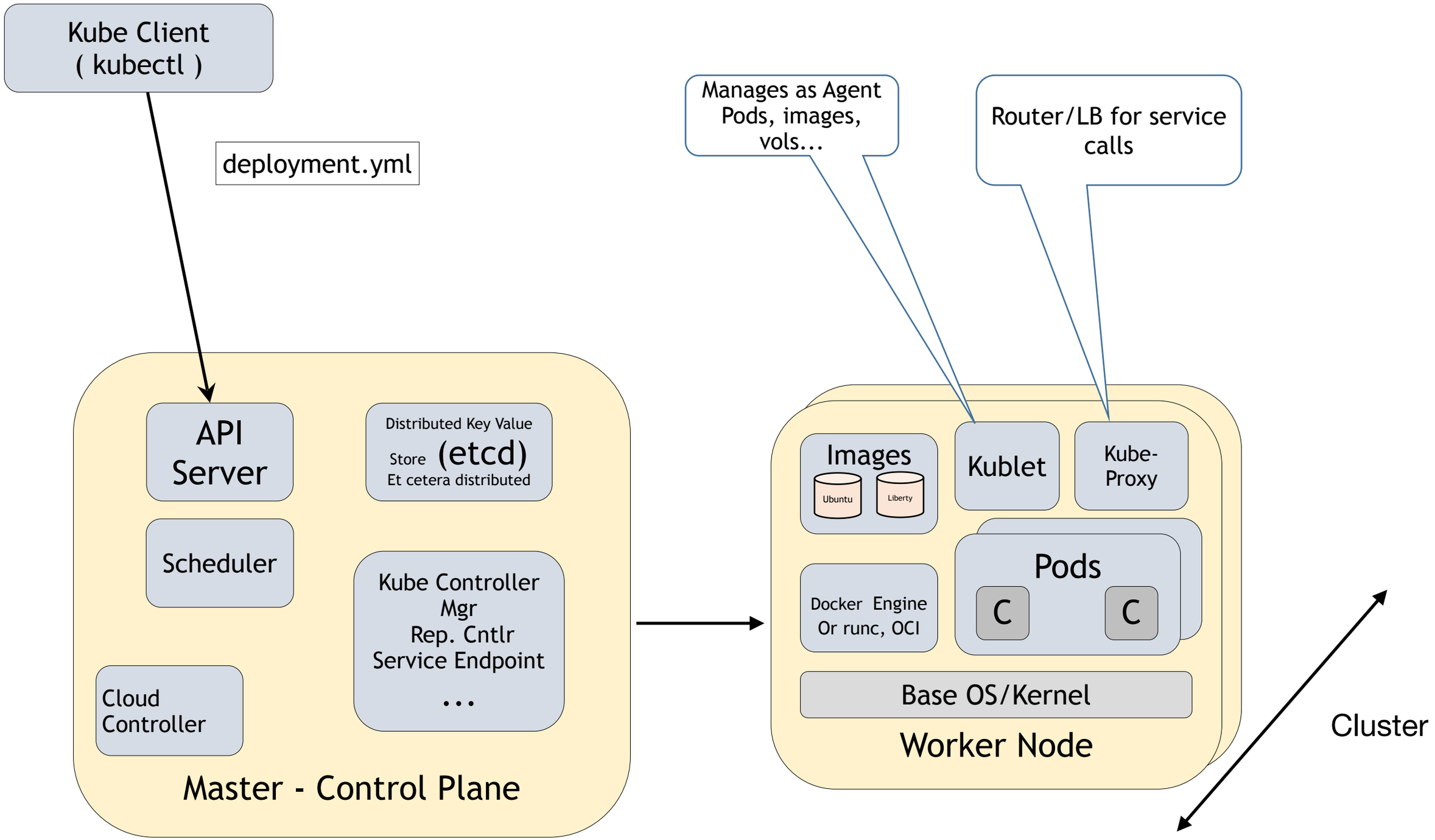
- Kubernetes - K8s
  - “Helmsman" (Steuermann / Steuerfrau) in ancient Greek

- Container Orchestration
  - Provisions **apps, services**, deployments, vols, nets, etc... **with a desired state**
  - Kubernetes then tries to align the system to that desired state

  - Similar to Docker's Swarm
  - But was there first and does so much more (?)

- FYI: Kubernetes == K8s == Kube
- Open Source
  - http://kubernetes.io
  - Started by Google, Initial release June 2014
  - Now part of the CNCF (Cloud Native Computing Foundation)
    - Moving K8s to an open governance model
    - Large/wide community
      - While still heavily controlled by Google they're trying to shift that

Kubernetes Components



https://kubernetes.io/docs/concepts/overview/components/

Container n — 1 Pod n — 1 Node n — 1 Cluster

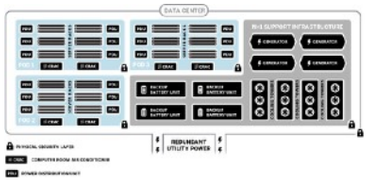
Worker Nodes, Worker Pool (IBM and others based on different HW flavours)

K8s Components

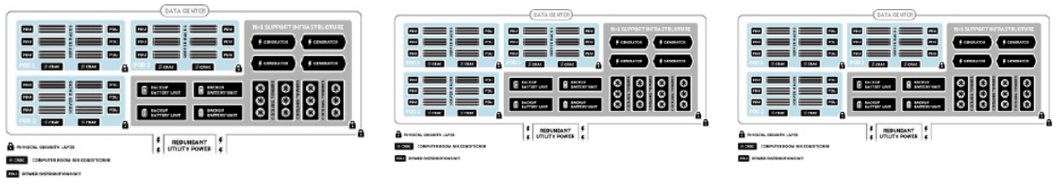
- Control Plane (Master)
  - API Server
    - exposes the Kubernetes API. It is the front-end for the Kubernetes control plane.
  - etcd
    - Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.
  - Scheduler
    - watches newly created pods that have no node assigned, and selects a node for them to run on.
      - Factors taken into account for scheduling decisions include individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines.
  - controller-manager
    - runs controllers (nodes, replications, security tokens, network service endpoint (Discovery) and more
      - Cloud Controller (Interface to cloud specific implementation of network, nodes and storage, e.g. Interfaces to Open Stack)
- Data Plane (Worker Nodes)
  - Kubelet
    - An agent that runs on each node in the cluster.
    - The Kubelet takes a set of PodSpecs and ensures that the containers described in those PodSpecs are running and healthy.
  - Kube-Proxy
    - enables the Kubernetes service abstraction by maintaining network rules on the host and performing connection forwarding.
  - Container runtime
    - is the software that is responsible for running containers. Kubernetes supports several runtimes: Docker, rkt, runc and any OCI runtime-spec implementation.

Kubernetes Physical Layouts

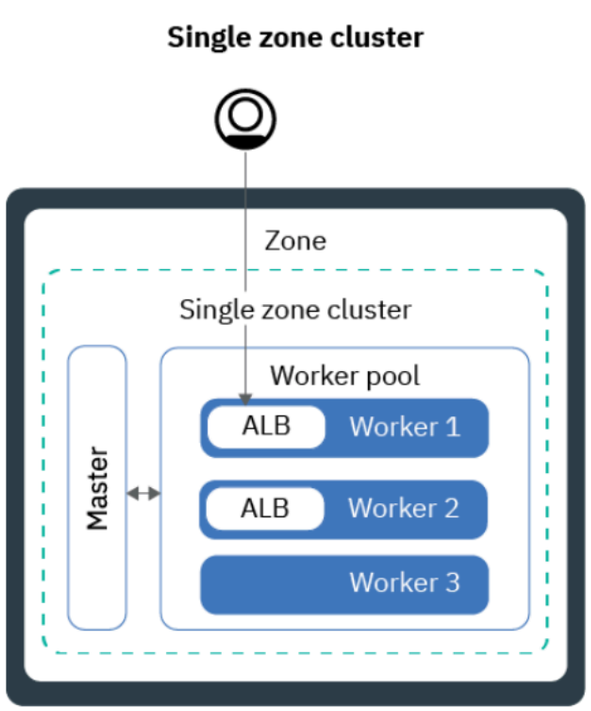
Availability Zone (AZ = DC), a Region is a set of AZs within a area of less 50 km (diameter e.g Frankfurt Region)

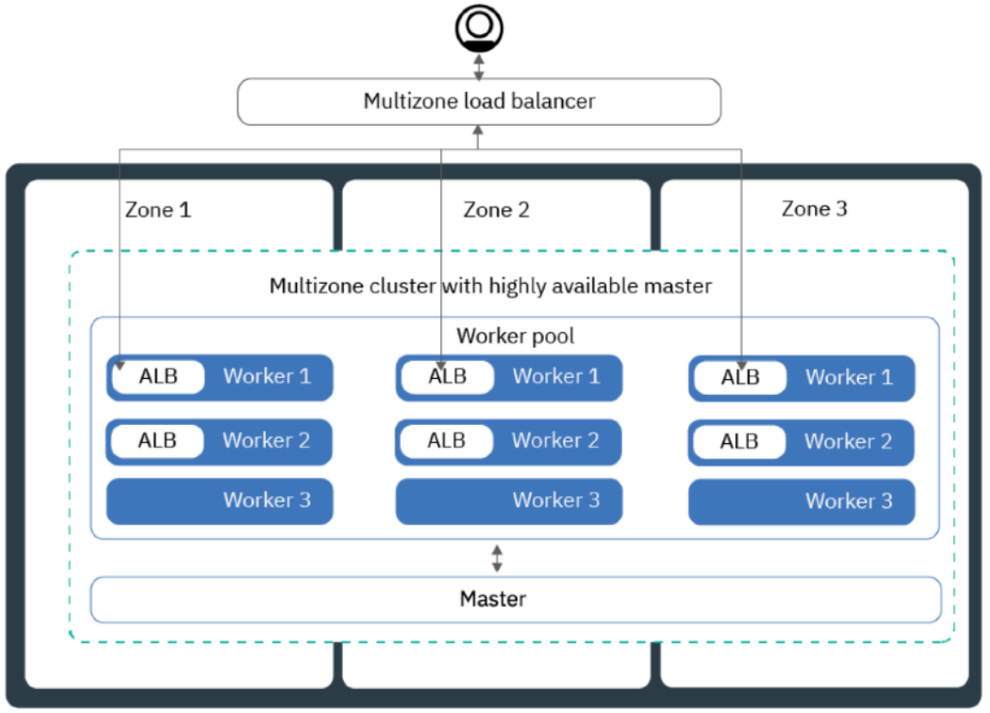


1 AZ no Region



3 AZ per Region, potentially reach 99.95 % Availability

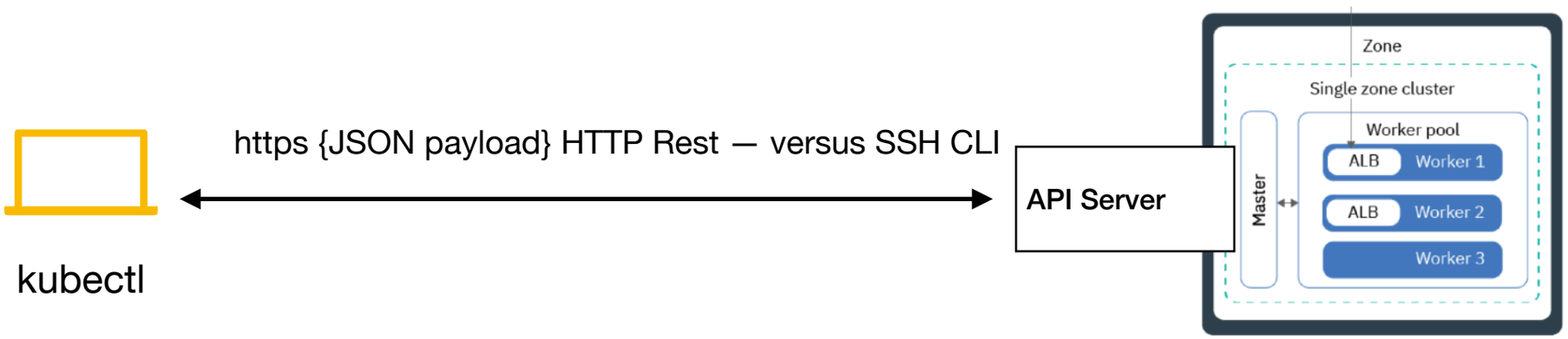




ALB = Application Load Balancer (HTTP Level Balancer, e.g NGINX)
Zone Load Balancer is basically a global LB (TCP Level Balancer)

K8s Objects

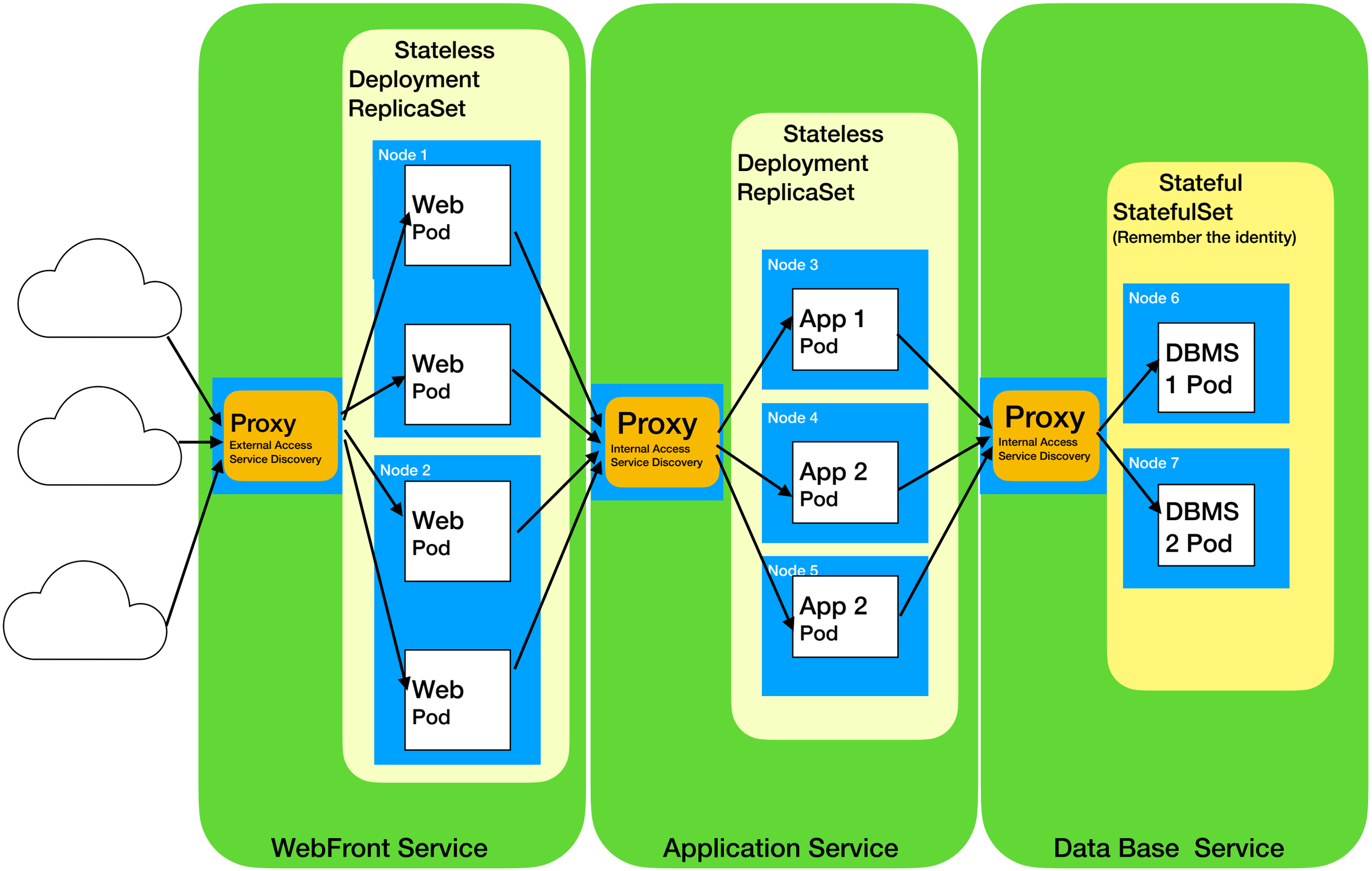
- *Kubernetes Objects* are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of the cluster and all his entities. Specifically, they can describe:
  - What containerized applications (pods) are running (and on which nodes)
  - The resources available to those applications (e.g. Docker images, ports, network access)
  - The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance
- A Kubernetes object is a "**record of intent**"–once you create the object, the Kubernetes system will constantly work to ensure that object exists. By creating an object, we're effectively telling the Kubernetes system what we want as this is your cluster's **desired state**. (declarative approach)
- The Objects current states can be queried using the API, GUI or the Kubectl CLI (kubectl get…)
- Objects are described as YAML or JSON file and referred in Kubectl CLI



K8s Objects

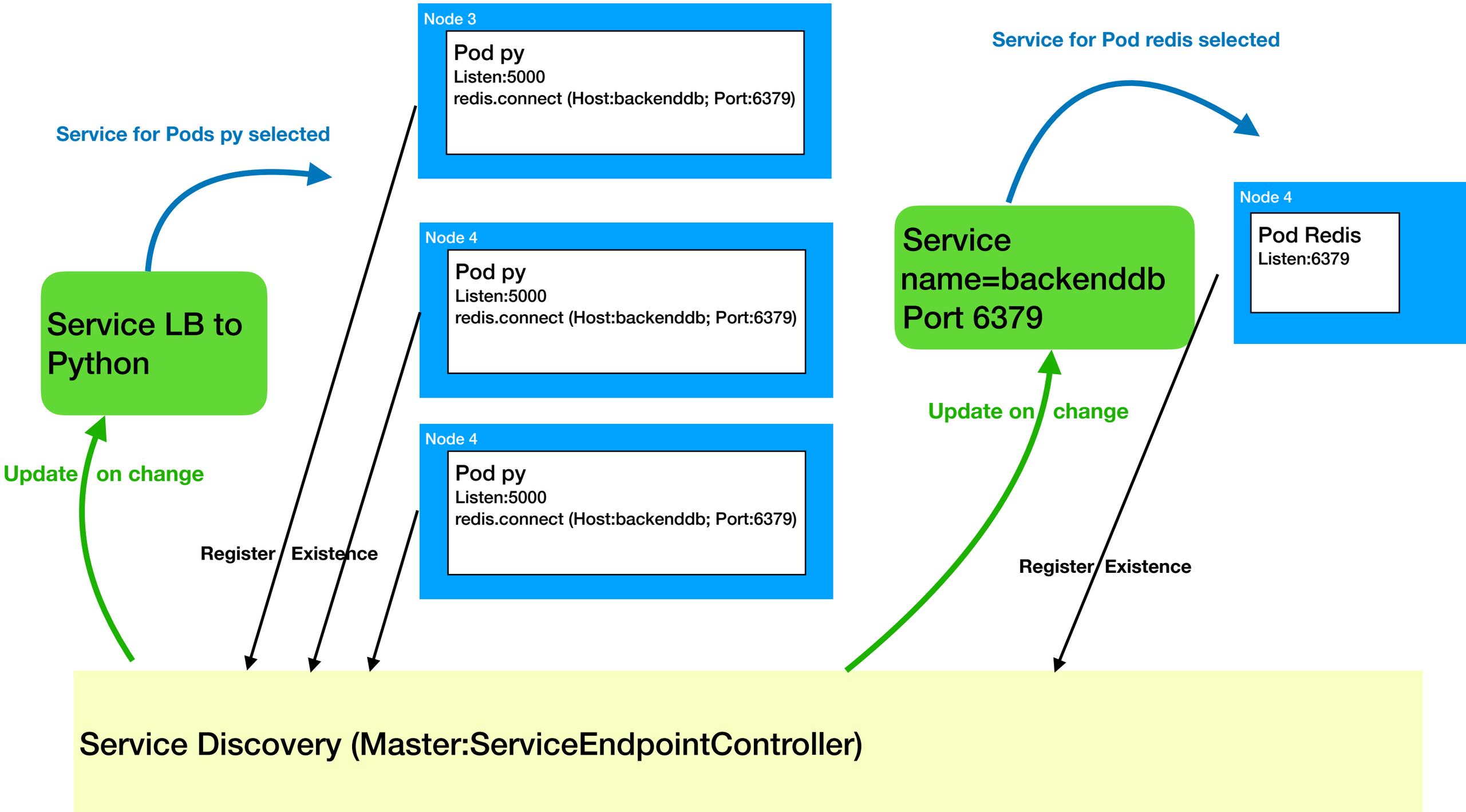
- Objects are used to describe
  - A **Pod** (Hülse) one or more Containers, including the storage attached, the image, and a unique internal IP
  - A **Service** is way to define a logical set of Pods which build an application tier and should be network accessed transparently in cases of movements and scale up and down. **(Service Discovery)**. A Service can also have a IP based external access as a whole.
  - A **Volume** is a directory, lifecycle managed outside the container/pod
    - Worker local Storage
    - Worker mounted external storage
- A **ReplicaSets** ensures that a specified number of stateless pod replicas are running at any given time. (e.g. is part of the Deployment YAML)
    - A **Deployment** is the object which defines the deployment of Pods including the Replicas the exposed network Interface and the Volumes. It support many options around Rollback etc. It is the major entity we work with.
- **Daemon Sets** are Pods which are started on each worker node
- **Stateful Sets** are Pods which typically run stateful apps
- …

Example Multi-tiered Application



Service Discovery

- Modern distributed applications run in containerized environments where the number of instances of an app tiers and their locations changes dynamically.
- Kubernetes supports Services Discovery w/o the need of Application code to be aware of such location and intense number changes.



Working with Objects

- All objects in the Kubernetes REST API are unambiguously identified by a **Name** and a **UID**.
- **Namespaces** provide a scope for names (in its natural sense). With Namespaces we conceptually build logical clusters backed by a physical cluster. They are used to organise resource separations among **different users**, **or projects using sets of Pods.** For example using **Namespaces**, we can limit the quota to each namespace for resources utilization
- **Labels** are key/value pairs that are attached to objects, such as pods. Labels are intended to be used to specify identifying attributes of objects that are **meaningful** and **relevant to users within a namespace** (not to the system). Labels can be used to organize and to select subsets of objects. (e.g. release:stable; release:beta). There are also pre-defined labels. **Selectors** can be used in commands to **sub-select objects** according labels. Those labels are executed by Kubernetes.
- **Annotations** additional meta data (key/value pairs) **not** as selector or filters but used by external tools (e.g monitoring, logging, or just documentation)
- **https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/**

K8s CLI (Operation Mode)

- K8s CLI (or REST) can be considered in three modes (sometimes confusing)
  - **Imperative commands**: (everything as part of the command)
    - e.g. kubectl run : Create a new Deployment object to run Containers in one or more Pods. (everything needed is part of the command)
    - e.g. kubectl expose : Create a new Service object to load balance traffic across Pods.
  - **Imperative object configuration** (Operation in Commands, Objects in Files)
    - the kubectl command specifies the operation (create, replace, delete, get), optional flags and at least one file name. The file specified must contain a full definition of the object in YAML or JSON format.
      - Problem: Updates to live objects (as above) must be reflected in configuration files, or they will be lost during the next replacement.
- **Declarative object configuration** (Everything on a local directory)
  - The **kubectl apply -f** (file), operates on the object configuration file stored locally, however the user does not define the operations to be taken on the file. Create, update operations are automatically detected per - object by Kubectl.
  - https://ralph.blog.imixs.com/2020/07/26/kustomize-your-kubernetes-deployments/

Juergen's approach

- Define all you need (deployment, replica, service) in a YAML/ JSON File
- Use ***kubectl apply*** *-f mydeployment.yml* to initially deploy and run changes
- Use ***kubectl get*** *-f mydeployment.yml* to see the current configuration
- Use ***kubectl delete*** *-f mydeployment.yml* to delete the current configuration
- The CLI https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#-strong-getting-started-strong-

Example : Stateless App Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name:  <name of the object>
  namespace:  <namespace of the deployment object>
  annotations:
    key :  value
  labels:
    key : value
spec:
    replicas: 3  <number of Pod instances desired>
     selector:     <who are my Pods>
     matchLabels:
       app: tinyapp
     template:   <template for creating Pods>
       metadata:
         labels:
           app: tinyapp    <must match with the selector above>
        spec:
          containers:
            - name:  <container name>
                image: juergenschneider/ultimativeapp:dhbw
                imagePullPolicy: Always
---
```

```
apiVersion : v1
kind: Service
metadata:
   name:  <name of the object>
   namespace:  <namespace of the service object>
  annotations:
    key :  value
  labels:
    key : value
 spec:
    selector:
       app: tinyapp
    type: <type of service>
    ports:
      - port: <the service port>
        targetPort: <the container port>
        protocol: TCP
```

https://kubernetes.io/docs/reference/kubernetes-api/workload-resources/deployment-v1/

Kubernetes Dokumentation

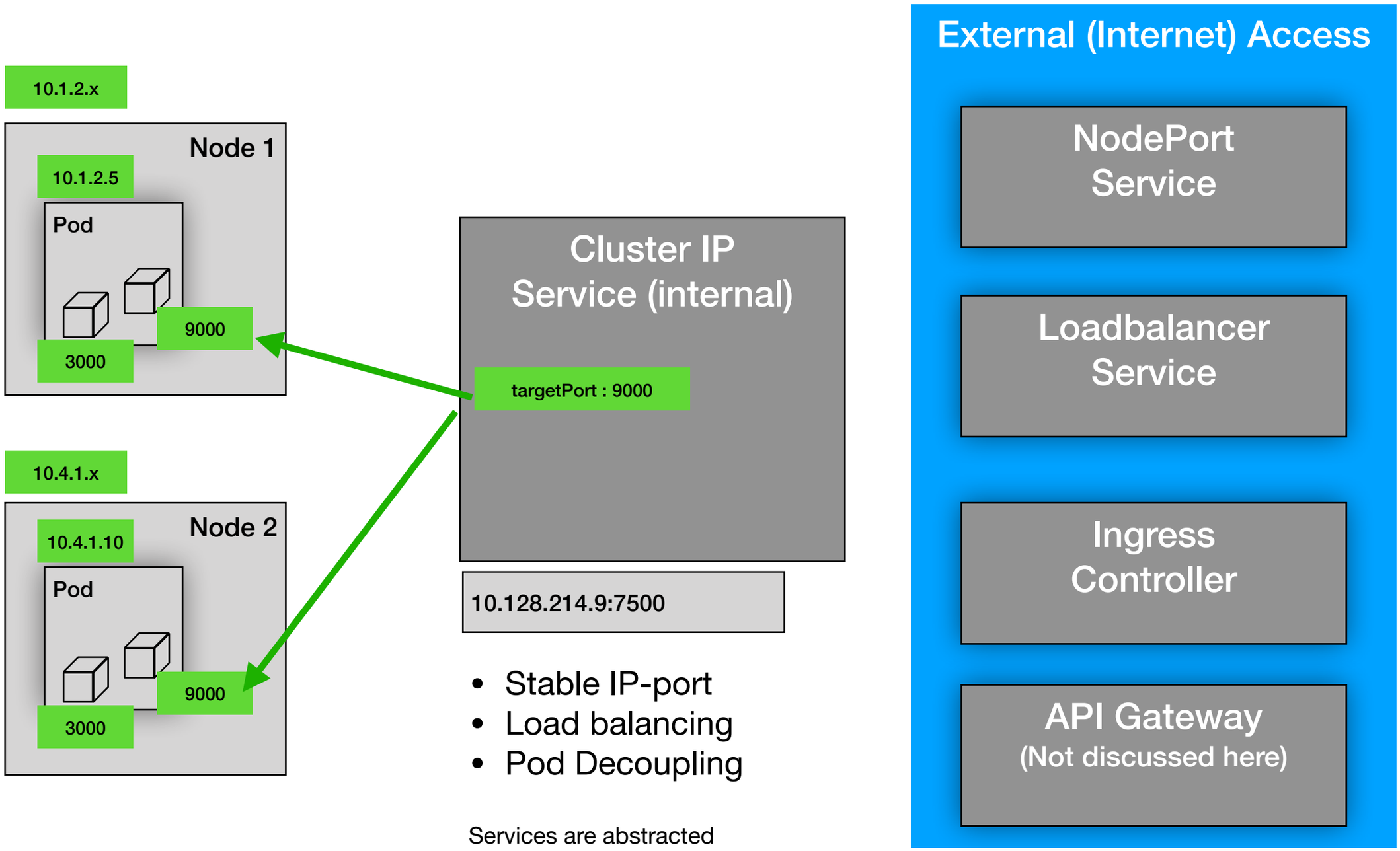
- https://kubernetes.io/docs/concepts/

TechWorld with Nana

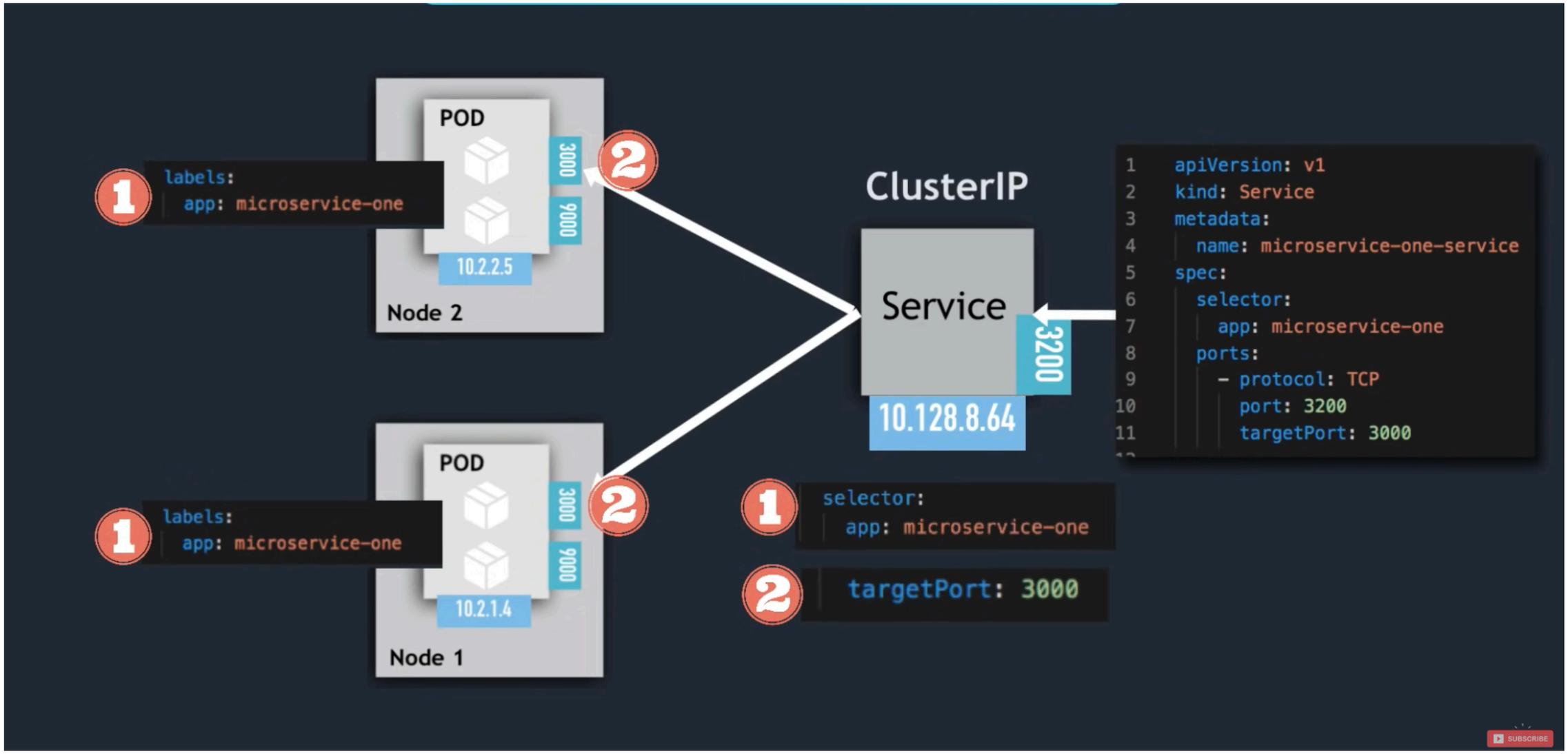
- https://www.youtube.com/watch?v=T4Z7visMM4E —> Services
- https://www.youtube.com/watch?v=80Ew_fsV4rM —> Ingress
- https://www.youtube.com/watch?v=pPQKAR1pA9U —> StatefulSet

Access to Pods via Services

- Modern distributed applications run in containerized environments where the number of instances of app tiers and their locations changes dynamically. Therefore internal IPs can change
- Kubernetes supports Services Discovery w/o the need of Application code to be aware of such location and intense number changes.



Service ClusterIP (internal)



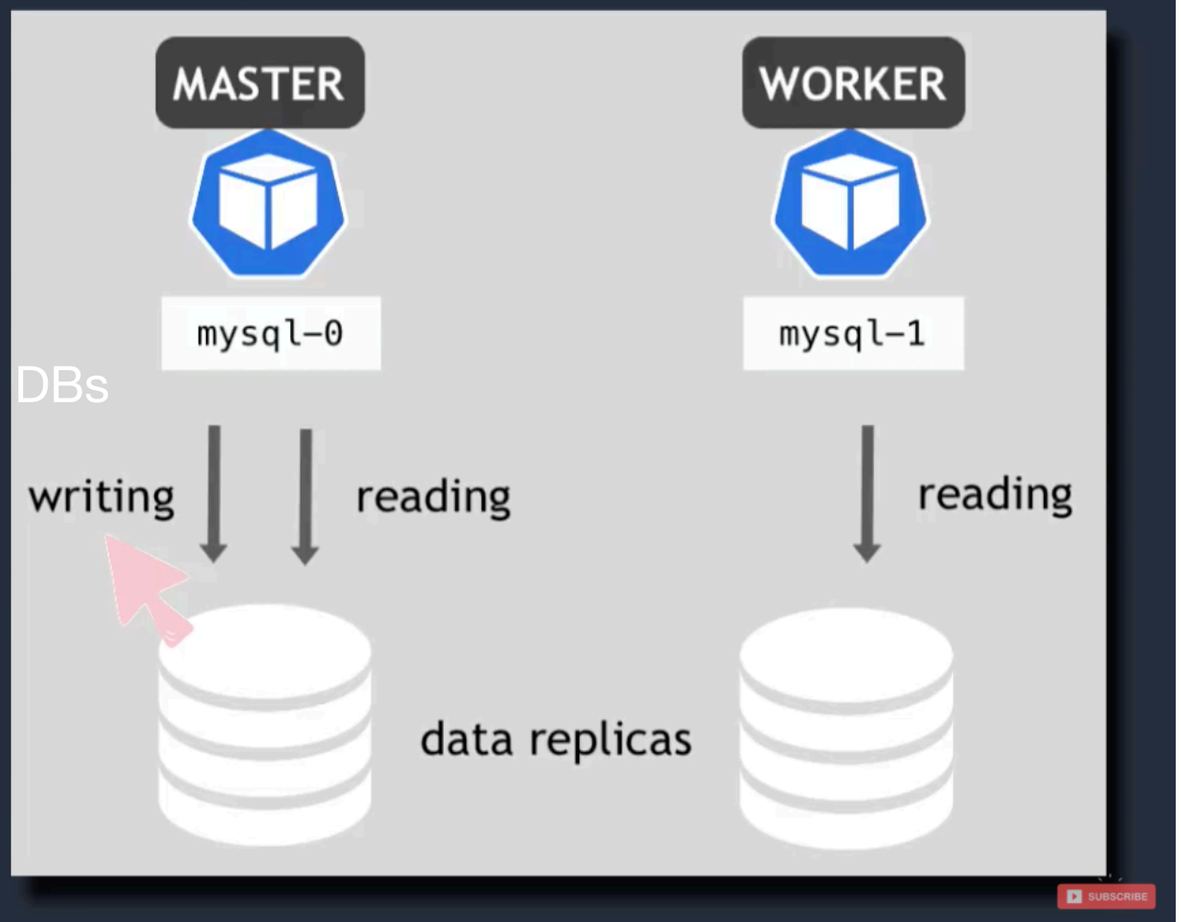
- As Service can be named (Meta.. Name) this name become the DNS entry to reach this ClusterIP Service
  - http://microservice-one-service:3200 —> 10.128.8.64:3200
  - The name of the service is also used as abstracted name in the programming (e.g. python)
- The Service load balance to all pods labeled app:microservice-one
  - LB to 10.2.2.5:3000 (targetPort) or 10.2.1.4:3000 (targetPort)
- Ref : https://www.youtube.com/watch?v=T4Z7visMM4E

Headless Services

- Client wants to communicate with **1 specific Pod directly**
- Pods want to talk **directly with specific Pod**
- So, **not randomly selected**
- **Use Case: Stateful applications**, like DBs

❌ Pod replicas are **not identical**

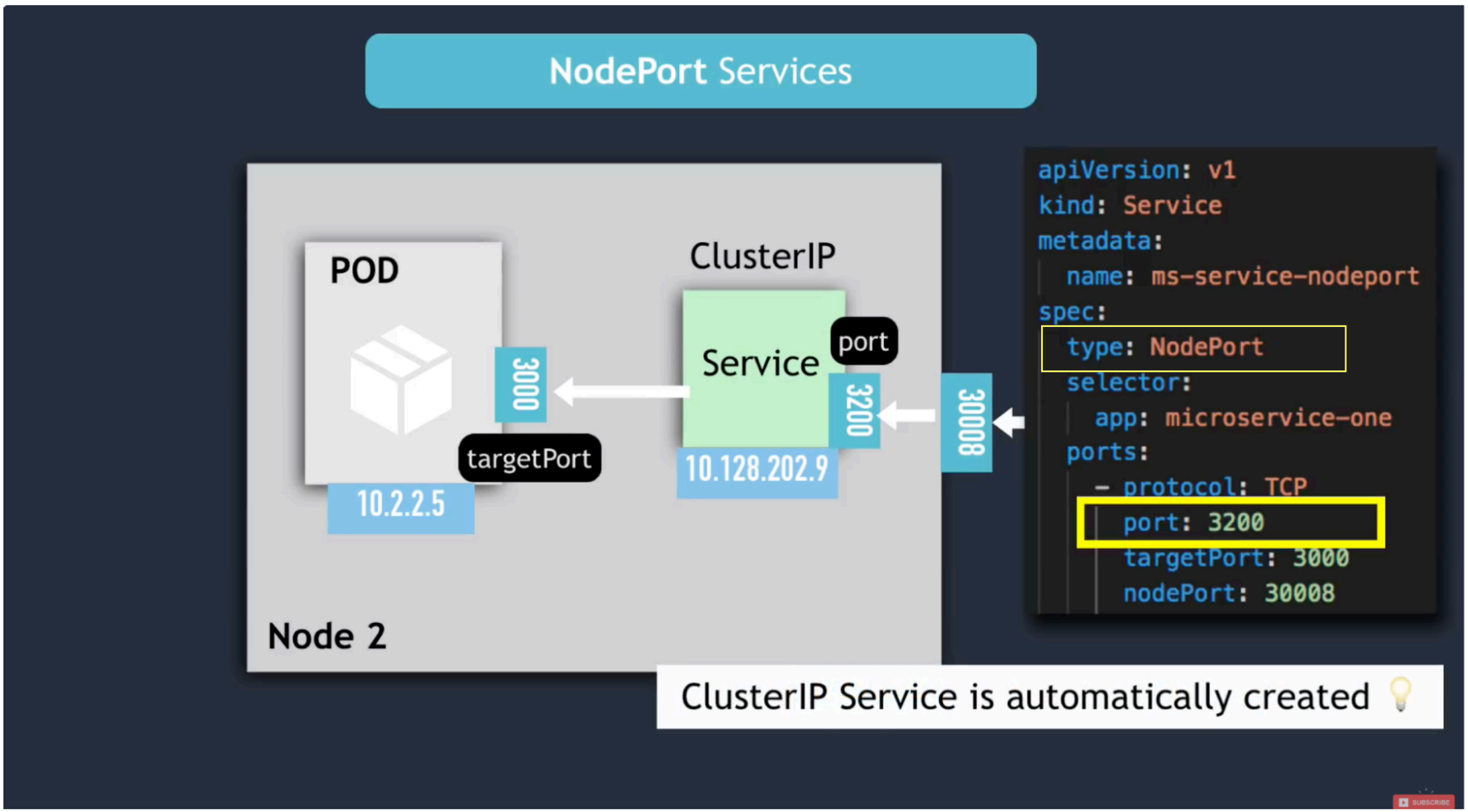
Only Master is allowed to write to DB



We have a replicated set of Pods, but need to communicate to a specific Pods (no random selection)

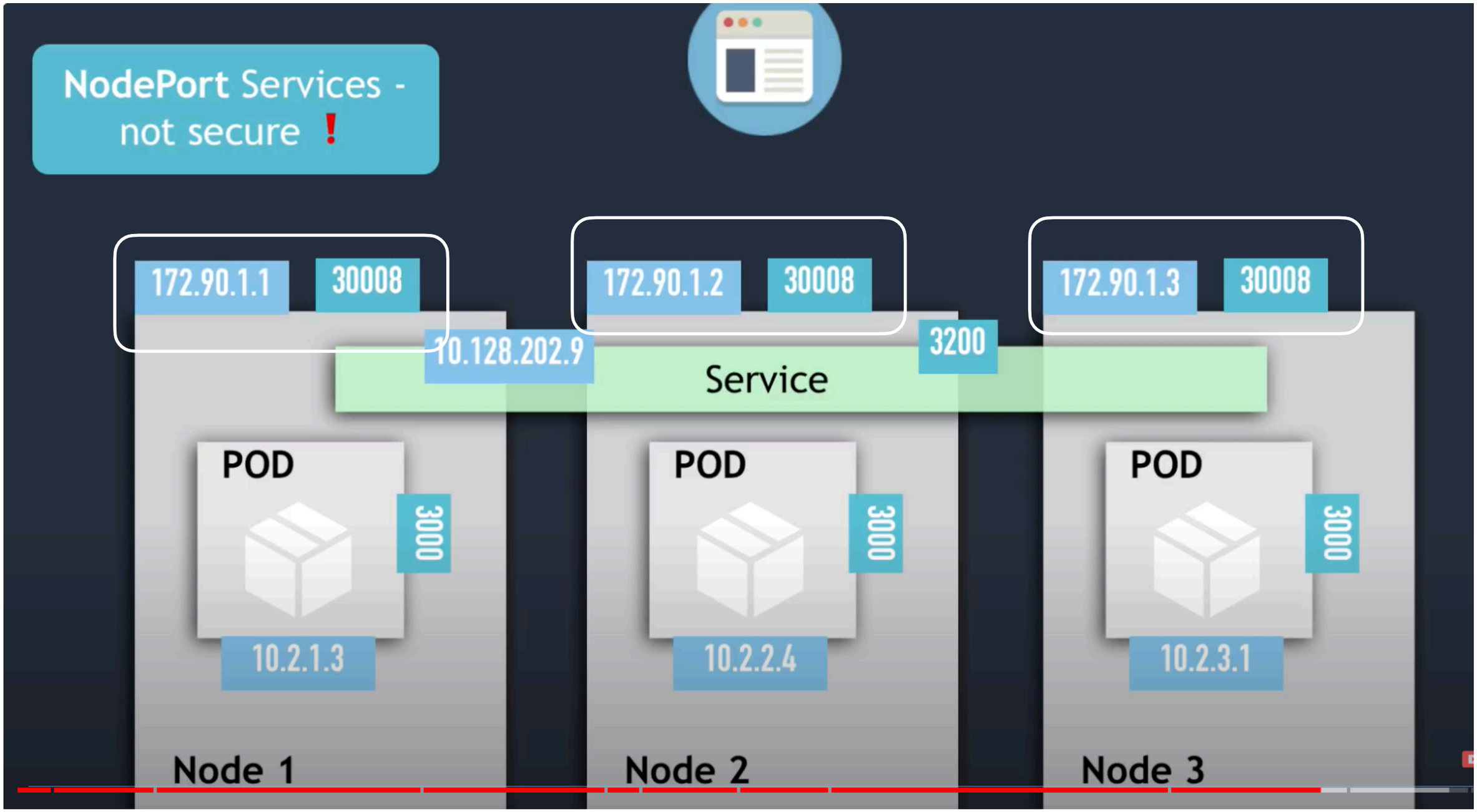
- Options are :
  - Programmed request to the API Server to get the IP:Port of a specific Pod (Disadvantage is the use of a specific K8s Interface)
  - DNS Lookup (plugin) via a headless service returns the IP:Port of the Pods

NodePort Service (External)



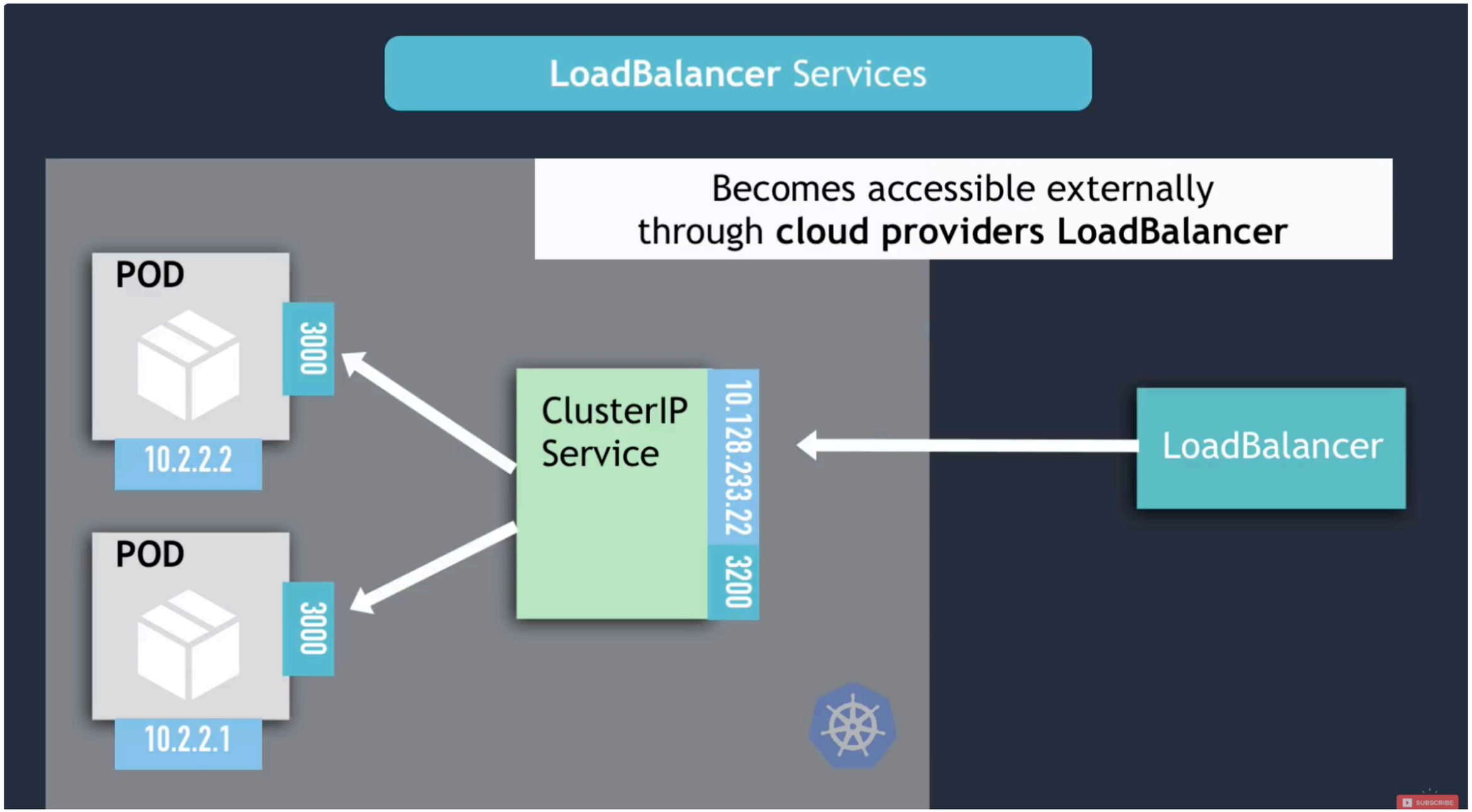
- In addition to the Cluster IP Service (which is generated automatically) each Node gets an additional Port in the range from 30000 to 32767 (or set by the configuration).
- With this port and the Node IP address the clusterIP Service can be reached.
- External Access is now possible via a external IP (if set by the cluster Administration) of that node and this node port.
- Ref : https://www.youtube.com/watch?v=T4Z7visMM4E

NodePort Service is not stable and in secure



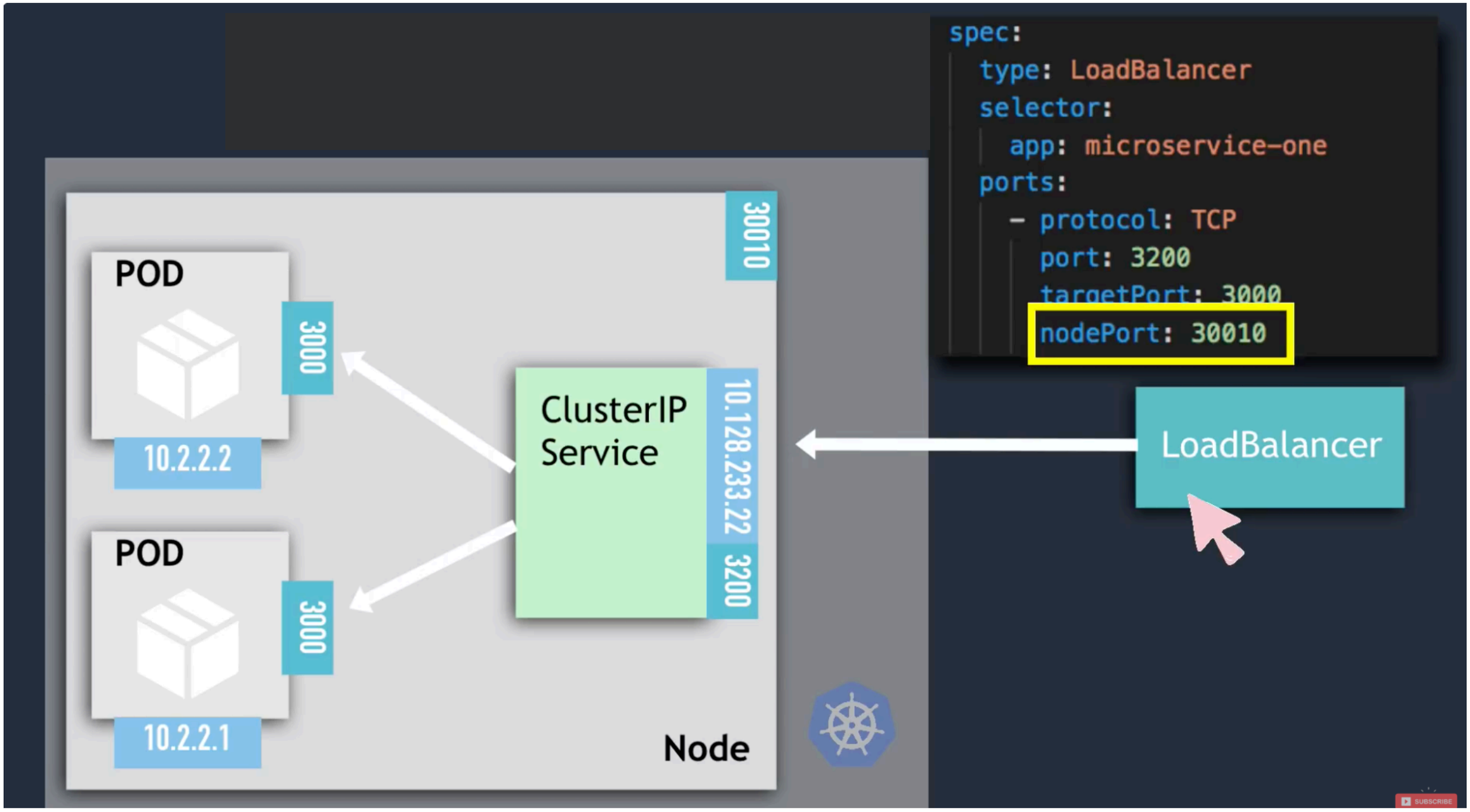
- Insecure access to worker Nodes as it requires an external IP and open Ports on each Node to even reach Pods on other Nodes. Furthermore a Node recycle will allocate a different external IP.
- Only used for first testing, not in production
- Ref : https://www.youtube.com/watch?v=T4Z7visMM4E

Loadbalancer Service



- Cloud Provider managed LoadBalancer supports the stable IP address.
- This Loadbalancer will be configured and integrated into Kubernetes via the Loadbalancer Service which is a extension of NodePort Service which again uses the Internal Service
- Ref : https://www.youtube.com/watch?v=T4Z7visMM4E

Load Balancer Service



- A Nodeport Service and a internal Service is generated out of the Specs
- nodePort and Node IP allows the access to the internal ClusterIP Service which then balance to the endpoint.
- The external Loadbalancer can balance between different nodes (eventually on different zones)
- External DNS Services can provide a Domain to the stable IP address (of the loadbalancer) mapping
- Ref : https://www.youtube.com/watch?v=T4Z7visMM4E

Ingress Resource = Enable ALB Routing to internal Services

- Capabilities varies very much on the cloud provider Implementation
- ALB Routing Rules finds the internal (ClusterIP) Services to access the Pods
- ALB Rules are defined in the Kubernetes Ingress Resource
- An Ingress Controller (one or more Pods)
  - Evaluates the ingress Rules
  - Provision/Configures the ALB (e.g. AWS ELB configured by the AWS LB Controller)
  - Can act as main external entry point for network traffic (e.g. minikube, with the K8s NGINX Controller)

Ingress and Internal Service configuration

In case of AWS it must be a Nodeport Service

Ingress:

```yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: myapp-ingress
spec:
  rules:
  - host: myapp.com
    http:
      paths:
      - backend:
          serviceName: myapp-internal-service
          servicePort: 8080
```

External Configured Host

any Path (in this case)

the service

Internal Service:

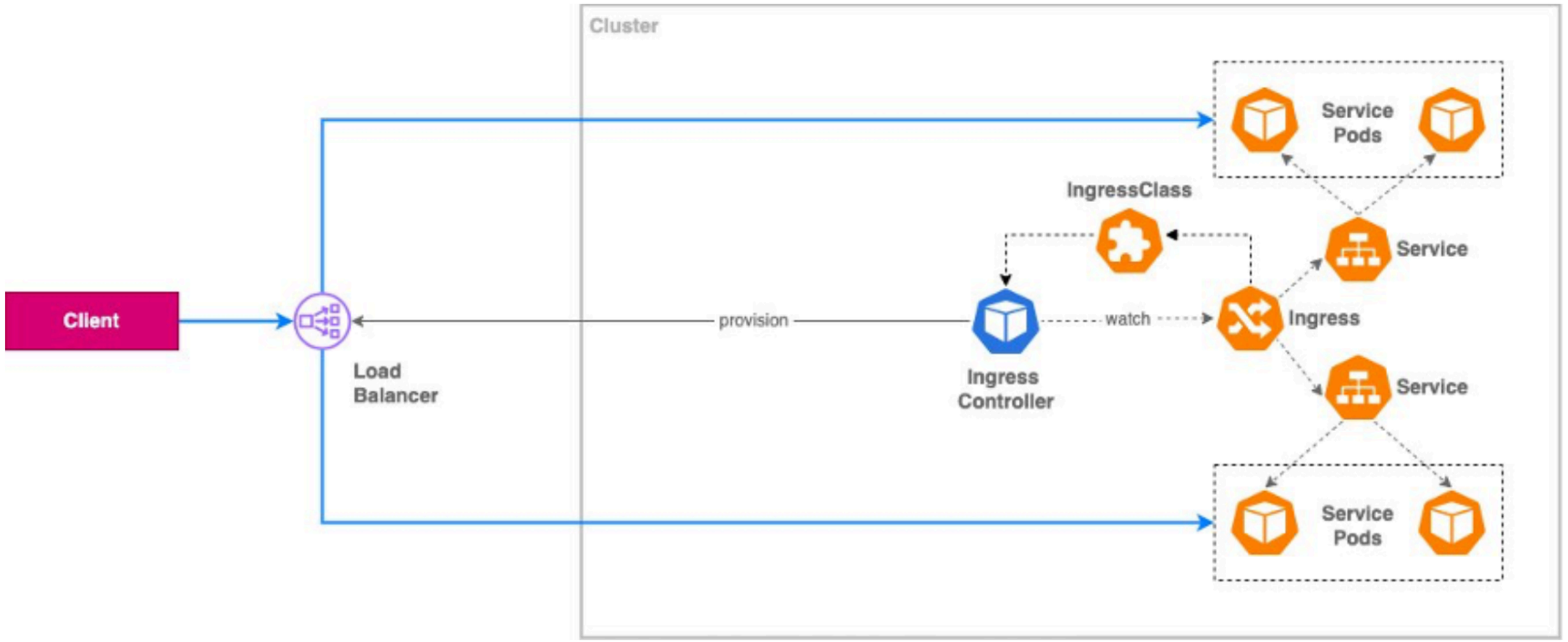
```yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-internal-service
spec:
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

the service

- https://www.youtube.com/watch?v=80Ew_fsV4rM

AWS Implementations

Case 1 : AWS Ingress ALB Controller

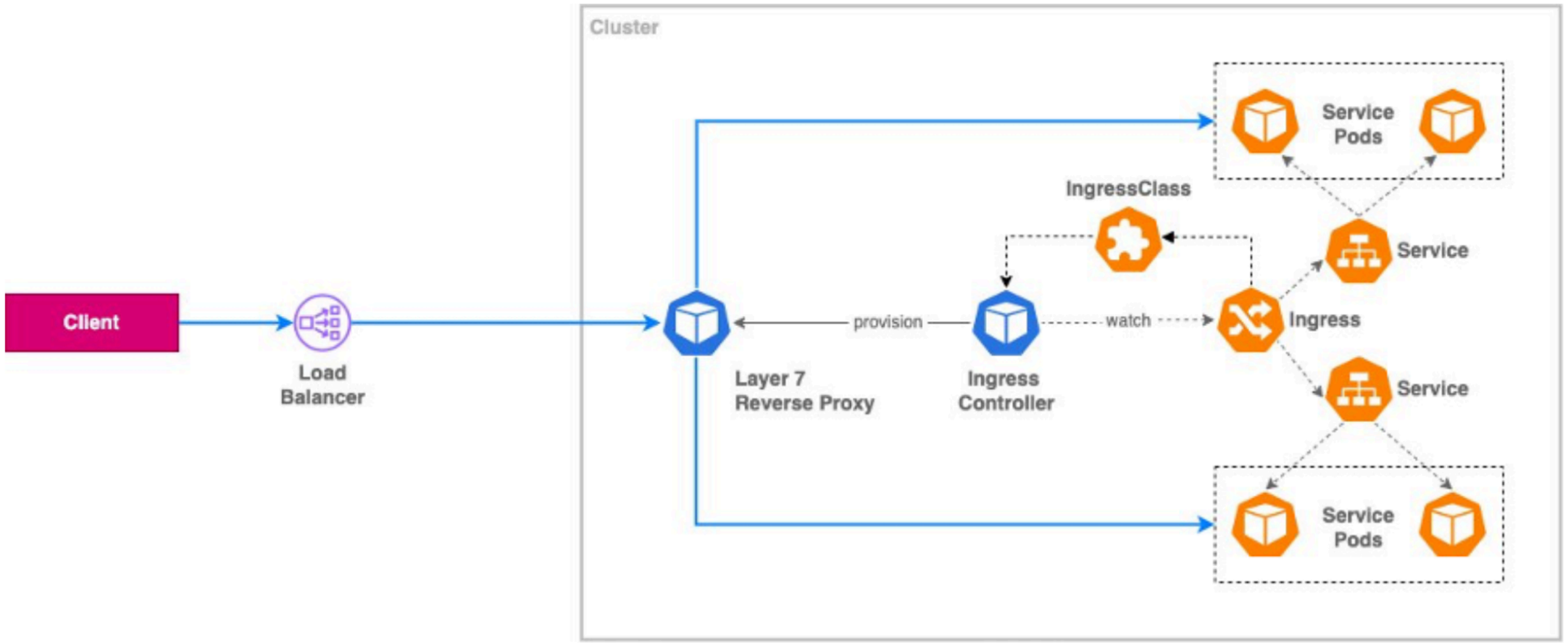


- AWS Ingress Controller provisions/configures the AWS ALB based on the Ingress rules and watches for rule changes
- The Services are actually NodePort Services, therefor the ALB can use the Node IP and Port to direct the incoming request the rules selected service (ClusterIP Service)

https://aws.amazon.com/de/blogs/containers/exposing-kubernetes-applications-part-1-service-and-ingress-resources/

AWS Implementations

Case 2 : Using an NGINX Controller implementations



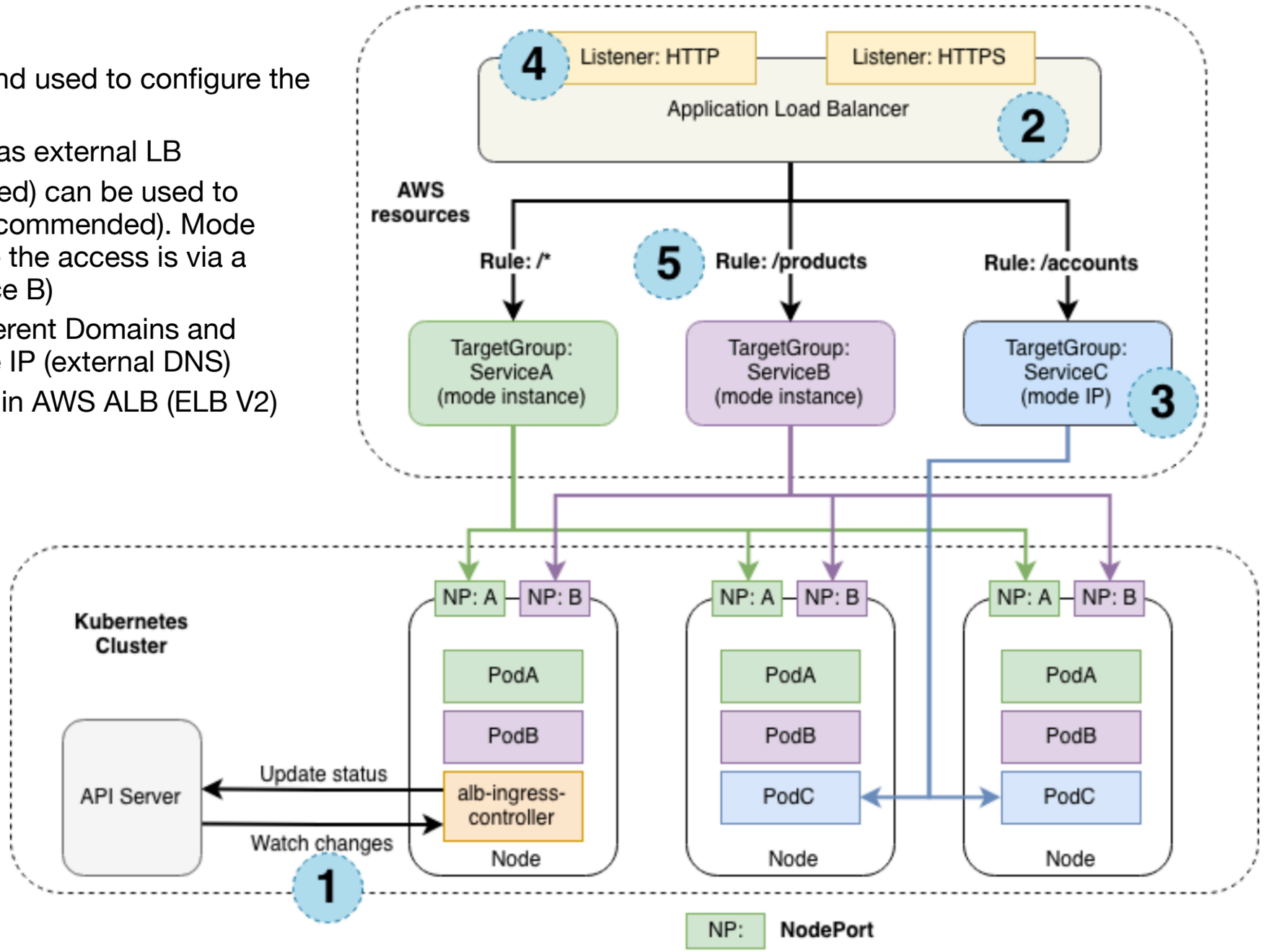
- The NGINX Implementation acts as reverse Proxy (the heart and soul of NGINX), The controller apply the NGINX ALB Rule based on the Ingress Resources.
- AWS Documentation prefers case 1

https://aws.amazon.com/de/blogs/containers/exposing-kubernetes-applications-part-1-service-and-ingress-resources/

AWS Implementations

Case 1 : AWS ALB Ingress Controller in more detail

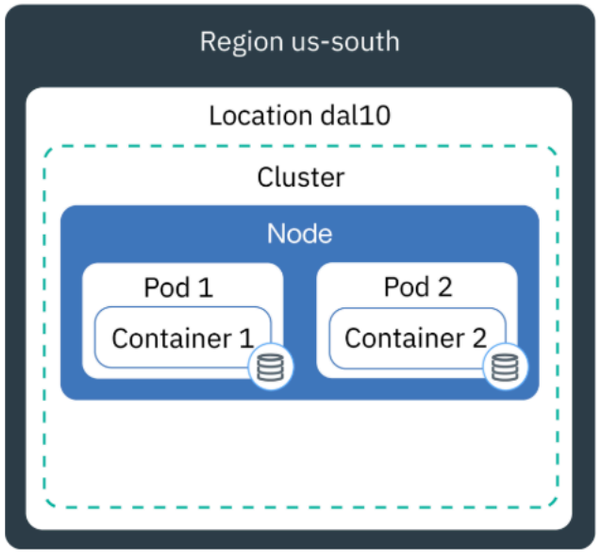
1. Rules Changes are monitored and used to configure the ALB (ELB V2)
2. The AWS ALB (ELB V2) is used as external LB
3. A headless Service (not discussed) can be used to directly access the Pods (not recommended). Mode Instance is the default and there the access is via a Node Port (Service A and Service B)
4. HTTP and HTTPS Listener. (Different Domains and Subdomains can have the same IP (external DNS)
5. The Ingress Rules implemented in AWS ALB (ELB V2)



https://kubernetes-sigs.github.io/aws-load-balancer-controller/v2.4/how-it-works/

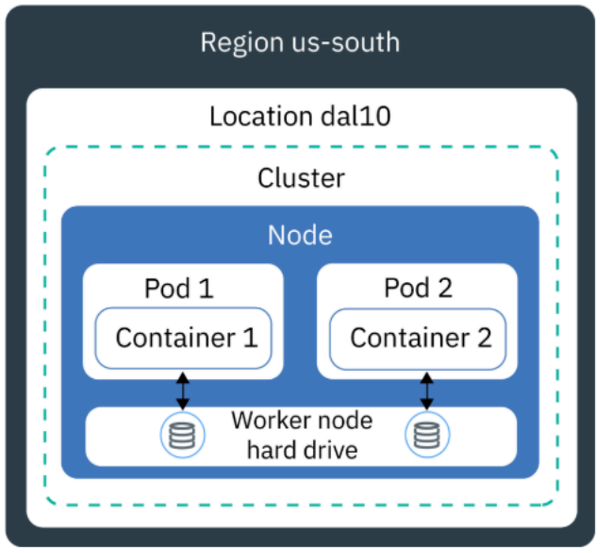
Storage in the K8s Cloud

1. Inside the container



- Inside the Image of the container (as discussed in our docker sections)
  - Local FS
  - Disappears when the container or worker node goes away

2. On the worker node

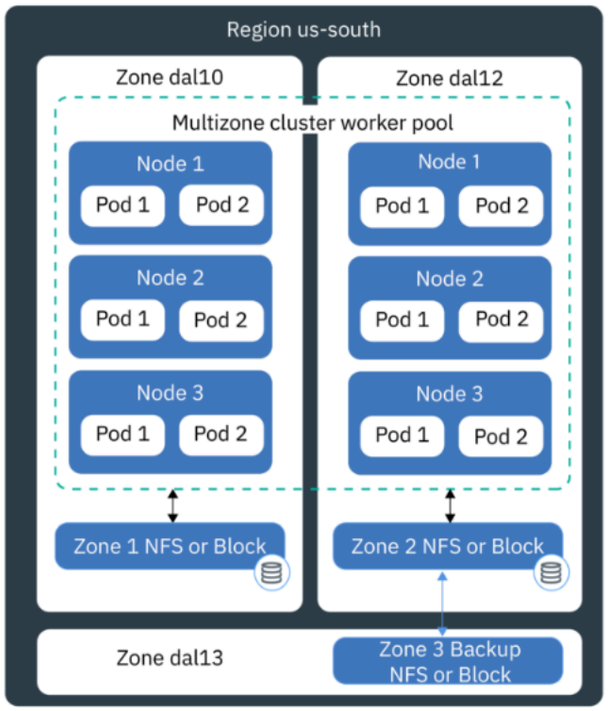


- As part of the worker node, comparable the docker mount-bind, but more specify volume types
- K8s allows access to such storage as mount-bind
- disappears when the worker nodes goes away (survives a container crash and restart)

https://kubernetes.io/docs/concepts/storage/

Storage in the K8s Cloud

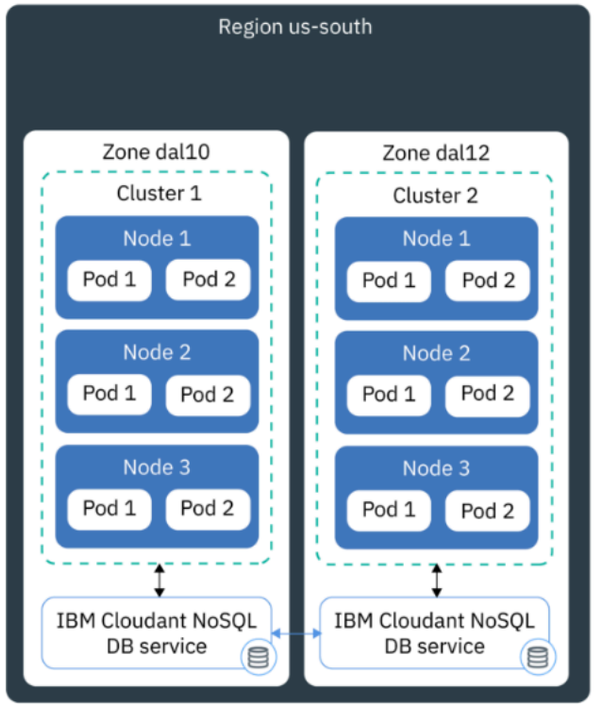
1. NFS file storage or block storage



- External to Workers
  - Cloud specific Volumes (e.g. EBS, AzureDisk, cephfs, emptydir, iscsi, ConfigMap, NFS and many more)

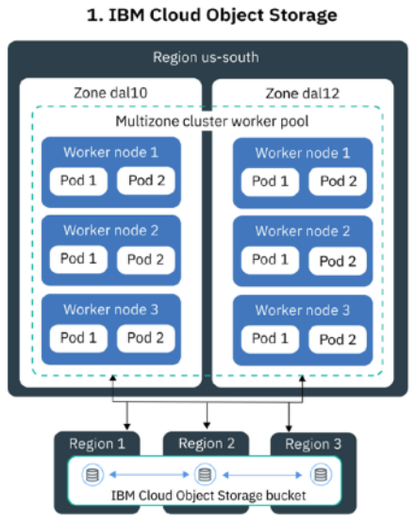
  - External e.g. NFS (e.g. IBM Cloud implementation)
  - Services with different QoS (IOPS, Size, Backup)
  - Data is encrypted per default
  - NFS can be shared among pods/nodes within a zone
  - No sharing across zones and regions (IBM Implementation)
  - Survive an outage of the pods and workers

2. Cloud database service



- Database as a Service. The best place to solve all the cross zone and region sharing and replication challenges. (This is why we have Databases)

1. IBM Cloud Object Storage

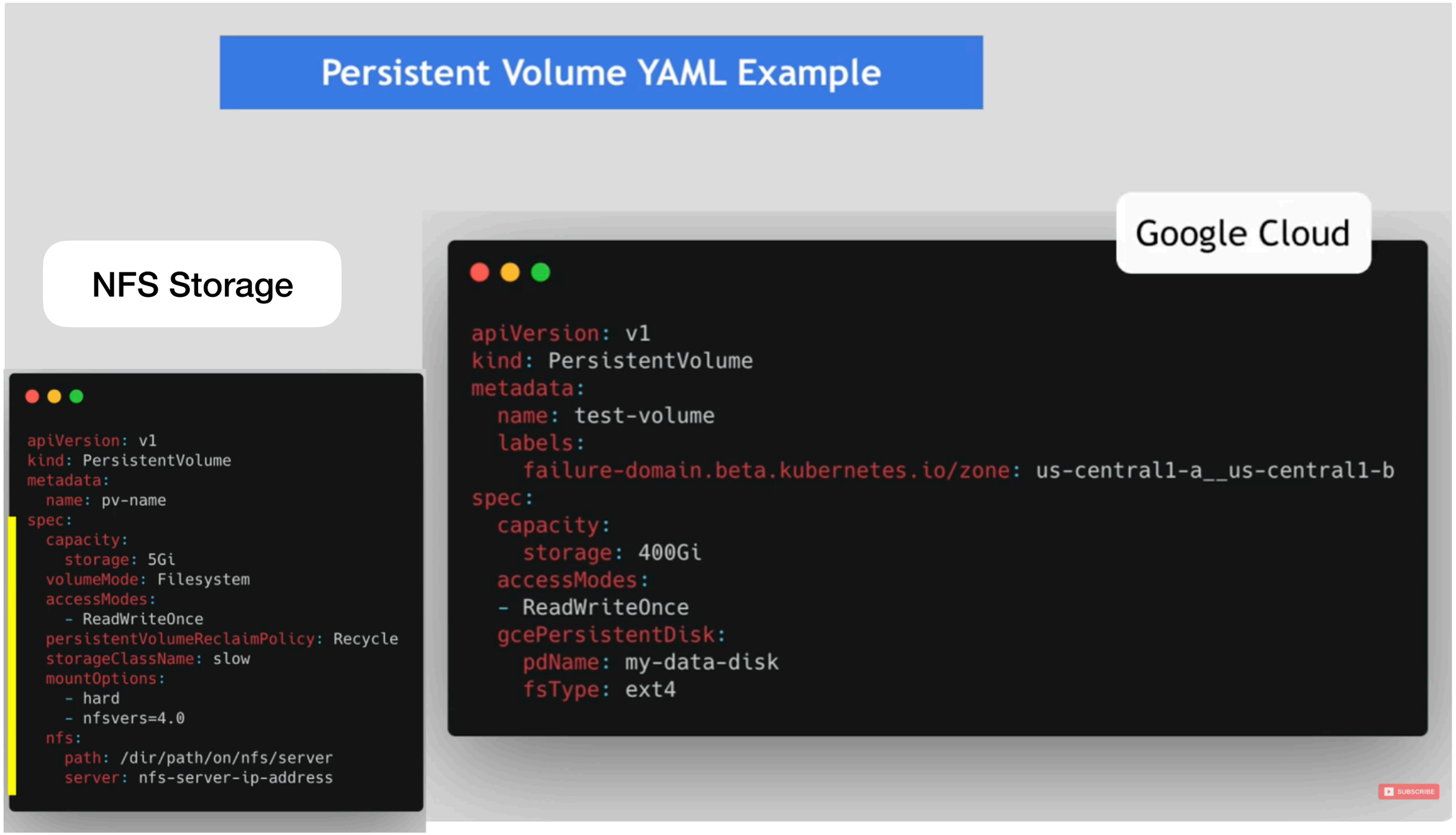


- Object Stores have the capability to store data into 3 different places (e.g. Regions) and can always recover when 2 places are still OK (outage of one place)

Storage in K8s

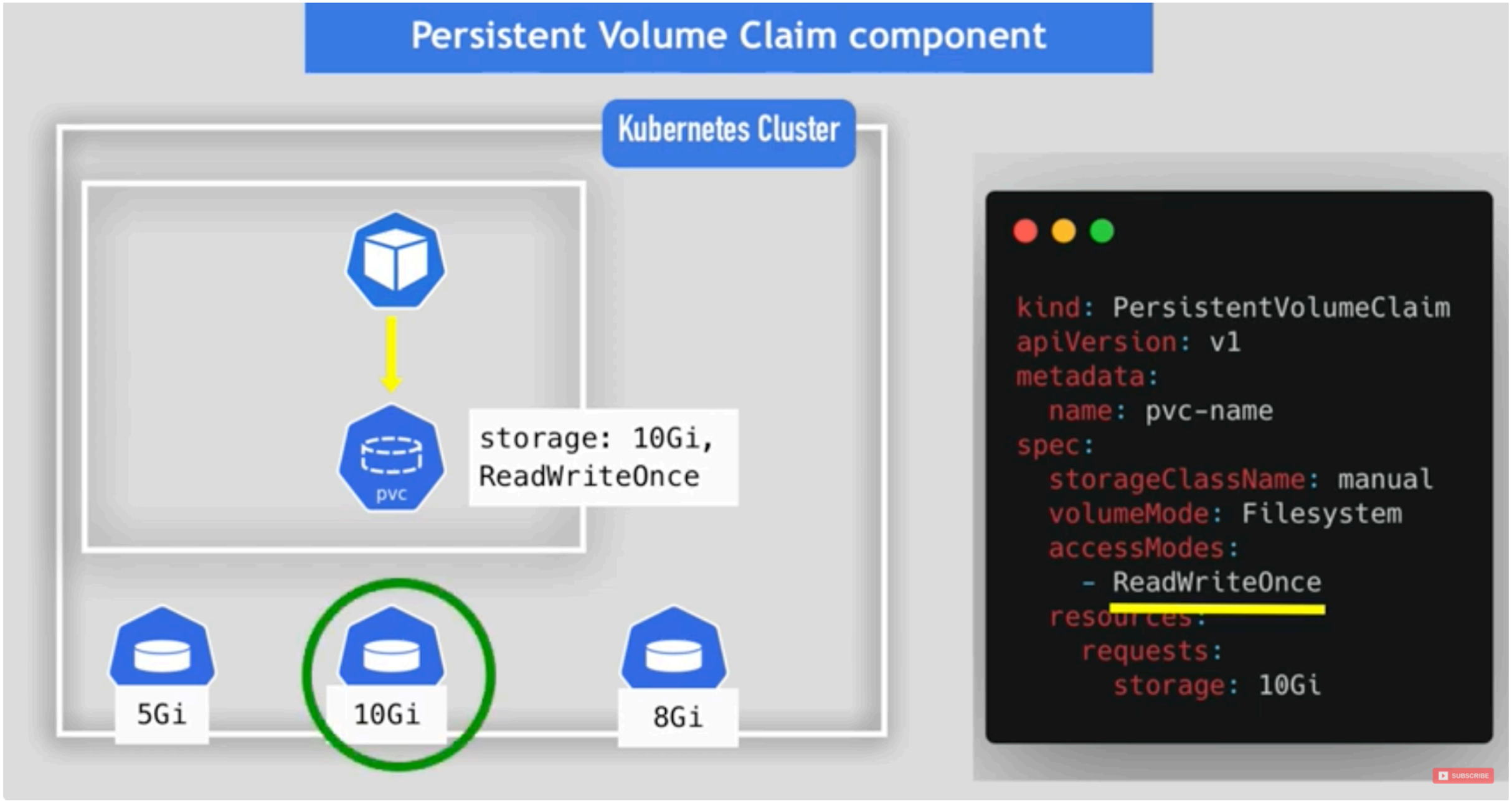
Persistent Volume	Persistent Volume Claim	Storage Class
		
Allocated Volumes Done by Storage and Cluster Admin	Interface for Pods Used by the App Developer (part of the Pod Specs)	Collections of Storage descriptions including an interface to a storage provisioner Storage provisioned when claimed, Managed by the Storage Cluster Admin

Persistent Volume in K8s



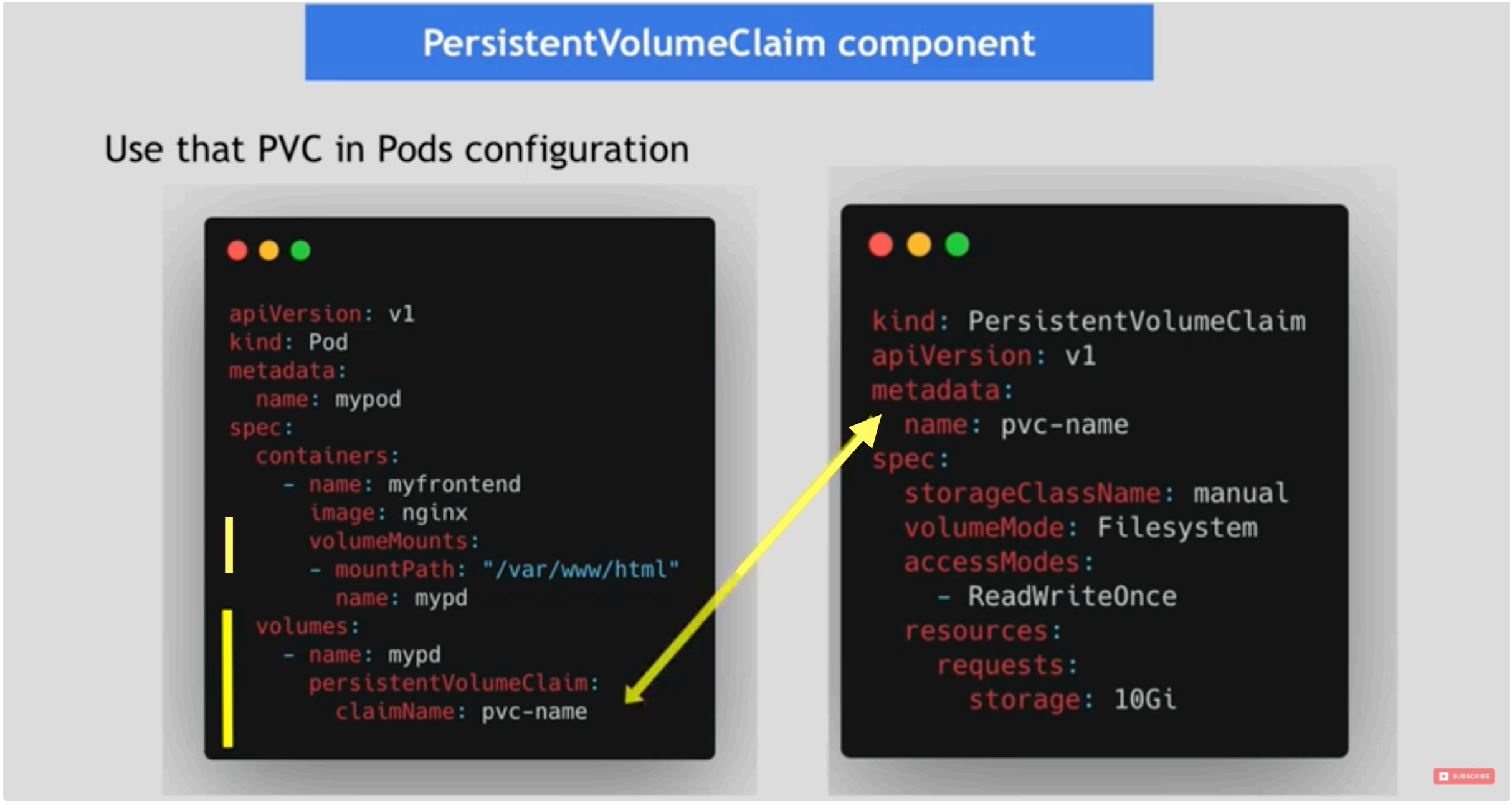
- The persistent Volume Spec describes the detailed (physical) properties of an existing (pre-allocated) volume. (e.g. Google Cloud, NFS Storage, the Spec s look very different)
- The mentioning of a storageClassName give more option, however it is still pre-allocated)
- https://www.youtube.com/watch?v=0swOh5C3OVM

Persistent Volume Claim in K8s



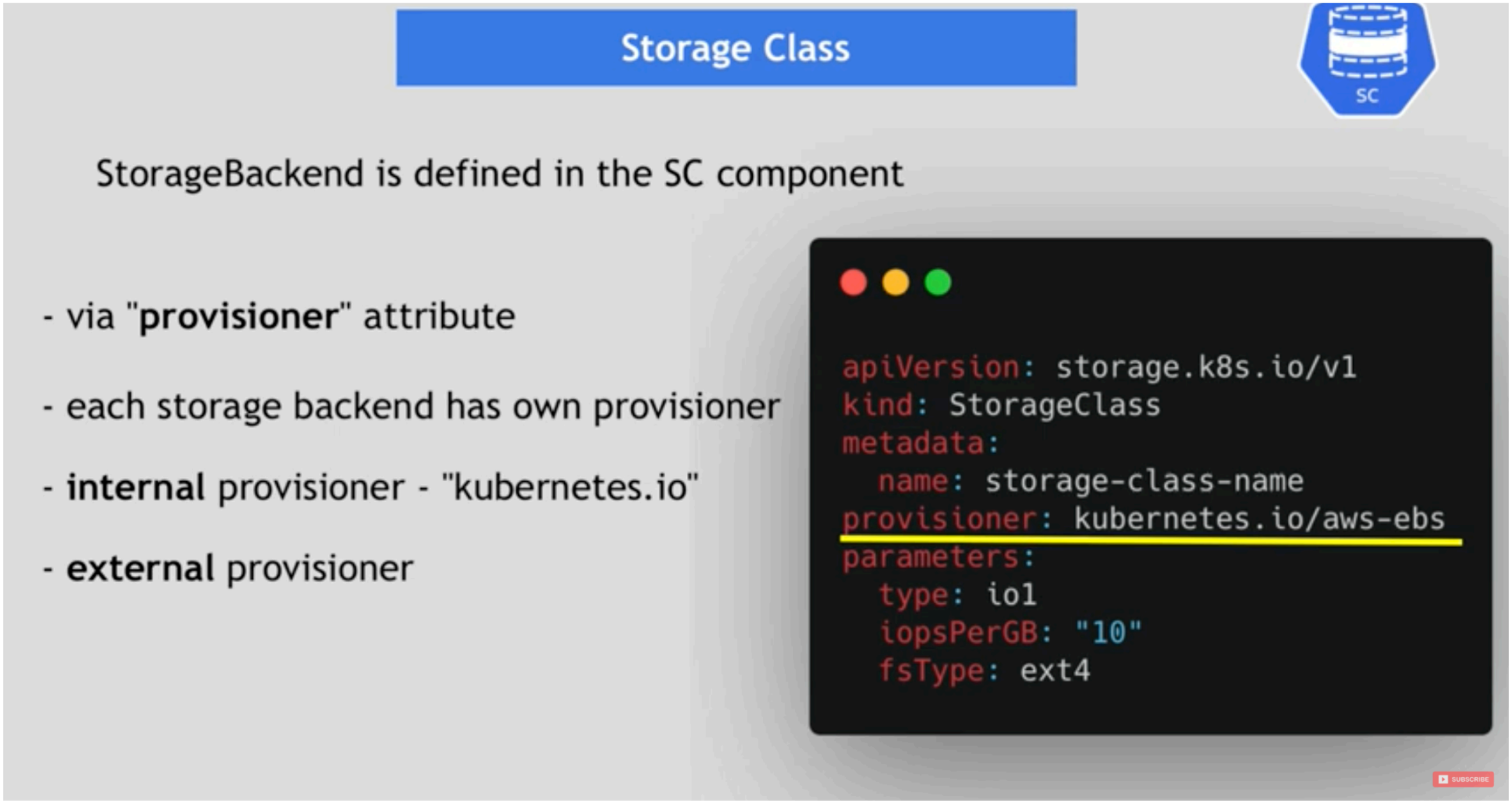
- The persistent Volume claim Spec is a smaller (Dev oriented) set of Properties needed to locate or provision the right volume
- StorageClassName : manual means the the volume ist pre-allocated, other Names will trigger a dynamic provisioning)
- https://www.youtube.com/watch?v=0swOh5C3OVM

Claims as part of the Pod



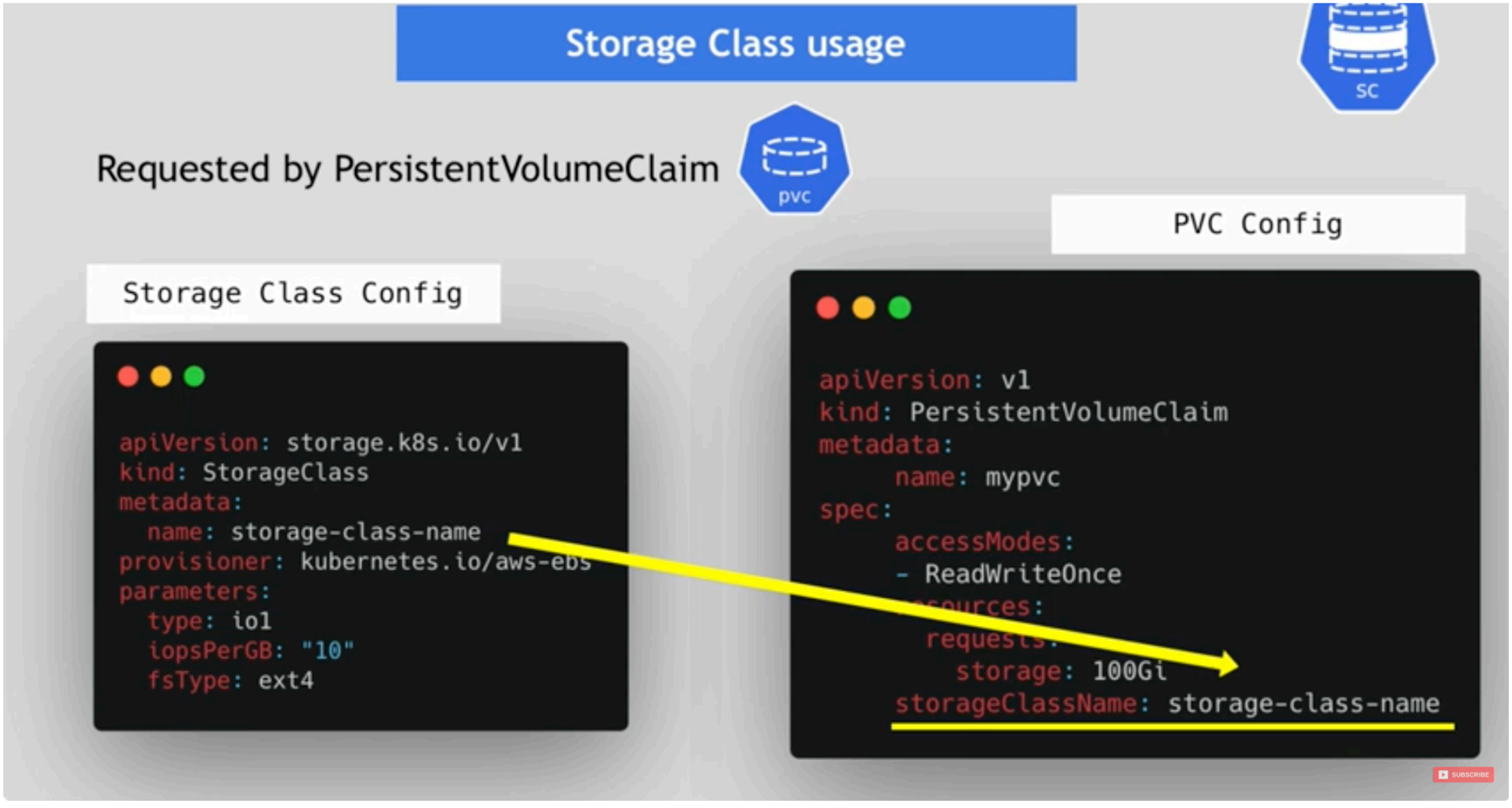
- The Pod Configuration Spec mentions the volumes pointing the the claims
- The mount Points is part of the container properties
- https://www.youtube.com/watch?v=0swOh5C3OVM
-

Storage Class in K8s



- The Storage Class defines the Storage Provisioner properties (in this case aws-elastic block storage) https://www.youtube.com/watch?v=0swOh5C3OVM

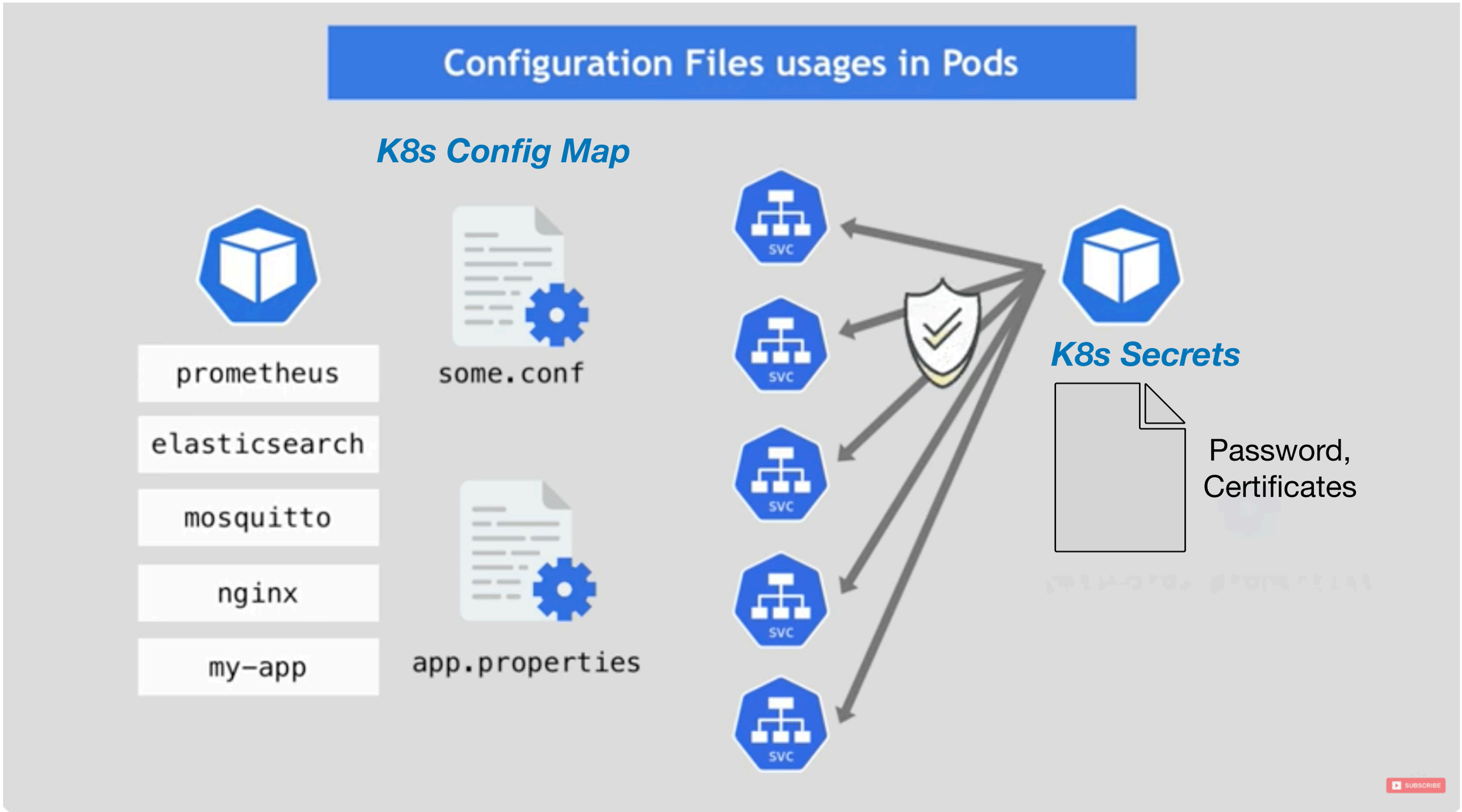
Volumes Claims using Storage Class in K8s



- The Persistent Volume Claim refers to the storage class. In this case the provisioning of storage happens when the Pod claims the volume using this PVC

- https://www.youtube.com/watch?v=0swOh5C3OVM

Special K8s local Files



- Configurations and Secrets can be mapped as
  - environment data -> Key:Value Pairs taken from that Object and added in :env Property in the Pod Specs
  - can be allocated as local file (Very similar to local file volume and mount in the Pod Specs)
- https://www.youtube.com/watch?v=FAnQTgr04mU

Minikube a fast way to play with Kubernetes

How to install :

- https://minikube.sigs.k8s.io/docs/start/
- .. are basically two commands, however it requires a active running local docker
- Example on an Mac:
- Dokument in Lösungen Folder

If the Homebrew Package Manager is installed:

```
brew install minikube
```

If `which minikube` fails after installation via brew, you may have to remove the old minikube links and link the newly installed binary:

```
brew unlink minikube
brew link minikube
```

2 Start your cluster

From a terminal with administrator access (but not logged in as root), run:

```
minikube start
```

If minikube fails to start, see the drivers page for help setting up a compatible container or virtual-machine manager.

3 Interact with your cluster

If you already have kubectl installed, you can now use it to access your shiny new cluster:

```
kubectl get po -A
```

Alternatively, minikube can download the appropriate version of kubectl and you should be able to use it like this:

```
minikube kubectl -- get po -A
```

You can also make your life easier by adding the following to your shell config:

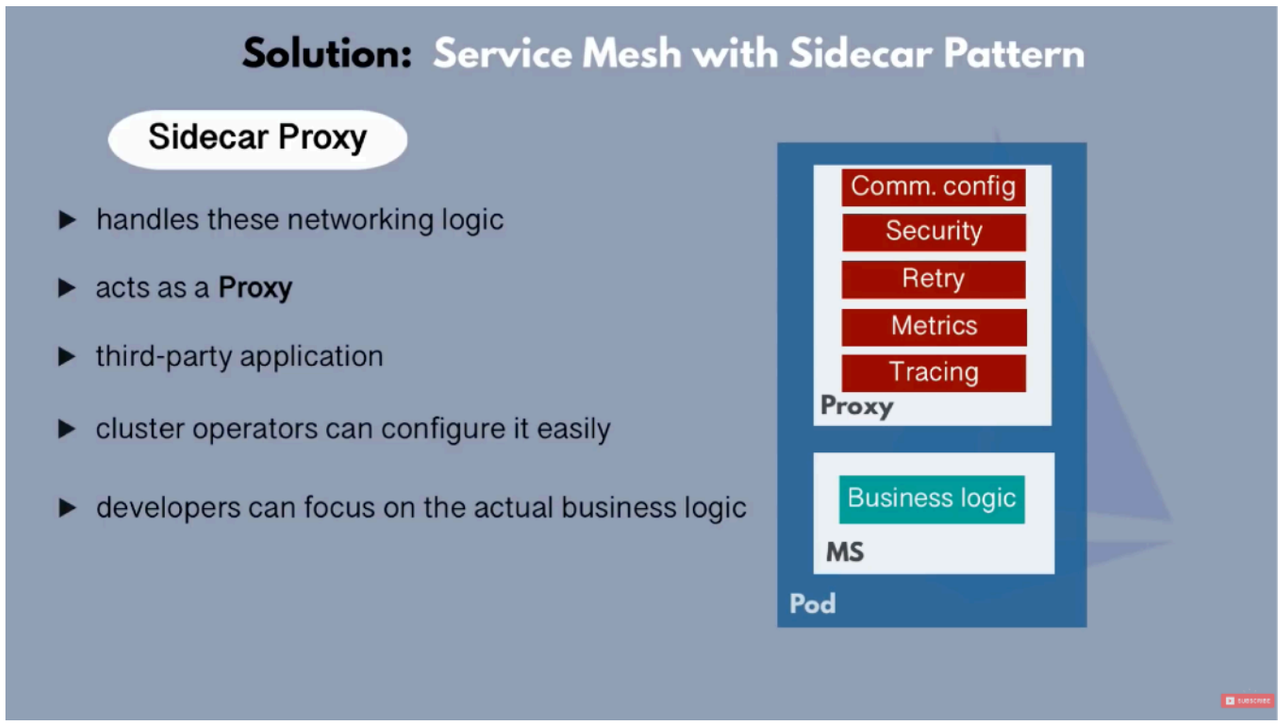
```
alias kubectl="minikube kubectl --"
```

Initially, some services such as the storage-provisioner, may not yet be in a Running state. This is a normal condition during cluster bring-up, and will resolve itself momentarily. For additional insight into your cluster state, minikube bundles the Kubernetes Dashboard, allowing you to get easily acclimated to your new environment:

```
minikube dashboard
```

A bit more on Networking

- Kubernetes main goal is to host so called microservices
  - A big monolithic App is broken down into many smaller independent parts (= Microservices)
  - Microservices are using a Internet protocol to interact (data driven)
  - Microservices can be composed into different Applications
  - Microservices do not share data with other Microservices
  - That results in
    - **Many** Pods hosting business logic for different services (stateful and stateless)
    - We have a Mesh of Services (Maschendrahtzaun)



A Service Mesh Solution separated the many infrastructure parts from the app business logic. (Dev/Ops)

The developer should care about the business logic

The operator setup the infrastructure

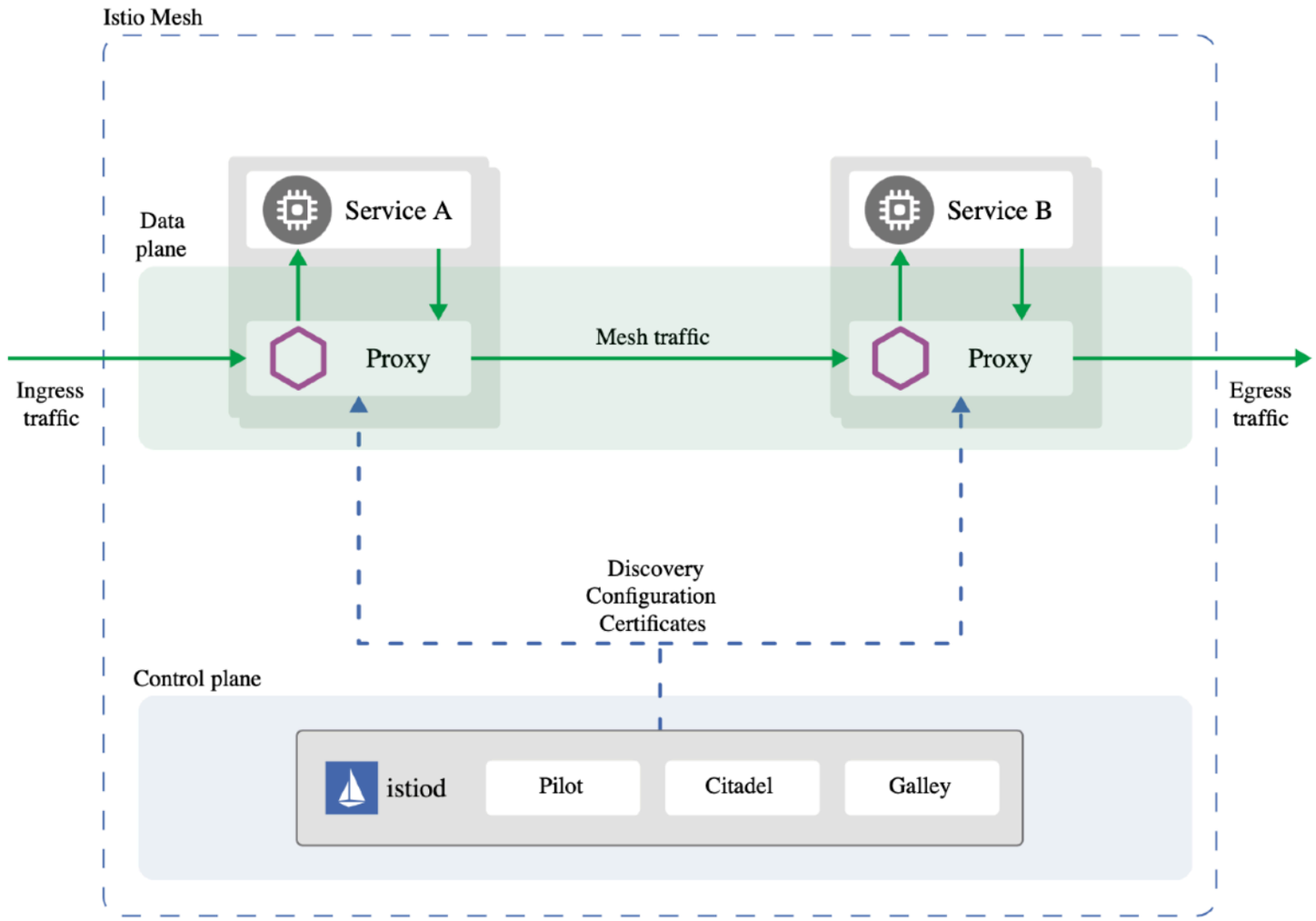
- Which MS can talk to which other MS
- Helps with network latency troubleshooting
- Securely connects the MS
- Observability for tracing and alerting

https://www.youtube.com/watch?v=16fgzklcF7Y

Service Mesh Implementation ISTIO

- Istio is an open platform (IBM, Google, Lyft) to connect, manage, and secure microservices.
- Istio provides an easy way to create a network of deployed services with load balancing, service-to-service authentication, monitoring, and more, without requiring any changes in the business logic code.
  - **Traffic Management**. Control the flow of traffic and API calls between Application Pods
  - **Service Identity and Security**. Provide services in the mesh with a verifiable identity and provide the ability to protect service traffic as it flows over networks of varying degrees of trustability.
  - **Policy Enforcement**. Apply organizational policy to the interaction between services, ensure access policies are enforced and resources are fairly distributed among consumers.
    - e.g. **Canary Deployment** (Traffic split between versions of a service)
- **Telemetry**. Gain understanding of the dependencies between services and the nature and flow of traffic between them, providing the ability to quickly identify issues.

https://istio.io/docs/concepts/what-is-istio/overview/



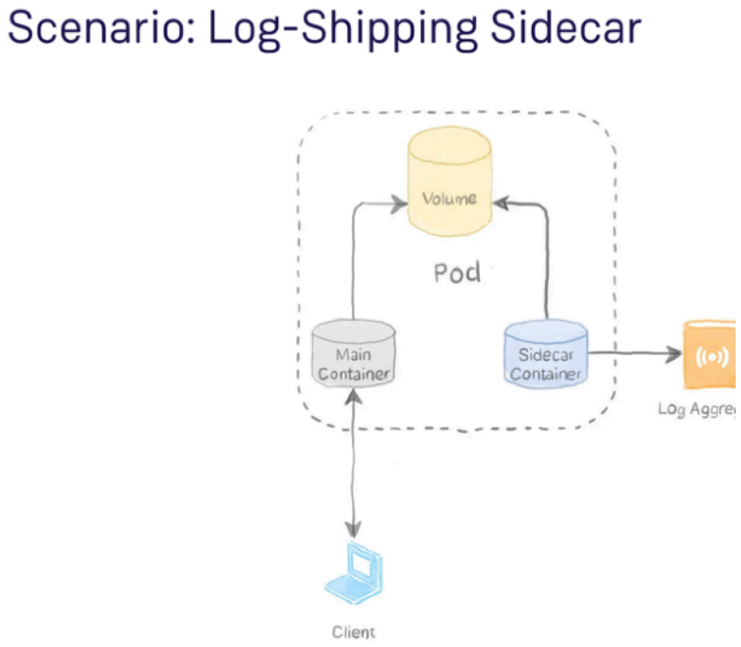
Istio Architecture

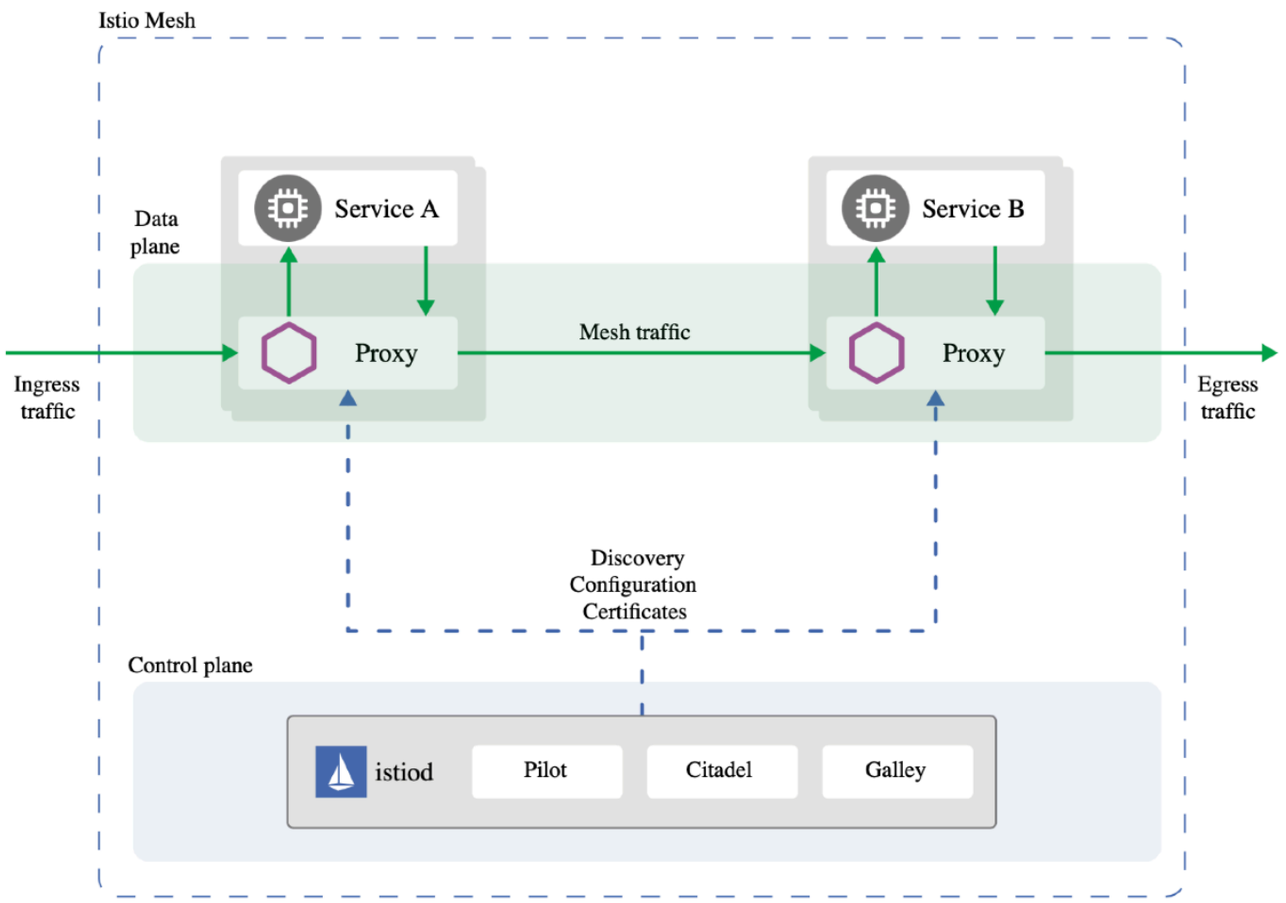
- Side Car Pattern : each Pods gets an Network Proxy (Data Plane)
- ISTIO Control Plane is a set of Pods managing
  - Discovery, Configurations including Monitoring and mTLS certificates

Note: A Service in ISTIO terms is a MicroService (MS)

Example of the Side-Car Pattern

A Pod is the basic atomic unit of deployment in Kubernetes. Typically, a Pod contains a single container. However, multiple containers can be placed in the same Pod. All containers running on the same Pod often share the same volume and network interface of the Pod.





Istio Architecture

References

- https://www.magalix.com/blog/the-sidecar-pattern
- https://istio.io/latest/docs/concepts/what-is-istio/

Kubernetes Recap

- Kubernetes ist im Grunde der de facto Standard (der große Bruder oder die große Schwester von Docker Swarm) für das Verwalten von containerized Apps über mehrere Container Hosts (e.g. Docker Hosts) = Worker Nodes . Diese Ansammlung von Hosts nennt man auch Cluster.
- Der Nutzen dieser Cluster ist die Verteilung solcher Hosts auf mehrere physikalischen Server in unterschiedlichen Availability Zonen (DC). Damit erhöht sich die Ausfallsicherheit und es kann auch zu einer besseren Performance führen
- Unterschiedliche physikalische Qualitäten solcher worker nodes können in worker pools oder auch durch taggen von Labels (key-values) beschrieben werden. Dafür gibt es dann Regeln um diese Container auf die unterschiedlichen nodes zu verteilen.
- Was ist die Rolle eines Master Plane, und welche Management Fähigkeiten werden dort (grobe Übersicht) und auf jedem Worker Node bereitgestellt (große Übersicht).
- Was ist der Unterschied von einem Pod und einem Container.
- Alle K8s Objekte stellen im Grunde den gewünschten Zustand dieser dahinter liegenden Resource dar. Das Kubernetes System versucht nun wenn diese Objekte appliziert werden (kubectl apply -f abc.yml) den dort beschriebenen Wunsch zu konfigurieren.
- Diese Objekte werden entweder in JSON oder YML Format zur Verfügung gestellt.
- Was kann man durch die Nutzung einer Namespace Isolierung in einem Kubernetes Cluster erreichen.
- Was kann ich durch labels oder annotations erreichen.
- Anwendungen die in Container laufen, haben unterschiedliche Laufzeitverhalten. Welche drei unterschiedlichen Typen gibt es dazu.
- Das Service Objekt beschreibt den IP basierten Zugriff auf ein oder mehrere Pods eines Anwendungstiers. Es gibt den Zugriff auf die Pods obwohl sich sich der Ort und die Anzahl der Pods verändern kann
- Was ist eine Service Discovery und wie funktioniert das im Groben in Kube ? Was ist eine Service Registry ?
- Was ist der ClusterIP Service ?
- Was ist ein Headless Service und wann brauche ich sowas
- Welche Möglichkeiten gibt es solche Services auch extern zu erreichen
    - Der Nodeport Service gibt in einer Testumgebung die Möglichkeit diesen Service über eine vom System gesetzte Port Nummer und der möglichen externen IP Adresse zu erreichen. Wie ist grob der Ablauf ? Warum git das als insecure und ineffizient ??
    - Der Loadbalancer Service setzt im Grunde einen vom Cloud Provider gestellten NLB voran. Dieser Loadbalancer findet nun den dahinter gesetzten Service (Nodeport Service). Da ich nun die externe IP stabil habe (die IP des Loadbalancer) kann man auch eine eigene DNS oder eine vom Cloud Provider zur Verfügung gestellten DNS Service nutzen.
    - Der Ingress Controller (eigener K8s Object) erweitert die Nutzung durch die Einführung eines ALBs, den man typischerweise zwischen dem von einem Cloud Provider gestellten LB der zwischen Zonen balanced und einem Nodeport Service setzt. Damit verbunden sind oft integrierte DNS und TLS (Zertifikate) Service.
    - Unterschied zwischen ALB und NLB.
    - Multi Zonen Support muss man über einen externen (ausserhalb der Zonen) Balancer (e.g Cloudflare) ermöglicht werden
- Welche drei Objekte gibt es um die Anbindung von Storage zu deklarieren. Was tun (sehr grob) die Implementierungen diese Objekte.
- Was ist der Sinn und Zweck von sogenannten Config Maps oder Secret Stores.
- Was versteht man unter einem Sidecar Pattern

Referenzen

- Docker
  - Steffen Uhlig, IBM. https://ws.uhlig.it
  - Doug Davis, Morgan Bauer IBM
  - https://docs.docker.com/v17.09/engine/userguide/eng-image/dockerfile_best-practices/

- Kubernetes
  - https://kubernetes.io/docs/concepts/
  - https://console.bluemix.net/docs/containers/container_index.html#container_index