DOCKER & KUBERNETES BOOTCAMP FOR SALESFORCE SF

Frederik Vos & Pascal Van Dam

December 8, 2019

ITGilde

"Samen Sterker, Beter, Slimmer en Leuker!"

ABOUT THIS COURSE

This course:

- Is developed for Linux professionals that would like to know more about running and managing workload in Docker & Kubernetes on Linux and in the cloud.
- Will introduce you to the core concepts of Docker and Kubernetes
- Enables you to manage the Kubernetes system on a high level
- Enables you to develop simple containerized workloads

CKA
Certified Kubernetes Administrator

CKAD
Certified Kubernetes Developer

See `https://training.linuxfoundation.org`
for more information and the "Certification preparation guide"

- 3 hours, lab-based exam
- All about managing K8S clusters

- 2 hours, lab-based exam
- Focusses on developing containerized workload and transforming workload

AGENDA DAY 1

- » Introduction
- » Why containers?
- » Container Components
- » Docker introduction
- » Docker Registries
- » Docker Volumes and Mounts

AGENDA DAY 2

- Docker Networking
- Docker Compose
- Cleaning up Docker
- Docker Swarm
- Secrets and Configs
- Docker Stack
- Troubleshooting Docker

- Introduction to Kubernetes
- Kubernetes architecture
- Installing Kubernetes
- Installing K8S using kubeadm
- First steps on K8S
- Managing K8S
- Kubernetes Objects
- Kubernetes Networking

AGENDA DAY 4

- Exposing services
- Volumes, Configs and Secrets
- Advanced Networking - Ingresses
- Advanced Networking - Istio
- Installing K8S on AWS/EKS/EKSCTL
- Installing KS using Fargate

AGENDA DAY 5

- Logging
- Monitoring
- Advanced K8S - Statefull Sets
- Troubleshooting
- What next

Getting started with Containers

Getting started with Containers

Why Containers

- In the beginning each application had its own server
  New application needed? $\rightarrow$ New server deployment
- Advantages:
  - Ultimate isolation of applications
  - Very secure
  - Easy to tune the OS for one single application
- Disadvantages:
  - Very expensive
  - Very inefficient (low utilization)
  - Not agile; long time to market

AND THEN THERE WAS VIRTUALIZATION

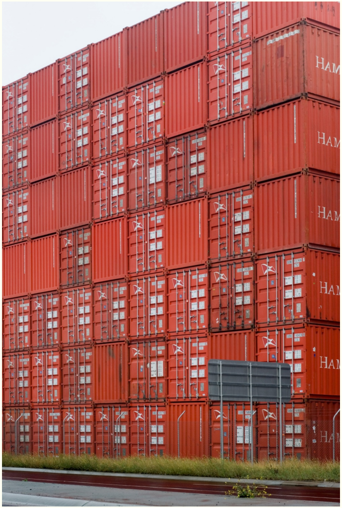
- Hypervisor technology introduced the possibility to run multiple operating systems on one server
- Advantages:
  - Much better server utilization (>80%)
  - Faster time to market
  - More agile
  - Well isolated (security & manageability)
- Disadvantages:
  - Still high CAPEX and OPEX costs for OSes
  - Configuration management challenge (VM Sprawl)
  - Still not fast and agile enough
  - Still a limited number of applications on one OS/server

- Containers allow multiple applications to run in a standardized isolated environment within one single OS
- Advantages:
  - Best utilization/density
  - Less overhead
  - Very agile
  - Blazingly fast time to market
- Disadvantages:
  - Less isolation compared to virtual machines
  - Access to physical/virtual hardware is difficult

Containers are the answer to business desire for having

» Better hardware utilization
» Increasingly faster times-to-market for the applications
» Reduction of risk and complexity while deploying
» A uniform industry standard way of deploying applications
» Having more control on the application environments
» The desire to be as agile as possible

WHAT ARE CONTAINERS



- Resource partition technology
- Very light weight
- An Industry Standard
- Revolutionizing working with applications:
  - Agile workflow from development to production
  - Integration with version control systems
  - Independent features
  - Automatic testing
  - Rapid failback

You may now start with the following labs:

- 1.1 Docker installation
- 1.2 Running Containers

Getting started with Containers

Container Components

- Namespaces
- Cgroups
- Storage
- Networking
- Security Framework

- Partition and isolate a global resource
- Processes in the name space see their own isolated instance
- Similar to `chroot`, extended to other global resources
- Heavily used by containers

- Mount name space image
- Chroot is used as the foundation
- Each container has it’ s own root filesystem (/)

- PID name space image - used in each container
- Each container has it’s own PID 1
- Outside of the container this will be a different PID
- None of the containers can see, start or kill processes on other containers or on the container host

NAMESPACES: USER

- User name space keep a dedicated isolated user database
- Each container has its own user database. For instance UID 0 (root) in the container is not UID 0 outside of the container

- **Inter Process Communications (IPC)**: Partitioning and Isolation of SysV IPC like shared memory, semaphores and message queues
- **Networking**: Partitioning and Isolation of the networking stack Processes within the namespace have the experience of seeing their own network stack, independent of the container host stack
- **UTS**: (Unix Timesharing System) Allow processes in a namespace to have their own hostname and (NIS) domain name

- By default on Linux and UNIX all processes are equal But some processes are more equal than others
- CGroups are resource pools to share and limit resources
- Processes are wrapped in CGroups or Child CGroups
- CGroups available for cpu, blkio, mem, network and devices
- Heavily used in systemd and in container technology

CAPABILITIES (1)

- Linux users are privileged user (root) or non-privileged
  - No restrictions apply to root
  - Regular users have restricted possibilities
- To get enough permissions, a process is often started as root
- Capabilities: allow fine-grained elevated privileges to non-privileged users

CAPABILITIES (2)

- Containers utilize capabilities to get access to privileged functions
- Some examples:
  - `CAP_NET_SERVICE_BIND`: to allow binding of network ports < 1023
  - `CAP_MKNOD` to allow creation of device nodes
  - `CAP_CHOWN` to allow changing ownership of files
- See `man 7 capabilities` for more information

Containers need storage to:

- Store the container images (filesystem, binaries, libs)
- Store the persistent data *(optional)*

STORING CONTAINER IMAGES (1)

- Container images are stored in layers
- Copy-on-write (CoW) mechanism
- Each layer stacks upon the previous layer
- Only the top layer is writable
- Different vendors are using different storage drivers:
  - Devicemapper (direct-lvm) → RHEL, Fedora and CentOS
  - AUFS → Ubuntu
  - BTRFS → SLES, OpenSUSE LEAP

STORING CONTAINER IMAGES (2)

- CoW makes starting and restarting containers very fast
- Changes are light-weight in the container images (stacked)
- Designed for storage efficiency, not speed

You may now start with the following labs:

» 1.3 Create your own images
» 1.4 Experiments with persistency

Getting started with Containers

Docker Introduction

DOCKER CONTAINER TECHNOLOGY

- Initial release in 2013
- Open-Source but strictly managed by Docker Inc.
- Provides GIT-like semantics and interface
- Docker's light weight images are built upon immutable FS layers
- Docker Hub, Docker Registry, Docker Datacenter
- Orchestration and clustering possible with Swarm or 3th party software

- **Container image:** The contents / package that can be run in a container. Base OS + your application.
- **Base image:** Only the fresh OS, not additional layers
- **Image layer:** Each change will result in a new layer stacked upon the container image
- **Container:** Running instance of an image; e.g. your running application / service
- **Container Host or Docker Host:** The host where Docker containers are running
- **Docker Hub:** Generic image store on the Internet
- **Docker Registry:** An (optional) on-premise image store

INSTALLING DOCKER (1)

- Docker can be installed from the distribution repositories or from the Docker website
- Two Editions:
  - Docker CE (Community Edition)
  - Docker EE (Enterprise Edition)
- Pre-requisites:
  - Min. 2Gb of RAM
  - Min. 3Gb of storage for container images
  - Optional: storage for persistent volumes

INSTALLING DOCKER CE (2)

RHEL based distributions:

- Install dependencies:
  - device-mapper-persistent-data
  - lvm2
- Add the repository using `yum-config-manager`
- Install docker-ce with `yum`

Debian based distributions:

- Install `apt-transport-https` to be able to access https repositories
- Import the the GPG key, using `apt-key`
- Add the repository using `add-apt-repository`
- Install docker-ce with `apt`

1. Start and enable Docker:

   ```
   systemctl enable --now docker
   ```

2. Add the user(s) that need to administer Docker to the docker group:

   ```
   sudo usermod -aG docker <user>
   ```

3. Logout and login again
4. Any user belonging to the docker group can run the Docker commands
5. Verify Docker status with such a user:

   ```
   docker info
   ```

- Pull an image in a container from Docker Hub and run it:

```
docker run <image name>
```

- Only pulling an image from Docker Hub

```
docker pull <image name>
```

- Start a container

```
docker start <container name>
```

- Stopping a container

```
docker stop <container name>
```

- Removing a container

```
docker rm [-f] <container name>
```

- Restart a container

```
docker restart <container name>
```

- Search images by name on Docker Hub:

```
docker search <name>
```

- Filter on official images:

```
docker search --filter "is-official=true"
```

- Other filters:
  - Rating: `stars=`
  - Automated builds `is-automated=true|false`

BUILD A DOCKER IMAGE

- Docker images can be pulled from Docker hub or created by hand:

  ```
  docker build
  ```

- This command build images, using a Dockerfile as an input file and built the images given the commands in this file
- The Dockerfile contains the commands how to build the image

```dockerfile
FROM ubuntu:latest
MAINTAINER Pascal van Dam (pascal@yunix.org)

RUN apt-get update
RUN apt-get install -y python python-pip wget
RUN pip install Flask

WORKDIR /home

ADD hello.py /home/hello.py

CMD ["python","./hello.py"]
```

- Build the image:

```
docker build . --tag dockertest:latest
```

- Verify if the built image is locally available now:

```
docker image ls | grep dockertest
```

- Try it:

```
docker run dockertest:latest
```

- Next steps:
  - Push it to a local registry or Docker Hub

LAYERS OF A DOCKER IMAGE

- Docker images will be built layer by layer
- Each command will generate a layer
- Tip: Limit the number of layers by grouping commands with compound statements

Dockerfile

```
FROM node

WORKDIR /app
COPY package.json .

RUN npm install express -save && npm install && mkdir /app/public
COPY helloworld.js /app/
COPY public/* /app/public/

EXPOSE 8081
CMD [ "node", "helloworld.js" ]
```

- At container startup you can run the container in interactive mode:

  ```
  docker run -it <image> --name <container>
  ```

- Leave the interactive mode using Ctrl + q , Ctrl + p
- Invoke a Bash shell, when the container is already running:

  ```
  docker exec -it <container name> /bin/bash
  ```

DEBUGGING DOCKER CONTAINERS (2)

» Get the stdout/stderr info from the containers console:

```
docker logs <container>
```

» Deep dive into the container configuration, to get information about:

  - Network information
  - Volume information
  - Image information

```
docker inspect <container>
```

Getting started with Containers

Docker Registries

- It is possible to store and retrieve images from a registry
- Docker Hub `https://hub.docker.com`
- A third Party registry (ACR, ECR etc)
- a private registry (docker, harbor)

DOCKER HUB

- Docker Hub is a registry owned by Docker INC.
- Can be used for publicly and privately
- Information about containers can be found on Docker Hub website:
  - Dockerfile
  - How to configure and the use image
  - Tips and tricks
  - Related images

- Search for an image:

```
docker search nginx
```

- More detailed search:

```
docker search --filter "is-official=true" --no-trunc nginx
```

- Download and run:

```
docker run --name nginxc01 -p 80:80 nginx
```

DOCKER HUB: STORING IMAGES

- Create a free account must be created on `https://hub.docker.com`
- Storing an image on Docker Hub:
  1. Log in with your Docker Hub account:

     ```
     docker login -u <username> -p <password>
     ```
  2. Tag your image for storing on Docker Hub:

     ```
     docker tag <id> <accountname>/<image>:versiontag
     ```
  3. Push the image

     ```
     docker push
     ```
  4. Verify that both local and remote images are listed:

     ```
     docker images
     ```

DOCKER PRIVATE REGISTRIES (1)

- Secure or in-secure
- Authentication is an option for secure registries
- Protocol used is HTTPS
- There are alternatives for the Docker Registry like Harbour or Quay.io

DOCKER PRIVATE REGISTRIES (2)

What is needed for a simple in-secure registry?

» A docker node to run the registry container on
» A TCP port on which the registry will listen
» Persistent storage to store the container images
» The **registry:2** image from Docker Hub

CREATE A PRIVATE REGISTRY

```
sudo docker run --detach \
  --restart=always \
  --name registry \
  --publish 5000:5000 \
  --volume /srv/registry:/var/lib/registry \
  registry:2
```

To use an in-secure registry we have to declare it as 'trusted' in the **/etc/docker/daemon.json** file. After that the docker daemon needs to be restarted.

/etc/docker/daemon.json

```
{
  "insecure-registries" : [ "st99node01:5000", "st99node01.itgildelab.net:5000"]
}
```

restart docker daemon

```
sudo systemctl restart docker
```

What is needed for a secure registry?

- A Docker node to run the registry container on
- A TCP port on which the registry will listen
- Persistent storage to store the container images
- SSL certificate(s) and key
- CA certificate must be added to `/etc/docker/certs.d`

First create the needed certificate and private key

```
mkdir certs
cd certs

openssl req -new -sha256 -newkey rsa:4096 -x509 -sha256 \
 -nodes -days 365 -out registry.crt -keyout registry.key \
 -subj "/C=NL/ST=LB/O=Acme, Inc./CN=registry.itgilde.lab"
```

CREATE A SECURE REGISTRY

Spin-up the container using the created certificate and key.

```
sudo docker run -d \
  --restart=always \
  --name registry \
  -v ${PWD}/certs:/certs \
  -e REGISTRY_HTTP_ADDR=0.0.0.0:443 \
  -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/registry.crt \
  -e REGISTRY_HTTP_TLS_KEY=/certs/registry.key \
  -p 443:443 \
  -v /srv/registry:/var/lib/registry \
  registry:2
```

On every docker client, create a directory under /etc/docker/certs.d and place the certificate in it. If the port is unequal to 443 please also specify the port in the URL directory name.

```
mkdir -p /etc/docker/certs.d/st99node01.itgildelab.net
cp certs/st99node01.itgildelab.net.crt /etc/docker/certs.d/st99node01.itgildelab.net
```

You may now start with the following labs:

» 1.5 Creating registries

Getting started with Containers

Volumes and Mounts

BIND MOUNTS

Since the early days of Docker there has been the concept of bind mounts.

- A file or directory from the host filesystem is mounted in the container
- Has limited functionaly compared to volumes
- Use `-v` or `--volume`

EXAMPLE: BIND MOUNT WITH VOLUME OPTION

```
docker run --detach --interactive --tty \
  --name devtest \
  --volume $(pwd)/html:/usr/share/nginx/html \
  nginx:latest
```

- The `--volume` is only supported in stand-alone containers
- `--mount` works for stand-alone containers and in swarm mode
- In general `--mount` is more explicit and verbose

EXAMPLE: BIND MOUNT WITH MOUNT OPTION

```
docker run --detach --interactive --tty \
  --name devtest \
  --mount type=bind,source=$(pwd)/html, \
    target=/usr/share/nginx/html \
  nginx:latest
```

Volumes are the preferred way to supply persisting storage to containers.

- Volumes are easier to backup than bind mounts
- Volumes can be managed using the Docker CLI
- Volumes work on Linux and Windows containers
- Volumes can be shared in a more safe way between containers
- Volume drivers are available for external storage provisioning
- New volumes can be pre-poplulated by a container

USING VOLUMES

- Create a volume:

```
docker volume create <volume name>
```

- List volumes:

```
docker volume ls
```

- More details of a volume:

```
docker volume inspect <volume name>
```

- Remove a volume:

```
docker volume rm <volume name>
```

If you start a container with a volume that does not exist yet, Docker will create it for you.

```
docker run -d \
  --name devtest \
  --mount source=myvol2,\
    target=/usr/share/nginx/html \
  nginx:latest
```

Getting started with Containers

Cleaning up the Docker Room

```
# Clean up dangling images
docker image prune

# Clean up all unused images
docker image prune -a

# Prune images which are older dan 1d
docker image prune -a --filter "until=24h"
```

CLEANING UP CONTAINERS

```
# Removing the container after exit
docker run --rm -detach --name <name> <image>

# Clean up old containers
docker container prune
```

```
# Clean up unreferenced volumes
docker volume prune

# Label a volume

docker volume create --label <label> <volume name>

# Clean up volumes that do not have a specific label
docker volume prune --filter "label!=<label>"
```

```
# Clean up unreferenced networks
docker network prune
```

CLEANING UP ALL DOCKER OBJECTS

```
# Clean up all unreferenced docker objects except volumes
docker system prune

# Clean up all unreferenced docker objects including volumes
docker system prune --volumes
```

Introduction to Kubernetes

Introduction to Kubernetes

History

- Started as GOOGLE project Borg.
- Opensourced and relased as Kubernetes
- in Ancient greek: κυβερνήτης
  - Meaning: Helmsman, navigator, pilot
- Google Project Seven
- Founded by Joe Beda, Brendan Burns and Craig McLuckie
- Maintained by the Cloud Native Computing Foundation (CNCF)
- Popular referenced as K8S. ('Kates')

Incorporated in many solutions:

- RedHat OpenShift
- Rancher 2
- MESOSPHERE - DC/OS
- Azure: Azure Kubernetes Service
- ECS: Elastic Container Service
- GKE: Google Kubernetes Engine
- IKS: IBM Kubernetes Service

Introduction to Kubernetes

Kubernetes Architecture

- Master/slave architecture
- Kubernetes Control Plane v.s. Worker Nodes
- Composed of Building Blocks (Primitives)
  - Deploying applications
  - Maintaining applications
  - Scaling applications
- Loosely coupled
- All revolving around the Rest API
- Extensible by API, containers and extensions.

PRIMITIVES

- Pods
- Labels and selectors
- Controllers
- Services

PODS

- Basic scheduling unit in Kubernetes
- Contains one or more containers that are scheduled together
  - Guaranteed to be co-located on the same host
  - Can share resources together
- Has a unique IP address
- Share network stack and volumes
- Can be managed manually using the rest API or by a controller

- Labels are key-value pairs that can be attached to API objects like
  - Pods
  - Nodes
- Label selectors are queries against labels
- Example use cases:
  - Select to which pods traffics is routed to.
  - Select which pods get updated/scaled up/down etc.
- Always use labels!

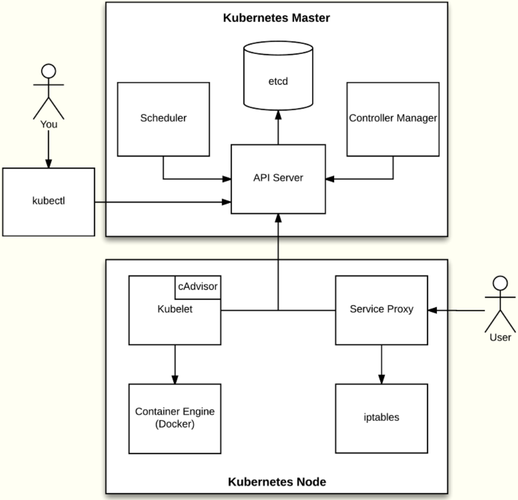
CONTROLLERS

- Managed by the Control Manager
- A control loop that watches the shared state
- Makes changes to move the current state towards the desired state
- Example controllers:
  - Replication controller
  - DaemonSet controller
  - Job Controller

- Logical set of PODs
- Provides a single IP address and DNS name by which PODs can be accessed
- Helps with LoadBalancing
- Types of services
  - ClusterIP: access only from within the cluster
  - NodePort: access from outside the cluster on a static port
  - LoadBalancer: Uses cloud providers' Load Balancer facility

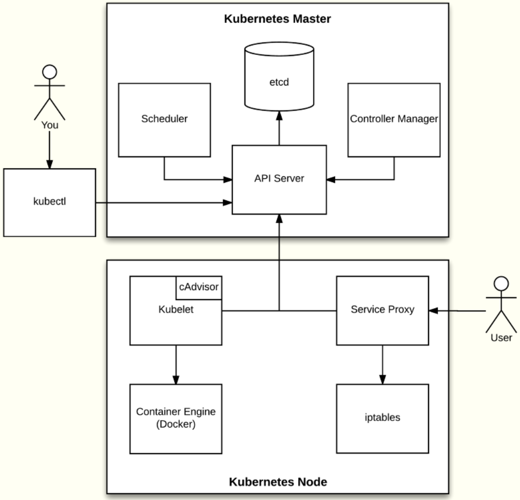
- API server
- ETCD
- Scheduler
- Controller Manager

KUBERNETES CONTROL PLANE - MASTER

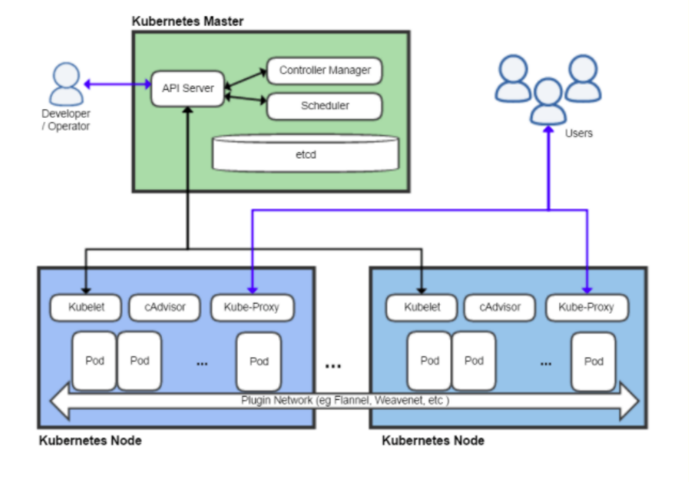


- Kubelet - Controls state/manages containers
- Container - contains the application
- Kube-proxy - routes IP traffic to container

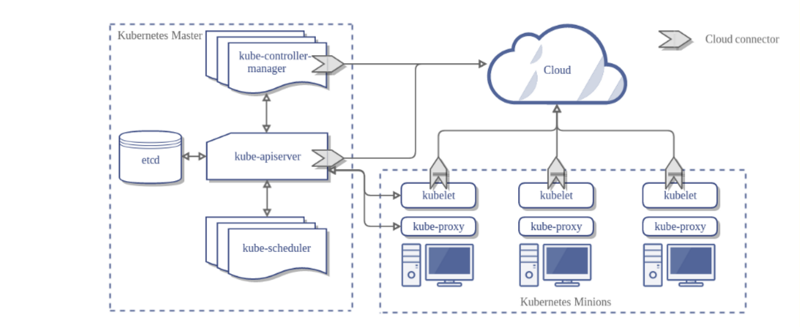
KUBERNETES WORKER NODE - NODE



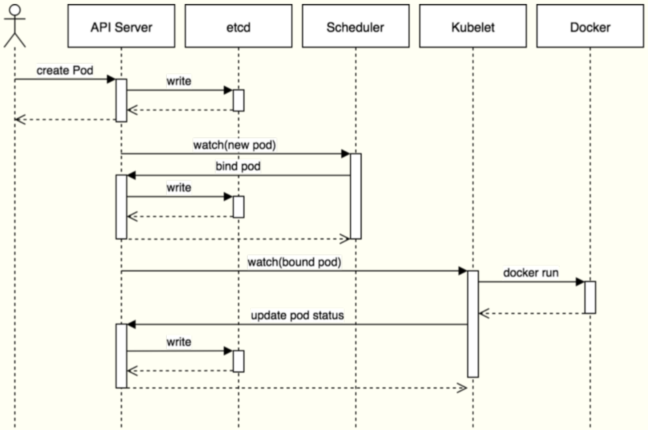
KUBERNETES ARCHITECTURE

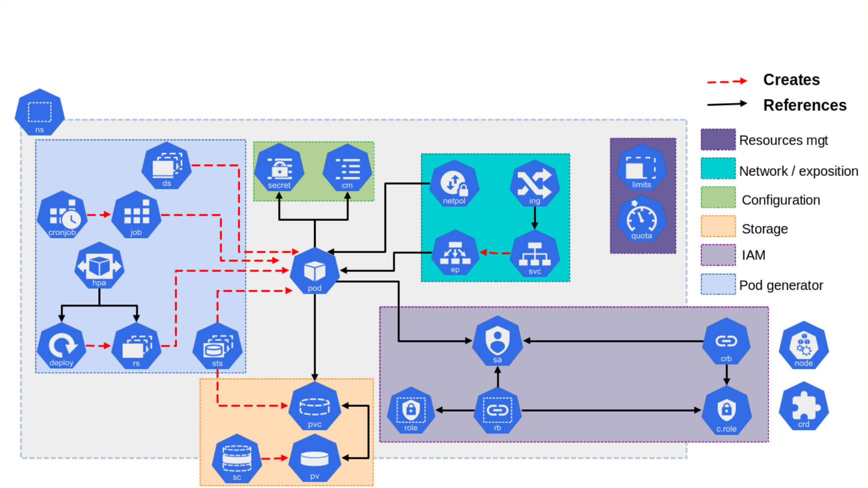


KUBERNETES ARCHITECTURE



EXAMPLE OF POD CREATION WORKFLOW




Creates
References
Resources mgt
Network / exposition
Configuration
Storage
IAM
Pod generator
ns
ds
cronjob
job
hpa
deploy
rs
sts
secret
cm
pod
netpol
ing
ep
svc
limits
quota
pvc
sc
pv
sa
crb
role
rb
c.role
node
crd

LEGEND OF ALL K8S 1.13.4 RESOURCES 1/2

- PO - POd
- RS - ReplicatSet
- DEPLOY - DEPLOYment
- HPA - Horizontal Pod Autoscaler
- STS - StaTeful Set
- CRJ - CronJob
- JO - Job
- SVC - SerViCe
- CRD - Custom Resource Definition
- RC - Replica Controller (deprecated)

LEGEND OF ALL K8S 1.13.4 RESOURCES 2/2

- EP - End Point
- PV - Persistent Volume
- PVC - Persistent Volume Claim
- SC - Storage Class
- CM - ConfigMap
- SECRET - SECRET
- DS - Daemon Set
- NETPOL - NEtwork Policy

Introduction to Kubernetes

Installing Kubernetes

- Fully from scratch
- Minikube
- Kubeadm
- Using a Kubernetes service on a public cloud

- Works on so called bare metal servers
- Supports latest kubernetes version (1.13)
- Can create single and multi master K8S clusters
- Can facilitate upgrade to newer K8S version

PRE-REQUISITES FOR KUBEADM/KUBERNETES

- Docker engine (Docker CE 18.06 recommended) installed and running
- Swap must be turned off
- At least 2GiB of RAM

- Disable swap *(Don't forget to edit /etc/fstab)*

  ```
  sudo swapoff -a
  ```

- Install the Docker container runtime and start the engine as discussed in the Docker introduction
- Configure systemd as the recommended driver for Docker

SYSTEMD AS DOCKER DRIVER (1)

```
su -

cat > /etc/docker/daemon.json <<EOF
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m"
  },
  "storage-driver": "overlay2"
}
EOF
```

SYSTEMD AS DOCKER DRIVER (2)

```
# Directory for control files
mkdir -p /etc/systemd/system/docker.service.d


# Restart docker.
systemctl daemon-reload
systemctl restart docker
```

INSTALLING KUBERNETES USING KUBEADM (2)

- Import the GPG key:

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg \
  | sudo apt-key add -
```

- Add the repository (Debian and Ubuntu, all versions)

```
sudo apt-add-repository "deb http://apt.kubernetes.io/ \
  kubernetes-xenial main"
```

- Install the software:

```
sudo apt update
sudo apt install -y kubeadm kubectl kubelet
```

INSTALLING KUBERNETES USING KUBEADM (3)

Configure the master and define the subnet:

```
kubeadm init --pod-network-cidr <private subnet>
```

Please save the output of this command!

In case of reported issues by kubeadm

- If `kubeadm` report sizing issues, add the parameter:

  `--ignore-preflight-errors=<list of errors>`

- If you are using another Docker registry, add the parameter:

  `--image-repository <url>`

Do not execute `kubeadm` as root! As a normal user execute:

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Check the current state:

```
kubectl get nodes
```

The master node will be listed a ’Not Ready’, because the network is not configured.

- Kubernetes is an orchestrator for containers
- It doesn't manage networks
- You'll need a Container Network Interface (CNI)
- Many CNI's are available, such as:
  - Calico
  - Weave
  - Flannel

On Azure Calico will not work properly so we will use CANAL

```
kubectl apply -f \
   https://docs.projectcalico.org/v3.10/manifests/calico.yaml
```

```
kubectl apply -f \
  https://docs.projectcalico.org/v3.10/manifests/canal.yaml
```

- Confirm that all of the pods are running:

```
watch kubectl get pods --all-namespaces
```

- And review the master availability:

```
kubectl get nodes
NAME        STATUS   ROLES    AGE   VERSION
debian      Ready    master   47h   v1.13.1
```

PREPARE THE WORKER NODES

- Execute the same procedure as for the Master
- Join the Kubernetes Master, using the saved output from the Master installation.

```
kubeadm join --token <token> <master-ip>:<master-port> \
  --discovery-token-ca-cert-hash sha256:<hash>
```

```
kubectl get nodes

NAME         STATUS   ROLES    AGE   VERSION
st00node01   Ready    master   47h   v1.13.1
st00node02   Ready    <none>   47h   v1.13.1
st00node03   Ready    <none>   46h   v1.13.1
```

Managing K8s

Managing K8S

First steps on K8S

RUNNING A POD (1)

- In K8S we don’t run containers, we run PODs
- Use `kubectl run` to start a POD.
- This is the adhoc use of Kubernetes
- This functionality will be deprecated.

RUNNING A POD (2)

```
kubectl run nginx --image nginx:latest --replicas=1
```

Use `kubectl get pods` to examine the results

```
kubectl get pods
kubectl get pods --namespace default
kubectl get pods --all-namespaces
kubectl get pods --namespace kube-system
```

MORE WAYS TO EXAMINE PODS

```
kubectl get pods -o wide
kubectl describe pod
```

DELETING A POD

```
kubectl delete pod <pod id>
```

EXAMINING THE REPLICATION CONTROLLER

```
kubectl get rc nginx
kubectl describe rc nginx
```

SCALING UP

```
kubectl scale --replicas=3 rc nginx
kubectl get pods -o wide
kubectl delete pod <pod-id>
kubectl get events | head -10
```

```
kubectl delete rc nginx
kubectl get pods -o wide
```

Managing K8S

Introduction to kubectl

- Kubectl is your one-stop-shop tool for managing K8S clusters
- User interface of `kubectl` is very untuitive
- General syntax: `kubectl <verb> <object> <options>`

```
kubectl get pods -o wide
kubectl get pods -o yaml --export
kubectl describe deployment
```

Kubectl knows the following `verbs` for read-like actions

- `describe`
- `get`

```
kubectl describe deployment mydeployment
kubectl get nodes
kubectl get pod -o wide
```

Kubectl knows the following `verbs` for create-like actions

- `create`
- `run`

```
kubectl create -f mydeployment.yml
kubectl run nginx --image=nginx --port=80
```

Kubectl knows the following `verbs` for deleting objects

- `delete`

```
kubectl delete pods -l myapp
kubectl delete svc nginx-service
```

Kubectl knows the following `verbs` for updating objects

- `set`
- `label`
- `annotate`
- `scale`
- `edit`

```
kubectl scale svc nginx-serice --replicas=3
kubectl edit deployment myapp
kubectl set image deployment.v1.apps/nginx-deployment
        nginx=nginx:1.9.1
```

Kubectl knows the following `objects`

- `nodes`
- `pods`
- `deployments`
- `services`
- `rc` to manage replica controllers
- `rs` to manage replica sets

```
kubectl expose deployment q10rv2 --type NodePort --port 5000
```

- Managing K8S using imperative commands.
- Managing K8S using imperative object confguration.
- Managing K8S using declarative object configuration.

- Easiest way to operate you cluster, good for starting
- With `kubectl` you can operate directly on live objects
- Useful for one-off tasks

```
kubectl run nginx --image nginx:1.7.1 --port=80 --replicas=3
kubectl set image deployment nginx nginx=nginx:1.9.1
kubectl scale deployment nginx --replicas=1
```

IMPERATIVE CONFIGURATION (1)

- More difficult, but more powerful
- A YAML file that describes the new object or how it should be altered
- Describe the desired state of the object(s) and have the controllers sort it out
- Create the object(s):

```
kubectl create -f <yaml cfg file>
```

- Delete the object(s):

```
kubectl delete
```

- Create / Update the object(s)

```
kubectl apply
```

```
kubectl get deployment nginx -o yaml --export > nginx.yml
kubectl delete -f nginx.yml
kubectl create -f nginx.yml
kubectl replace -f nginx.yml
kubectl create -f http://myrepo.itgilde.lab/calico.yml
```

- Configuration to reach the desired state
- Complex to manage and to design, but very powerfull
- Describe the desired state using a directory of manifest files and have K8S controllers figure it out
- Will be extensively discussed in the Advanced Kubernetes Course

```
kubectl apply -f config/
kubectl apply -R -f config/
kubectl diff -R -f config/
```

HANDY KUBECTL COMMANDS (1)

- Get help

  ```
  kubectl get pods --help
  ```

- Show logs of a pod

  ```
  kubectl logs <pod>
  kubectl logs <pod> -c <container>
  ```

- Explain the yml config file format for an object

  ```
  kubectl explain <object>
  ```

HANDY KUBECTL COMMANDS (2)

```
kubectl explain pod
kubectl explain pod.spec
kubectl explain pod.spec.volumes
```

Together with `explain`, `kubectl run` can serve as an easy template generator

» For a deployment:

```
kubectl run nginx --image nginx:1.15.1 --port 80
```

» For a bare pod:

```
kubectl run nginx --image=nginx --port 80 --restart=Never
```

» For a cron job:

```
kubectl run busybox --image=busybox \
--schedule="* * * * *"  --restart=OnFailure
```

Object Manifests

Each object in K8S can be described using a manifest

» The manifest file is writting in YAML

» The manifest file has at least 4 parts: AKMS

- A API (version)
- K KIND (kind of object)
- M METADATA (labels, annotations etc)
- S SPEC (object specifications and attr)

EXAMPLE DEPLOYMENT MANIFEST (1)

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
spec:
  replicas: 3
      labels:
        app: myhelloworld
```

EXAMPLE DEPLOYMENT MANIFEST (2)

```
  spec:
    containers:
    - image: pamvdam/myhelloworld:v0.4
      imagePullPolicy: Always
      name: myhelloworld
      ports:
      - containerPort: 8081
        name: http
        protocol: TCP
    resources: {}
    restartPolicy: Always
```

- List a deployment specified in YAML format and redirect it to a file

```
kubectl get deployment <label> -o yaml --export > file
```

- This file can be edited and used to create a new deployment

Kubernetes Networking

- Kubernetes is a great at orchestarting PODs
- Kubernetes does not do POD-to-POD communication
- This is the task for Container Network Interfaces (CNI)

- All Pods can communicate with all other Pods without using network address translation (NAT)
- All Nodes can communicate with all Pods without NAT
- The IP that a Pod sees itself as is the same IP that others see it as

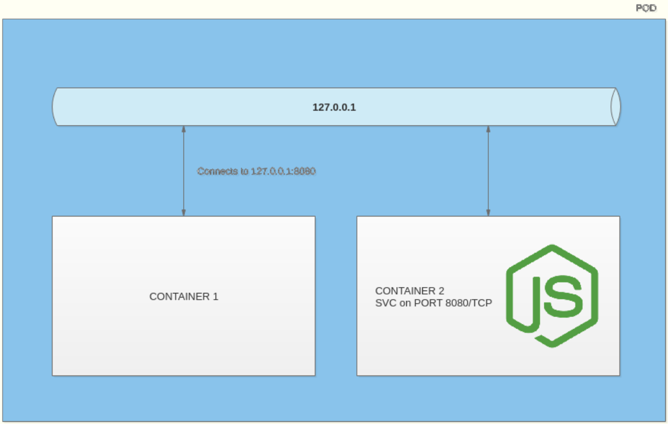
KUBERNETES NETWORKING PROBLEMS TO SOLVE

- Container-to-Container networking
- Pod-to-Pod networking
- Pod-to-Service networking
- External-to-Service networking

Container-to-Container networking is simple:

- Containers in the same POD share the same NS namespace
- So they share the same IP address
- Containers can communicate via the local loopback network
- Please mind there is only one PORT space

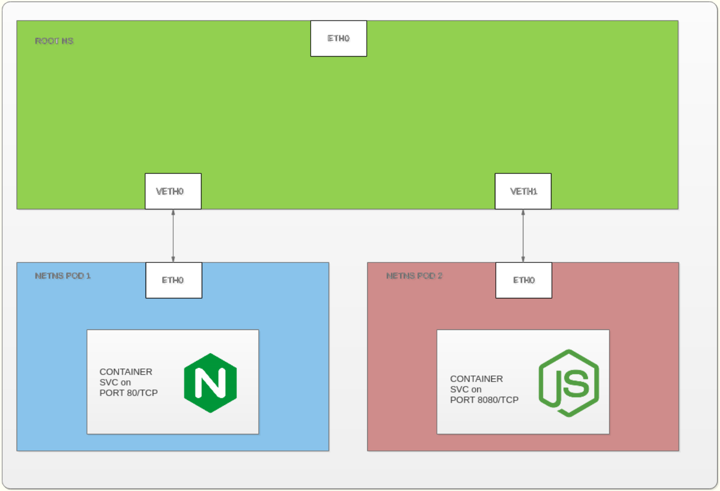
CONTAINER-TO-CONTAINER NETWORKING



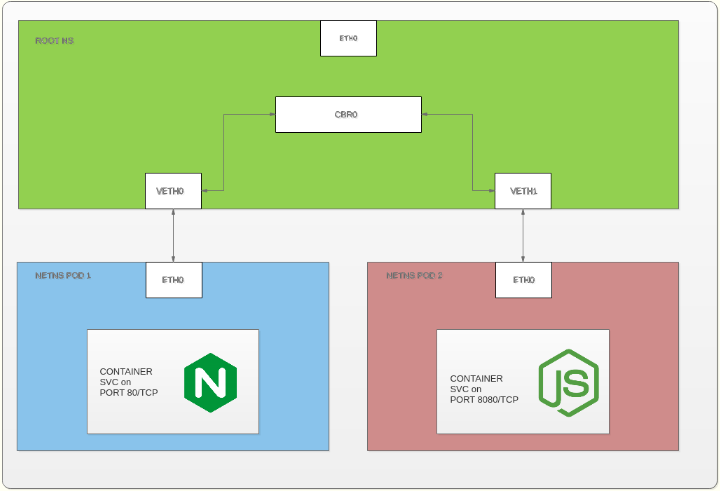
Pod-to-Pod networking has the following 2 cases:

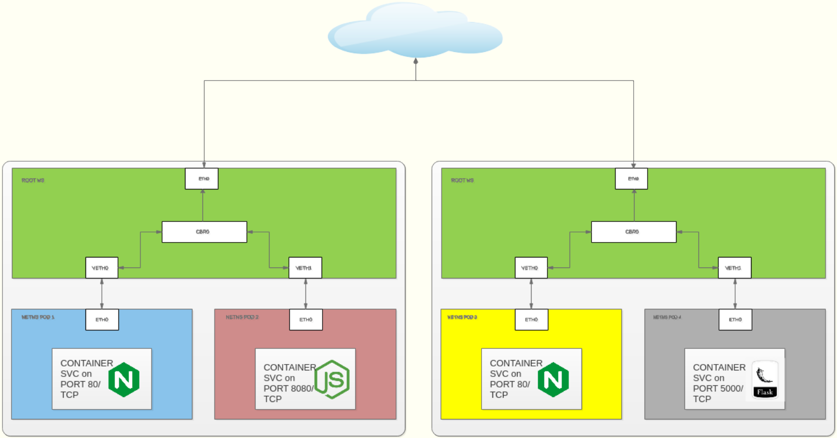
- Pod-2-Pod on the same node
  - Network traffic stays on the same node
  - The container bridge will be used to communicate to the other pod
- Pod-2-Pod traffic across nodes
  - Network traffic will leave the node
  - The container bridge will be used
  - Using the node's ethernet adapter the pod on other node is contacted

POD-TO-POD NETWORKING



POD-TO-POD NETWORKING SAME NODE

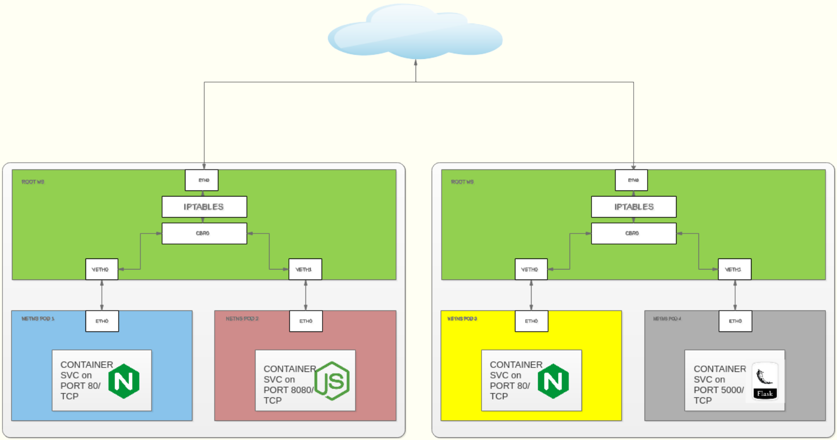



ROOT NS
ETH0
CBR0
VETH0
VETH1
NETNS POD 1
ETH0
CONTAINER
SVC on
PORT 80/
TCP
NETNS POD 2
ETH0
CONTAINER
SVC on
PORT 8080/
TCP
ROOT NS
ETH0
CBR0
VETH0
VETH1
ETH0
CONTAINER
SVC on
PORT 80/
TCP
ETH0
CONTAINER
SVC on
PORT 5000/
TCP
Flask

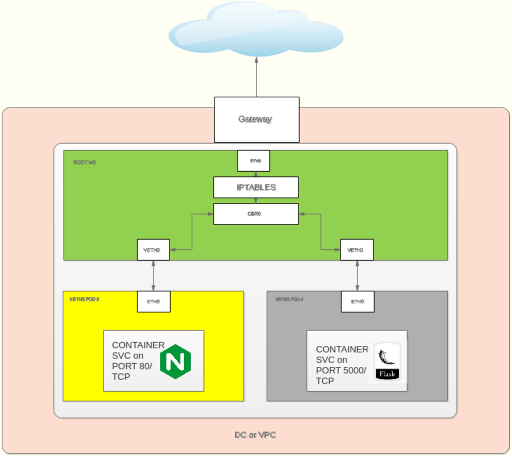
As PODs are volatile so are their IP addresses. Hence we need a solution to address the ports using more stable resource. E.g. `services`

- Outgoing and incoming traffic is still using the container bridge
- IPTABLES is loadbalancing and forwarding the request to the proper set of `pods` belonging to the `service`

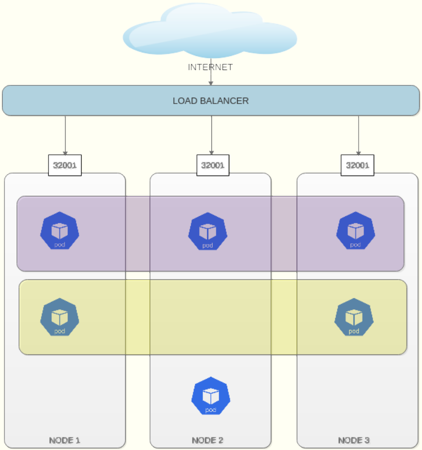
POD-TO-SERVICE AND SERVICE-TO-POD NETWORKING



EGRESS - SERVICE TO EXTERNAL NETWORKING



INGRESS - EXTERNAL TO SERVICE NETWORKING



The container networking work is done by a CNI plugin. There are many network plugins for K8S available:

- Flannel
- Weave(net)
- Calico
- Canal
- Romana
- Cilium
- Etc..

- Most simple of all CNU plugins
- Easy to setup
- Works on public clouds
- No network policies supported
- L2, no IPv6, VXLAN

CNI PLUGIN - WEAVENET

- Network policies supported
- L2, IPv4 + IPv6, VXLAN
- Supports encryption

- Network policies supported
- Works on L3 layer (IPinIP, BGP)
- Support IPv4+IPv6

CNI PLUGIN - CANAL

- Hybrid form of Calico with Flannel
- Networking from Flannel
- Network Policies by Calico
- IPv4, VXLAN, L2
- Easy to setup

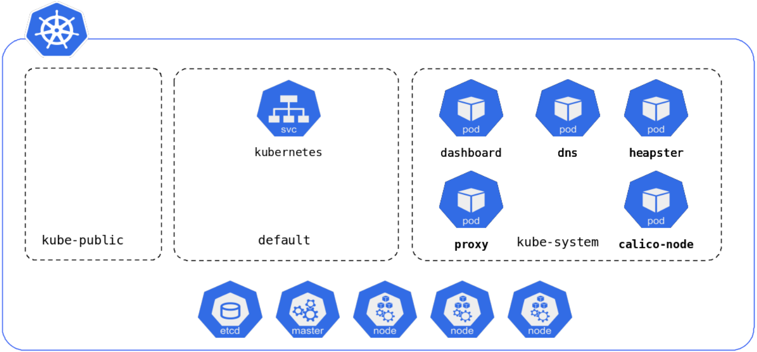
Namespaces

NAMESPACES

Kubernetes allows the ‘physical cluster” to be split up in multiple virtual clusters. This is done by creating so called `namespaces`. These `namespaces` are different from the Linux kernel `namespaces`. After the K8S cluster has been initialized by `kubeadm` you will find 3 pre-created `namespaces`

- The `kube-system` namespace containing all Kubernetes infra objects
- The `kube-public` namespace is specific for `kubeadm`
- The `default` namespace

The standard namespaces available after bootstrapping with `kubeadm`. The `kube-system` is populated with the k8s-infra objects



EXAMPLE OF LIST OF PODS RUNNING IN KUBE-SYSTEM NAMESPACE

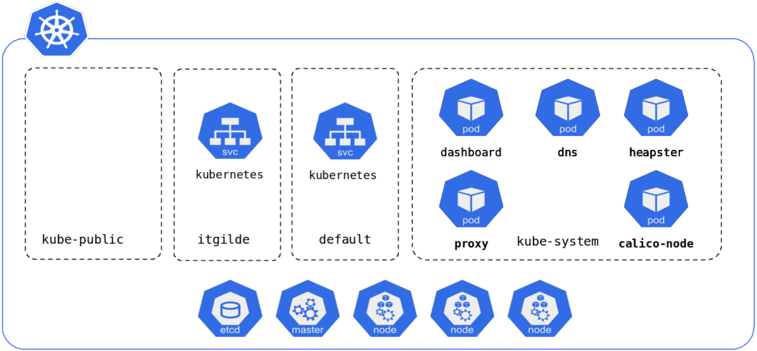
```
kubectl get pods -n kube-system

NAME                               READY   STATUS    RESTARTS   AGE
calico-node-qvvvc                  2/2     Running   0          124m
coredns-86c58d9df4-zdnhr           1/1     Running   0          126m
etcd-kub14n01                      1/1     Running   0          125m
kube-apiserver-kub14n01            1/1     Running   0          125m
kube-controller-manager-kub14n01   1/1     Running   0          125m
kube-proxy-hnsnk                   1/1     Running   0          125m
kube-scheduler-kub14n01            1/1     Running   0          125m
```

- List namespaces with `kubectl get ns`
- Create a namespace with `kubectl create ns <namespace>`
- Delete a namespace with `kubectl delete ns <namespace>`
- List all pods in a namespace using `kubectl get pods -n <namespace>`
- List all pods in all namespaces using `kubectl get pods --all-namespaces`
- Create an object in a namespace use `kubectl create -f mypod.yml -n <namespace>`

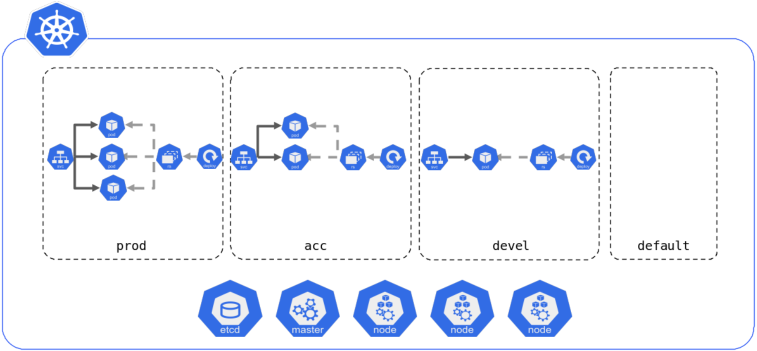
CREATE A NAMESPACE (1)

```
kubectl create ns itgilde

kubectl get ns

NAME          STATUS   AGE
default       Active   136m
kube-public   Active   136m
kube-system   Active   136m
itgilde       Active   2s
```

CREATE A NAMESPACE (2)



CREATE A NAMESPACE (3)



- K8S provides means for Service Discovery using `kubedns`
- When a service is created to exposed a POD a DNS entry is created
- One can find the POD using: `<servicename>.<namespace>.svc.cluster.local`
- If only `<servicename>` is used, it will try to lookup the service local to the namespace
- Use the FQDN to connect to services across namespaces

OBJECTS THAT ARE NOT LOCAL TO A NAMESPACE

- namespaces
- nodes
- persistentvolumes
- clusterinformations
- podsecuritypolicies
- storageclasses
- volumeattachments
- ...

OBJECTS THAT ARE LOCAL TO A NAMESPACE

- persistentvolumeclaims
- pods
- replicationcontrollers
- services
- daemonsets
- deployments
- replicasets
- ingresses
- …

- Namespaces provide logical partitioning of the Kubernetes cluster
- There's no TRUE `isolation`
- Use `namespaces` to separate workloads
- Use separate clusters to provide `isolation`
- Alternatives enforced CRI like: `gVisor` or `Kata Containers`

- Namespaces are suitable for soft-multitenancy
- Use it for trusted-workloads within one cluster
- If you need hard-multitenancy use
  - Separate workloads on separate nodes
  - Enforced CRI
  - Separate workloads on separate clusters

Kubernetes PODs

Kubernetes PODs

Concept of PODs

PODs

- Hold the containers in K8S
- Can hold 1 or `multiple` containers
- Are the unit of scheduling on K8S
- Get an IP attached to them
- Get volumes attached to them
- Are seldom created 'bare'

Containers in a POD

- Share one IP address
- Share the attached volumes
- Can connect to eachother using the local-loop (127.0.0.1) network

POD manifest in YAML

```yaml
apiVersion: v1

kind: Pod

metadata:

    run: bb

spec:

  containers:

  - args:

    - sleep

    - "3600"

    image: yauritux/busybox-curl

    imagePullPolicy: Always

    name: bb
```

Kubernetes supports 3 type of probes. These probes are configured in the POD spec.

» `startupProbe`

» `livenessProbe`

» `readinessProbe`

STARTUPPROBE

- Monitors the startup of the container
- When failed: the container is killed
- When once succesful: disarms startupProbe and arms readinessProbe / livenessProbe

startupProbe

```
apiVersion: v1
kind: Pod
metadata:
    run: bb
spec:
  containers:
    image: pamvdam/astro:sf1
    name: astro
    startupProbe:
      httpGet:
        path: /health
        port: 8080
      failureThreshold: 30
      periodSeconds: 10
```

- Detects unresponsiveness of the container
- When failed: the container is killed

livenessProbe

```yaml
apiVersion: v1
kind: Pod
metadata:
    run: bb
spec:
  containers:
    image: pamvdam/astro:sf1
    name: astro
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

» Detects if a container is ready to receive network traffic

» When failed: network traffic will not be routed to this container

readinessProbe

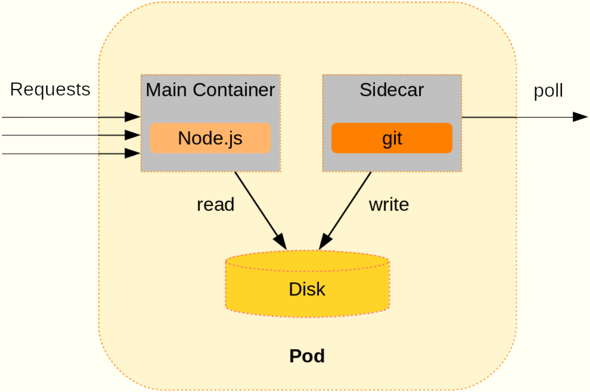
```
apiVersion: v1
kind: Pod
metadata:
    run: astro
spec:
  containers:
    image: pamvdam/astro:sf1
    name: astro
    readinessProbe:
      exec:
        command:
        - cat
        - /tmp/iamready
      initialDelaySeconds: 5
      periodSeconds: 5
```

MULTI-CONTAINER/POD PATTERNS

There are 4 common Multi-Container/POD Patterns

- Sidecar Container
- InitContainer
- Ambassador Container
- Adapter Container

The `Sidecar` pattern allows to extend or augment the functionality of a pre-existing container without changing it.



- Allow for single-purpose reuable containers
- Extending the functionality by using Sidecar containers
- Usecase: initializing an environment for the App containers
- Usecase: keeping App container config updated (nginx)

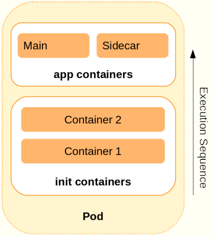
Example Sidecar Container

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-sidecar
spec:
  volumes:
  - name: shared-logs
    emptyDir: {}

  containers:
  - name: app-container
    image: alpine
    command: ["/bin/sh"]
    args: ["-c", "while true; do date >> /var/log/app.txt; sleep 5;done"]

    volumeMounts:
    - name: shared-logs
      mountPath: /var/log
```

Example Sidecar Container

```yaml
- name: sidecar-container
  image: nginx:1.7.9
  ports:
    - containerPort: 80

  volumeMounts:
  - name: shared-logs
    mountPath: /usr/share/nginx/html
```

The `InitContainer` pattern foresees in a means to initalize an environment before the actual application container is run. One could for example clone the most recent website from GIT using an InitContainer and once done have it served by an nginx container



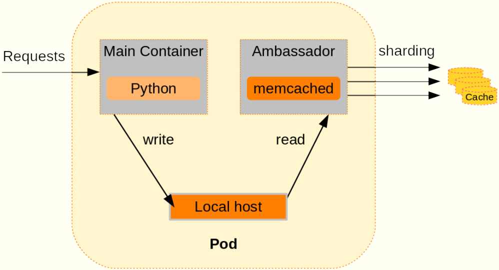
InitContainers

- `initContainers` are special 'containers'
- They have their own 'container' description in the POD manifest
- They are executed first
- While `initConainers` are executed, the other containers are not started
- `initContainers` run to completion
- If an `initContainer` fails the POD is restarted
- `initContainers` are started in the order of appearance

InitContainer

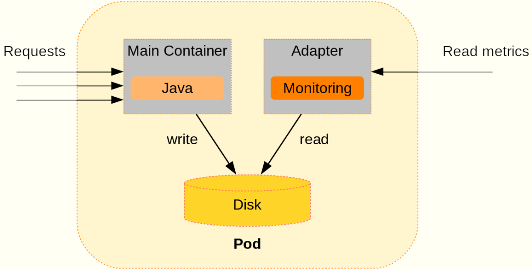
```yaml
spec:
  containers:
  - name: web-server
    image: nginx
  initContainers:
  - name: init-clone-repo
    image: alpine/git
    args:
        - clone
        - --single-branch
        - --
        - https://thegitcave.org/k8s4all/website.repo
        - /usr/share/nginx/html
```

The Ambassador Container pattern is a specialized `Sidecar` pattern that provides a unified interface for accessing services outside of the pod



- Does not enhance the app container
- Provides the function of a smartcache
- Usecase: SSL termination
- Usecase: memcached and twemproxy

The Adapter Container pattern provides a way to have uniform access to a heterogenous system.



MULTI CONTAINER PATTERN 4 - ADAPTER CONTAINER

- Usecase: Expose metrics in a standard way
- Usecase: Expose log records in a standard way
- Actually a specialized case of the `Sidecar` pattern
- Or a reverse `Ambassador` pattern

kubectl and PODs

```
# To look inside a POD, docker exec equivalent

kubectl exec -it <podname> [-c <containername] -- sh

# To execute a command inside a pod

kubectl exec <podname> [-c <containername] -- cat /etc/hosts

# To get the logs from a container in a pod

kubectl logs <podname> [-c <containername]

# To follow the logs from a container in a pod

kubectl logs -t  <podname> [-c <containername]
```

Kubernetes ReplicaControllers

Kubernetes ReplicaControllers

Concept of ReplicaControllers

As stated earlier a POD alone is a POD alone. PODs running unmanaged:

- Cannot scale by being deployed on other nodes
- Do not get restarted when killed

ReplicaControllers

- Manage PODs of the same type in a set
- Ensure that 'n' replicas of the POD keep running
- Make it possible to scale POD workload over multiple nodes
- Ensure that PODs get recreated when failed
- Allow upgrades of container images in PODs

ReplicaControllers can be created adhoc using `kubectl` with the `generator=run/v1` option

```
kubectl run nginx --image=nginx:1.7.1 --generator=run/v1
```

CREATING A REPLICACONTROLLER WITH YAML MANIFESTS

ReplicaControllers are created with YAML manifests:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx-www
spec:
  replicas: 2
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

The YAML manifest of a ReplicaController has the following important sections and attributes:

» `spec.replicas` defines the nr of replicas
» `spec.selector` PODs with this label will be managed by the ReplicaController
» `spec.template` This defines what kind of PODs need to be (re-)created

Tasks on ReplicaControllers

```
# To view replicacontrollers in the K8S cluster
kubectl get replicacontrollers
kubectl get rc

# To describe the state of a replicacontroller
kubectl describe rc frontend

# To delete a replicacontroller
kubectl delete rc frontend
```

- Use of ReplicaControllers is strongly discouraged
- Replaced by ReplicaSets and Deployments
- Still can be found on old K8S YAML files
- ReplicaSets offer more flexibility with `Selectors`
- Updates on ReplicaControllers are client side based
- Updates on Deployments are server side based
- `Deployments` provide more reliable upgrades

Kubernetes ReplicaSets

Kubernetes ReplicaSets

Concept of ReplicaSets

As stated earlier a POD alone is a POD alone. PODs running unmanaged:

- Cannot scale by being deployed on other nodes
- Do not get restarted when killed
- Allow loadbalancing over the available PODs

ReplicaSets

- Manage PODs of the same type in a set
- Ensure that 'n' replicas of the POD keep running
- Make it possible to scale POD workload over multiple nodes
- Ensure that PODs get recreated when failed
- Are an evolutionary progress on ReplicaControllers

ReplicaSets vs ReplicaControllers

- ReplicaSets are 'nextgen' ReplicaControllers
- ReplicaSets have a set-based selector
- ReplicaSets update PODs using `rollout` command or using `deployments`
- ReplicaControllers update PODs using `rolling-update` command.
- ReplicaSets deliver 'declarative' control
- ReplicaControllers deliver 'imperative' control

ReplicaSets are created with YAML manifests:

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: todo
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: todo-frontend
        image: todo:v0.1
```

The YAML manifest of a ReplicaSet has the following important sections and attributes:

- `spec.replicas` defines the nr of replicas
- `spec.selector.matchLabels` PODs with this label will be managed by the ReplicaSet
- `spec.template` This defines what kind of PODs need to be (re-)created

Tasks on ReplicaSets

```
# To view replicasets in the K8S cluster
kubectl get replicasets
kubectl get rs

# To describe the state of a replicaset
kubectl describe rs frontend

# To delete a replicaset
kubectl delete rs frontend
```

`ReplicaSets` are almost never created directly. Most times they get created by the higher order API object `deployment` which will be discussed in the next module. The `deployment` will generate a `ReplicaSet`. When the generated `ReplicaSet` is killed the `Deployment` will create a new one.

`Deployments` are more flexible and enable features like upgrade patterns etc.

Kubernetes Deployments

Kubernetes Deployments

Concept of Deployments

A `Deployment`

- Provides a way to deploy managed `ReplicaSets`
- The generated `ReplicaSet` will deploy a set of identical `PODs`
- Gives a more declarative interface to `RS` and `POD` updates
- The `DeploymentController` manages the `Deployment`

USE CASES FOR DEPLOYMENTS

With `Deployments`

- One can deploy a `ReplicatSet`
- One can update PODs
- One can rollback to older versions of the `Deployment`
- One can pause and resume the `Deployment`
- One can execute various `Deployment` and `upgrade` patterns

To create a `Deployment` adhoc using `kubectl` execute:

```
kubectl run nginx --image nginx:1.7.1
```

To create a `Deployment` using a YAML manifest

```yaml
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: httpd
  replicas: 4
  template:
    metadata:
      labels:
        app: httpd
    spec:
      containers:
      - name: apache
        image: httpd:2.4.39-alpine
        ports:
        - containerPort: 80
```

The YAML manifest of a Deployment has the following important sections and attributes:

- `spec.replicas` defines the nr of replicas
- `spec.selector.matchLabels` PODs with this label will be managed by the Deployment
- `spec.template` This defines what kind of PODs need to be (re-)created

Tasks on Deployments

```
# To view deployments in the K8S cluster
kubectl get deployments
kubectl get deploy

# To describe the state of a deployment
kubectl describe deployment httpd

# To delete a deployment
kubectl delete deployment httpd

# To update a deployment
kubectl set image deployment/nginx-deployment nginx=nginx:1.91 --record

# To display the history of updates
kubectl rollout history deployment.v1.apps/nginx-deployment

# To rollback an update
kubectl rollout undo deployment.v1.apps/nginx-deployment

# List PODs created by this deployment
kubectl get pods -l httpd
```

Kubernetes DaemonSets

Kubernetes DaemonSets

Concept of DaemonSets

Sometimes a POD only needs to run one single instance per node. The DaemonSet ensures this. A `DaemonSet`

- Manages a set of PODs
- Ensures that each `node` gets exactly 1 POD
- When a `node` is added to the cluster this `node` will automatically get it's own instance of the POD
- When a `node` is removed from the cluster, no other node will receive an extra POD.

The following cases are suitable for deploying PODs using `DaemonSets`

- Security, vulnerability or virus scanners
- Logging agents
- Performance collector agents
- Ingress controllers

CREATING A DAEMONSET WITH YAML MANIFESTS

DaemonSets are created with YAML manifests:

```
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      containers:
      - name: fluentd-elasticsearch
        image: k8s.gcr.io/fluentd-elasticsearch:1.20
```

The YAML manifest of a `DaemonSet` has the following important sections and attributes:

- `spec.selector.matchLabels` PODs with this label will be managed by the DaemonSet
- `spec.template` This defines what kind of PODs need to be (re-)created

Tasks on DaemonSets

```
# To view daemonsets in the K8S cluster
kubectl get daemonsets
kubectl get ds

# To describe the state of a daemonset
kubectl describe ds frontend

# To delete a daemonset
kubectl delete ds frontend
```

You may now work on the LABs in chapter 5

- 5.1 Creating DaemonSets
- 5.2 Communicating with PODs managed by DaemonSets
- 5.3 Upgrading PODs in DaemonSets

Kubernetes Jobs

Kubernetes Jobs

Concept of Jobs

PODs can be restarted automatically upon exiting using a `ReplicaSet` or `Deployment`. This ideal for PODs that have to process an (virtually) infinite amount of work. However some PODs have work that is finite and only need to be restarted upon failure and not upon completion. For these type of PODs K8S provides the `Job`. These `Jobs`

- Manage a set of PODs to carryout a finite amount of workload
- Will restart them upon failure
- Will not restart them upon completion

Use cases for `Jobs`

- Batch processing of a finite amount of work at a time
- Work that needs to be done to transform or update data
- Work that setups an environment for other PODs

CREATING A JOB ADHOC

To create a `Job` adhoc using `kubectl` execute:

```
kubectl run busybox --image=busybox --restart=OnFailure
```

To create a `Job` using a YAML manifest

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: example-job
spec:
  template:
    metadata:
      name: example-job
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl"]
        args: ["-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

The YAML manifest of a `Job` has the following important sections and attributes:

- `spec.spec.restartPolicy` Always set to Never for a Job
- `spec.template` This defines what kind of POD needs to be (re-)created

Tasks on Jobs

```
# To view jobs in the K8S cluster
kubectl get jobs

# To describe the state of a job
kubectl describe job example-job

# To delete a job
kubectl delete job httpd
```

Kubernetes CronJobs

Kubernetes CronJobs

Concept of CronJobs

If a Job needs to be executed at a scheduled time, K8S provides so called `CronJobs` for this purpose. `CronJobs`:

- Execute a set of PODs using a predefined schedule
- Use a UNIX/Linux `crontab` like notation
- Jobs that are completed are not restarted
- Jobs that fail get restarted

Use cases for `CronJobs`

- Batch processing of a finite amount of work periodically
- End of day processing
- Periodic scanning

To create a `CronJob` adhoc using `kubectl` execute:

```
kubectl run busybox --image=busybox --schedule="4 10 * * *"  --restart=OnFailure
```

To create a `CronJob` using a YAML manifest

```yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: my-crontab
spec:
  schedule: "*/5 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: pi
            image: perl
            command: ["perl"]
            args: ["-Mbignum=bpi", "-wle", "print bpi(2000)"]
          restartPolicy: OnFailure
```

The YAML manifest of a `CronJob` has the following important sections and attributes:

- `spec.schedule` Specifies when the Job should be scheduled
- `spec.jobTemplate` Describes the job to be executed

Tasks on Jobs

```
# To view cronjobs in the K8S cluster
kubectl get cronjobs
kubectl get cj

# To describe the state of a cronjob
kubectl describe cronjob my-cronjob

# To delete a cronjob
kubectl delete cronjob my-cronjob
```

Kubernetes Persistent Storage

- Like Docker containers, pods are designed to be volatile
- This means they cannot keep state themselves
- If an application needs to keep state, you'll need (persistent) storage

VOLUME TYPES

- Local storage
- iSCSI
- NFS
- Cloud storage
- emptyDir
- hostMount
- configMaps
- secrets
- etc...

EMPTYDIR VOLUMES

- Persistence: only for the lifetime of the POD
- Can be shared with other containers in the POD

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: simple-volume-pod
spec:
  volumes:
    - name: simple-vol
      emptyDir: {}
  containers:
    - name: my-container
      volumeMounts:
        - name: simple-vol
          mountPath: /var/simple
      image: alpine
      command: ["/bin/sh"]
      args: ["-c", "while true; do date >> /var/simple/file.txt; sleep 5; done"]
```

- Persistent Volume (PV) resources are used to manage durable storage in a cluster
- Unlike volumes the lifecycle is mmanaged by Kubernetes
- A PV can already exist or dynamically provisioned via plugins
- The PV must be bind to the cluster using a Persistent Volume Claim (PVC)
- The PVC dictates the kind and size of the storage

- Create one or more PVs (they map storage)
- Create a Persistent Volume Claim
- A POD is created claiming storage using a PVC
- The scheduler selects which PV is suitable for the POD
- Storage is bound to the POD

PERSISTENT VOLUME MANIFEST (1)

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-nfs-002
spec:
  capacity:
    storage: 20Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
```

PERSISTENT VOLUME MANIFEST (2)

```
mountOptions:
  - hard
  - nfsvers=4.1
nfs:
  path: /k8s/vol002
  server: 10.8.62.222
```

PERSISTENT VOLUME CLAIM MANIFEST

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-nfs-001
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: "slow"
  resources:
    requests:
      storage: 1Mi
```

POD EXCERCISING A PVC

```yaml
spec:
  containers:
  ...
  ports:
  volumeMounts:
  # name must match the volume name below
    - name: nfs-html
      mounthPath: "/usr/share/nginx/html"
volumes:
    - name: nfs-html
      persistentVolumeClaim:
        claimName: pvc-nfs-001
```

Kubernetes ConfigMaps

- Like Docker configs, ConfigMaps are used to store configuration data
- They are used to separate the configuration from the container image
- ConfigMaps are not encrypted

ConfigMaps can materialize the data contained as:

- Environment Variables in the container/POD
- Files in a filesystem in the container/POD

CREATING CONFIGMAPS

```
# Adhoc from a file
kubectl create configmap mysqlcfg1 --from-file=/etc/mysql.conf

# Adhoc from literal
kubectl create configmap myconfig --from-literal=color=red --from-literal=mascot=astro
```

CONFIGMAPS YAML MANIFESTS

```yaml
apiVersion: v1
data:
  color: red
  mascot: astro
kind: ConfigMap
metadata:
  name: myconfig
```

USING CONFIGMAPS IN PODS

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: color-container
spec:
  containers:
    - name: color-container
      image: pamvdam/nginxc:v1.1
      env:
        - name: COLOR
          valueFrom:
            configMapKeyRef:
              name: myconfig
              key: color
  restartPolicy: Never
```

USING CONFIGMAPS IN PODS - VOLUMEMOUNT

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: color-container
spec:
  containers:
    - name: color-container
      image: pamvdam/nginxc:v1.1
      volumemounts:
        - name: myconfigvol
          mountPath: /myconfig
  volumes:
    - name: myconfigvol
      configMap:
        name: myconfig
```

TASKS ON CONFIGMAPS

```
# Delete a configmap

kubectl delete configmap myconfig

# Describe a configmap

kubectl describe configmap myconfig

# Update PODs in a deployment where ConfigMap has been updated
kubectl rollout restart deploy
```

You may now start with LAB 7 - ConfigMaps

Kubernetes Secrets

SECRETS

- Like Docker secrets, Secrets are used to store sensitive configuration data
- They are used to separate this type of configuration from the container image
- Use secrets to store privkeys, passwords, certs etc
- Secrets are encoded not encrypted
- Use encryption at rest to get them encrypted

Secrets can materialize the data contained as:

» Environment Variables in the container/POD
» Files in a filesystem in the container/POD

CREATING SECRETS

```
# Adhoc from a file
kubectl create secret generic mysecret1 --from-file=/etc/mysql.passwd

# Adhoc from literal
kubectl create secret generic mysecret2 --from-literal=color=red --from-literal=mascot=astro
```

SECRETS YAML MANIFESTS

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
data:
  username: bXktYXBw
  password: Mzk1MjgkdmRnN0pi
```

USING SECRETS IN PODS

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: env-single-secret
spec:
  containers:
  - name: envars-test-container
    image: nginx
    env:
    - name: SECRET_PASSWORD
      valueFrom:
        secretKeyRef:
          name: test-secret
          key: password
```

USING SECRETS IN PODS - VOLUMEMOUNT

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-pod
spec:
  containers:
    - name: test-container
      image: nginx
      volumeMounts:
        - name: secret-volume
          mountPath: /etc/secret-volume
  volumes:
    - name: secret-volume
      secret:
        secretName: test-secret
```

TASKS ON SECRETS

```
# Delete a secret

kubectl delete secret mysecret

# Describe a secret

kubectl describe secret mysecret

# Update PODs in a deployment where Secret has been updated
kubectl rollout restart deploy
```

Kubernetes Ingresses

Kubernetes Ingresses

Concept of Ingresses

Ingresses

- Manage routing of a network traffic to services
- Limit costs and effort spent on LoadBalancers and Firewalls
- Loadbalances traffics
- Offers encrypted communication (TLS)

The following cases are suitable for deploying `Ingresses`

- Name-based virtual hosting (CNAME records)
- Path-based routing
- Any combination of the above

Ingresses are composed of the following objects

- ClusterRole / RBAC for installing the Ingress
- Ingress controller as a deployment or daemonset
- Ingress rules

INGRESS CONTROLLER OPTIONS IN K8S

- Nginx
- Kong
- Traefik
- HAProxy
- Ambassador

Tasks on Ingresses

```
# Install Traefik as a daemon-set

kubectl create -f traefik-ds.yaml

# Install the ingress rules

kubectl create -f my-ingress.yaml

# Test the ingress

curl http://st99node01.itgildelab.net/red
```

INGRESS RULES EXAMPLE - PATH BASED

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: path-rule-ingress
  annotations:
    traefik.frontend.rule.type: PathPrefixStrip
spec:
  backend:
    serviceName: nginxc-black
    servicePort: 80
  rules:
  - http:
      paths:
      - backend:
          serviceName: nginxc-blue
          servicePort: 80
        path: /blue
  - http:
      paths:
      - backend:
          serviceName: nginxc-red
          servicePort: 80
        path: /red
```

INGRESS RULES EXAMPLE - HOST BASED

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: host-rule-ingress
  annotations:
    traefik.frontend.rule.type: PathPrefixStrip
spec:
  backend:
    serviceName: nginxc-black
    servicePort: 80
  rules:
  - host: blue.itgildelab.net
    http:
      paths:
      - backend:
          serviceName: nginxc-blue
          servicePort: 80
  - host: purple.itgildelab.net
    http:
      paths:
      - backend:
          serviceName: nginxc-purple
          servicePort: 80
```

Kubernetes Scheduling

Normally there's no need to steer the K8S scheduler in placing PODs on the nodes. The Kubernetes Scheduler knows best where to schedule a POD and will take the PODs needs into account. With three primitives we are able to steer the scheduling:

- nodeSelector (deprecated)
- Node Affinity
- Inter-Pod Affinity

To make use of the `nodeSelector` we will need to label the nodes. This can be done with the `kubectl label nodes` command.

```
kubectl label node st00node02 NICtype=40Gb
node/st00node02 labeled

kubectl get nodes -l NICtype=40Gb
NAME         STATUS   ROLES    AGE   VERSION
st00node02   Ready    <none>   46h   v1.14.1

kubectl get nodes --show-labels
```

The `nodeSelector` will schedule a POD onto a node whose labels match that of the `nodeSelector` The `nodeSelector` is part of the POD spec.

```
...
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    NICtype: 40GiB
```

Node Affinity and Node Anti Affinity supply more powerful methods compared to the use of `nodeSelector`

- The Affinity language is more expressive and powerful
- Enables `soft` and `hard` scheduling rules
- Support PODS co-location with inter-POD affinity.

USE OF NODE AFFINITY

- `requiredDuringSchedulingIgnoredDuringExecution`: supplies a `hard` rule.

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: NICtype
            operator: In
            values:
            - 25Gb
            - 40Gb
```

» `preferredDuringSchedulingIgnoredDuringExecution:` supplies a `soft` rule.

```
spec:
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: non-batch
            operator: In
            values:
            - true
```

POD AFFINITY

POD Affinity can be usefull to

- Co-locate PODs in the same availability zone
- Co-locate PODs that have stron interdependency on one node

HARD POD AFFINITY

```
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security-zone
            operator: In
            values:
            - high
        topologyKey: failure-domain.beta.kubernetes.io/zone
```

SOFT POD AFFINITY

```yaml
spec:
  podAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 100
      podAffinityTerm:
        labelSelector:
          matchExpressions:
          - key: security-zone
            operator: In
            values:
            - S2
        topologyKey: kubernetes.io/hostname
```

POD Anti Affinity use cases:

- Spread PODs of a service over nodes or availability zones
- Grant a POD exclusive access to a certain node
- Isolate PODs that could interfere with eachother on one node

POD ANTI AFFINITY

```yaml
spec:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 100
      podAffinityTerm:
        labelSelector:
          matchExpressions:
          - key: security-zone
            operator: In
            values:
            - S3
        topologyKey: kubernetes.io/hostname
```

Kubernetes Kubeconfig

- Kubectl is used to administer the kubernetes cluster
- Kubectl is able to administer multipe kubernetes clusters
- Ways to instruct kubectl which cluster to address:
  - Commandline options
  - Seperate kubeconfig files
  - Merged kubeconfig file

Kubectl uses the following precedence for the kubeconfigs

- kubectl –kubeconfig flag
- KUBECONFIG environment variable
- $HOME/.kube/config file

KUBECTL –KUBECONFIG FLAG

```
kubectl get pods --kubeconfig=kubtst
```

KUBECTL USING KUBECONFIG ENV VAR

```
KUBECONFIG=kubtst kubectl get pods

export KUBECONFIG=kubtst:kubprd

kubectl get pods --context=cluster-1
kubectl get pods --context=cluster-2
```

MERGING MULTIPLE KUBECONFIG FILES

```
KUBECONFIG=kubdev:kubtst:kubacc:kubprd \
  kubectl config view --merge \
  --flatten > allkubeconfig
```

SETTING CLUSTER CONTEXT FOR KUBECTL

```
kubectl use-context kubprd
kubectl get nodes

kubectl use-context kubtst
kubectl get pods
```

TASKS ON KUBECONFIGS

```
# Get cluster managed by kubeconfig file

kubectl config get-clusters

# Get context managed by kubeconfig file

kubectl config get-contexts

# Show current kubeconfig file contents

kubectl config view
```

JAVA workload on K8S

Running JAVA in a legacy (VM or Physical) OS results in having the JVM configure itself for the amount of physical memory available on the system. Typically 25% of the memory is configured for HEAP.

Inside a container JAVA also reads the total memory. But even when there are memory limits set like with `--limits=\'memory=600Mi\'` the JVM just ignores this and configures the HEAP config using the total physical memory on the host/node.

Result: JVM thinks it has more memory available than it's entitled too and once the entitlement has been fully used, the POD/container will be killed.

How to prevent this? The solution depends on the JAVA version used.

To prevent a JAVA 7 JVM in a POD from getting killed by (auto-)misconfiguration:

» Set the maximum usable heap memory with `-Xmx` etc.

Dockerfile JAVA 7

```
FROM openjdk:7
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
RUN javac ShowMeYourHeap.java
CMD ["java","-Xmx300","ShowMeYourHeap"]
```

SOLUTION FOR JAVA VERSIONS 8 AND 9

To prevent a JAVA 8 or 9 JVM in a POD from getting killed by (auto-)misconfiguration:

- Use the experimental JVM option: `-XX:+UnlockExperimentalVMOptions`
- and `-XX:+UseCGroupMemoryLimitForHeap`

Dockerfile JAVA 8 or 9

```
FROM openjdk:7
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
RUN javac ShowMeYourHeap.java
CMD ["java","-XX:+UnlockExperimentalVMOptions","-XX:+UseCGroupMe
```

JAVA JVM version 10 and above directly recognize when they are running in a POD or container. There's no need to use the experimental options. It will work out-of-the-box and take the `--limits` constraints into account when configuring the JVM

Dockerfile JAVA 10+

```
FROM openjdk:7
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
RUN javac ShowMeYourHeap.java
CMD ["java","ShowMeYourHeap"]
```

Troubleshooting

A good start begins with best practices

- Single proces per container
- Document the design thoroughly
- Do not intermingle different techniques
- Use a log collector and aggregator

- Check docker container logs `docker logs <container>`
- Take a peek inside the container using `docker exec -it` or `kubectl exec -it`
- Get process stats info with `docker top <container>`
- View container details with `docker inspect <container>`
- View image layers with the `docker image history` command

- Check exposed port configuration
- Check local and external firewalls
- Check DNS resolving in /etc/docker/daemon.conf
- Check secure/in-secure registries in /etc/docker/daemon.conf

- Check where disks and config(maps) are mounted
- Check permission in- and outside the container