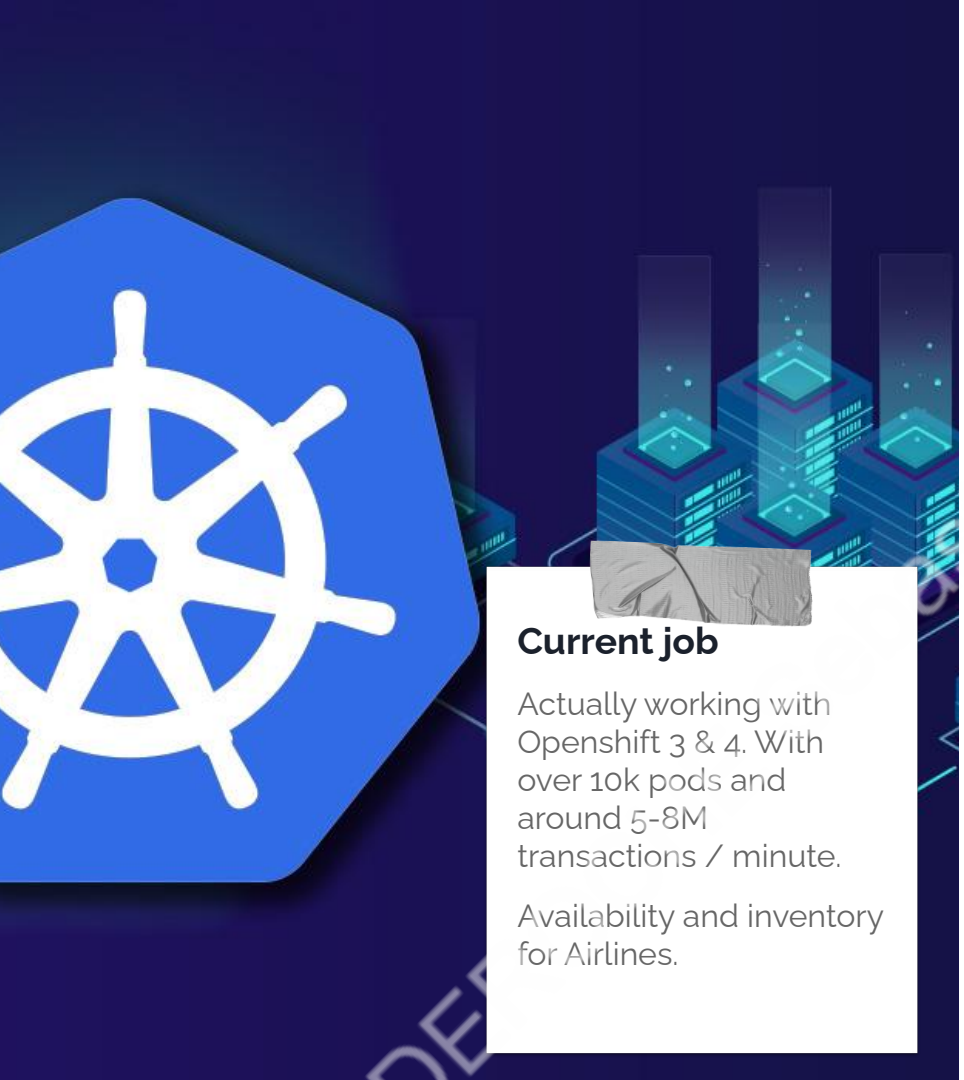
Advanced Kubernetes

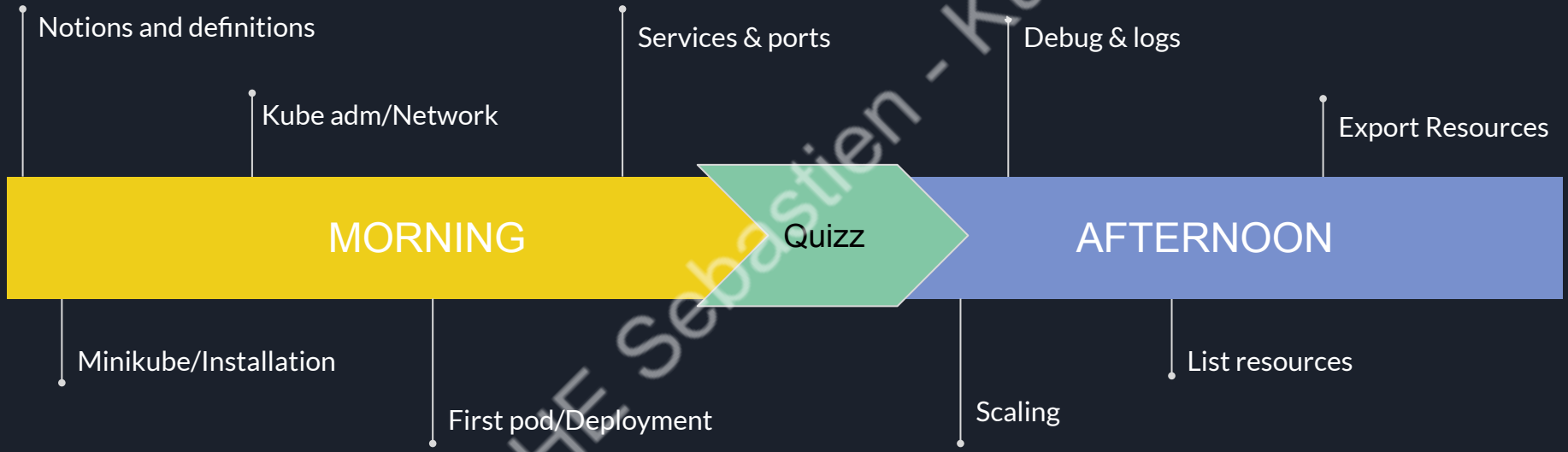
DEROCHE Sebastien



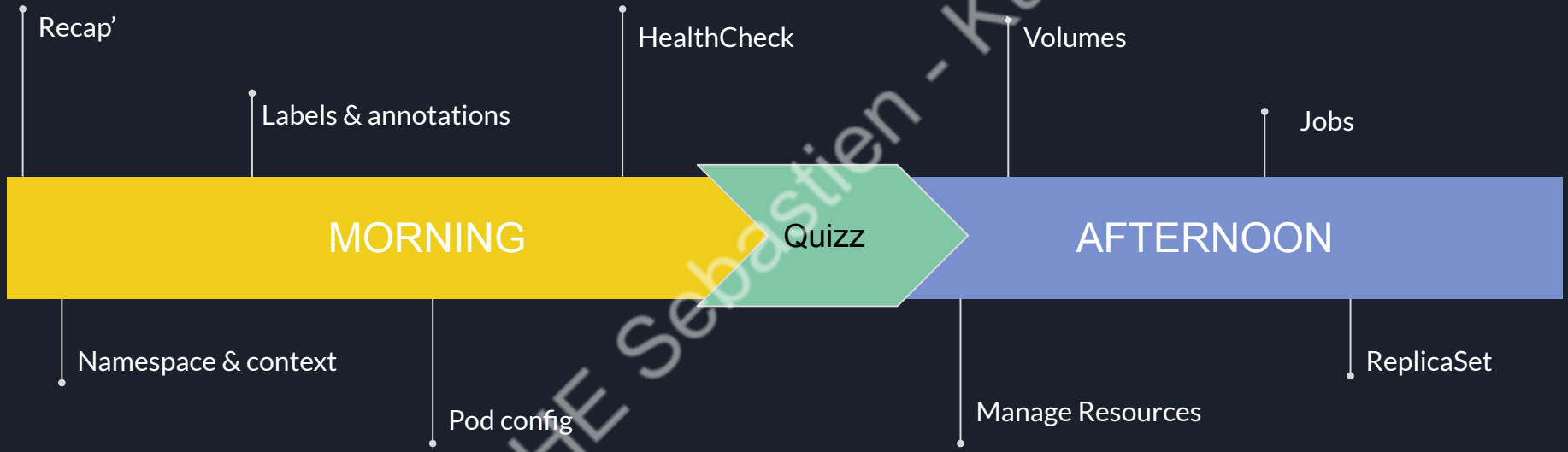
Who am I?

- DEROCHE Sebastien (SeebOmega)
- Service Reliability Engineer (Amadeus)
- Active / Competitive Coder
- Self taught

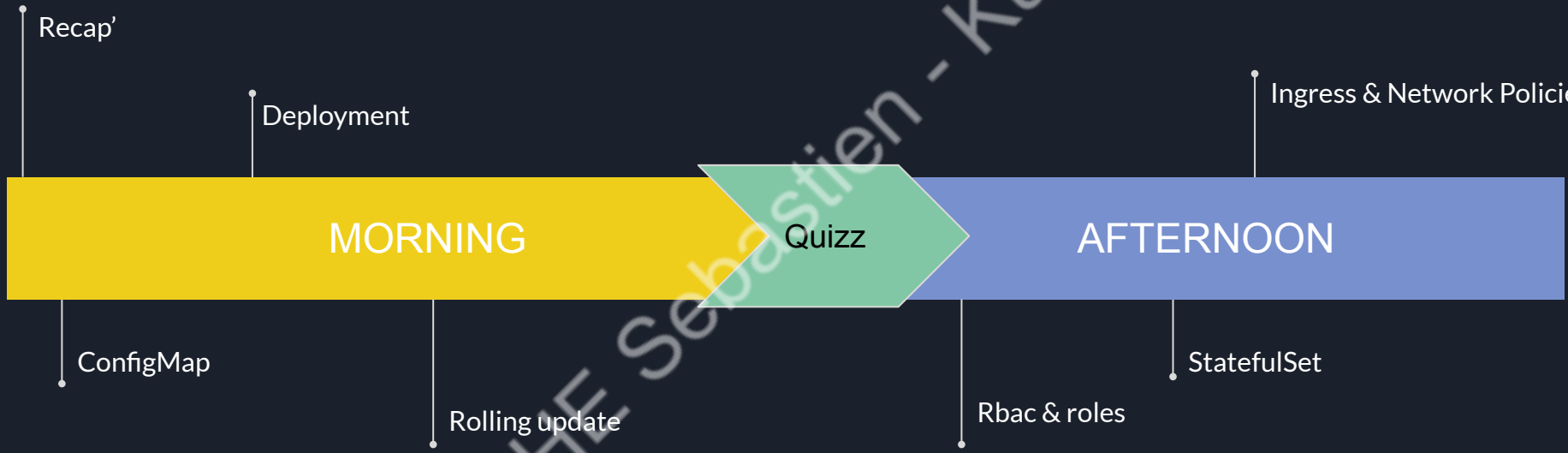
DAY 1:



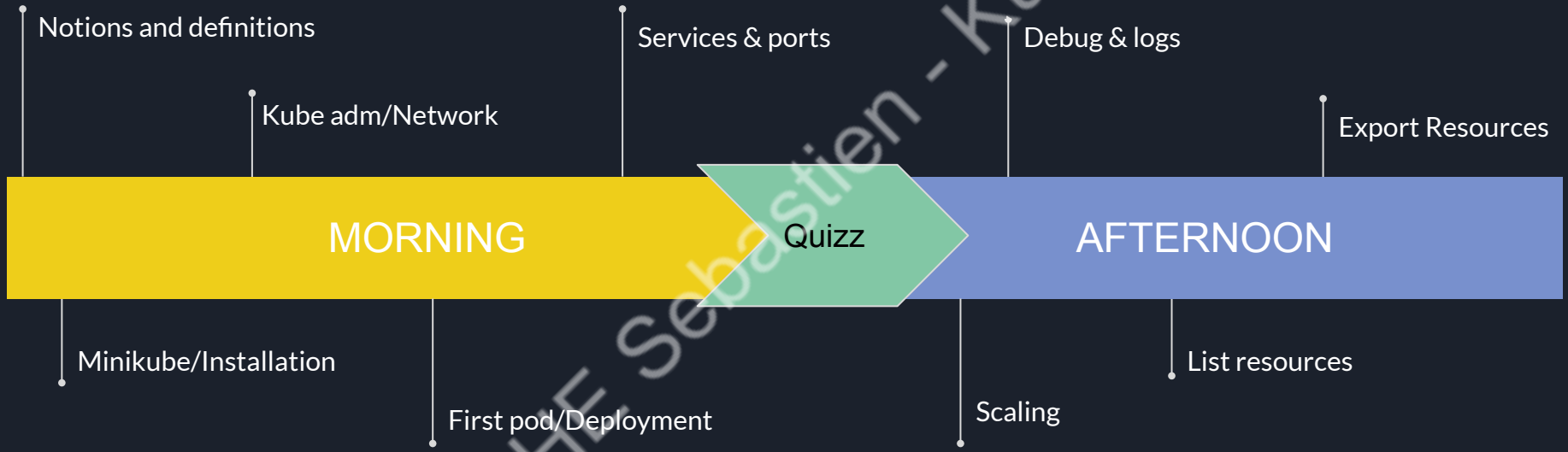
DAY 2:



DAY 3:



DAY 1:



“It groups containers that make up an application into logical units for easy management and discovery. Kubernetes builds upon 15 years of experience of running production workloads at Google, combined with best-of-breed ideas and practices from the community.”

Notions & Definitions

Notions & Definitions

- Node: Is a server physical or virtual
  - Can be Master or Simple Execution
- Pods: Is k8s basic notion/instance
  - One or More Container
- Service: Abstract pod network layer
  - No IP usage

Notions & Definitions

- Volume: Can be persistent or not
  - Exchange data between pods or with Host
- Deployment: Declaration for deployment management
  - Creation / Deletion / Scaling
- Namespace: Cluster in the cluster(virtual)
  - Uniformity and partitioning

Minikube Installation

Installation Minikube

First, we are going to install VirtualBox as Hypervisor

https://www.virtualbox.org/wiki/Downloads

Then we are going to get Minikube

https://kubernetes.io/docs/tasks/tools/install-minikube/

Minikube Linux

```
curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64 \
  && chmod +x minikube
```

Minikube Windows

```
Choco install minikube
```

OR

https://github.com/kubernetes/minikube/releases/latest/download/minikube-installer.exe

Minikube MacOS

```
Brew install minikube
```

OR

```
curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64 \
  && chmod +x minikube
```

OR

https://github.com/kubernetes/minikube/releases/latest/download/minikube-installer.exe

Minikube commandes

First run of Minikube: `./minikube start`

Check status: `./minikube status`

Stop it: `./minikube stop`

Trash it: `./minikube delete`

Kubectl Installation

Kubectl Linux

```
curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s
https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl"

chmod +x ./kubectl

sudo mv ./kubectl /usr/local/bin/kubectl
```

OR

```
sudo apt-get update && sudo apt-get install -y apt-transport-https gnupg2
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee -a /etc/apt/sources.list.d/kubernetes.list
sudo apt-get update
sudo apt-get install -y kubectl
```

Kubectl Windows

```
choco install kubernetes-cli
cd ~
mkdir .kube
cd .kube
New-Item config -type file
```

OR

https://storage.googleapis.com/kubernetes-release/release/v1.18.0/bin/windows/amd64/kubectl.exe

Kubectl MacOS

```
Brew install kubectl
```

OR

```
curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/darwin/amd64/kubectl"
```

K8s Installation

K8s Installation

Requirements:

- Some VMs / Servers
- No swap
- Ports Master: 6443 2379-2380 10250-10252
- Ports Nodes: 10250 3000-32767
- Docker installed *(https://docs.docker.com/engine/install/)*

K8s Installation

- *swapoff -a*
- *vim /etc/fstab*
- *apt-get update && apt-get install -y apt-transport-https curl*
- *curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -*
- *sudo add-apt-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"*
- *sudo apt-get update && sudo apt-get install -y kubelet kubeadm kubectl kubernetes-cni*
- *systemctl enable kubelet*

K8s Setup

- Initialisation:

```
kubeadm init --apiserver-advertise-address=192.168.0.40 \
--node-name $HOSTNAME --pod-network-cidr=10.244.0.0/16
```

- Network (Calico):

https://kubernetes.io/docs/concepts/cluster-administration/addons/

```
kubectl apply -f https://docs.projectcalico.org/v3.8/manifests/calico.yaml
sysctl net.bridge.bridge-nf-call-iptables=1
```

- Join:

```
kubeadm join [output of kubeadm init]
```

K8s Extended

- Auto-complete:

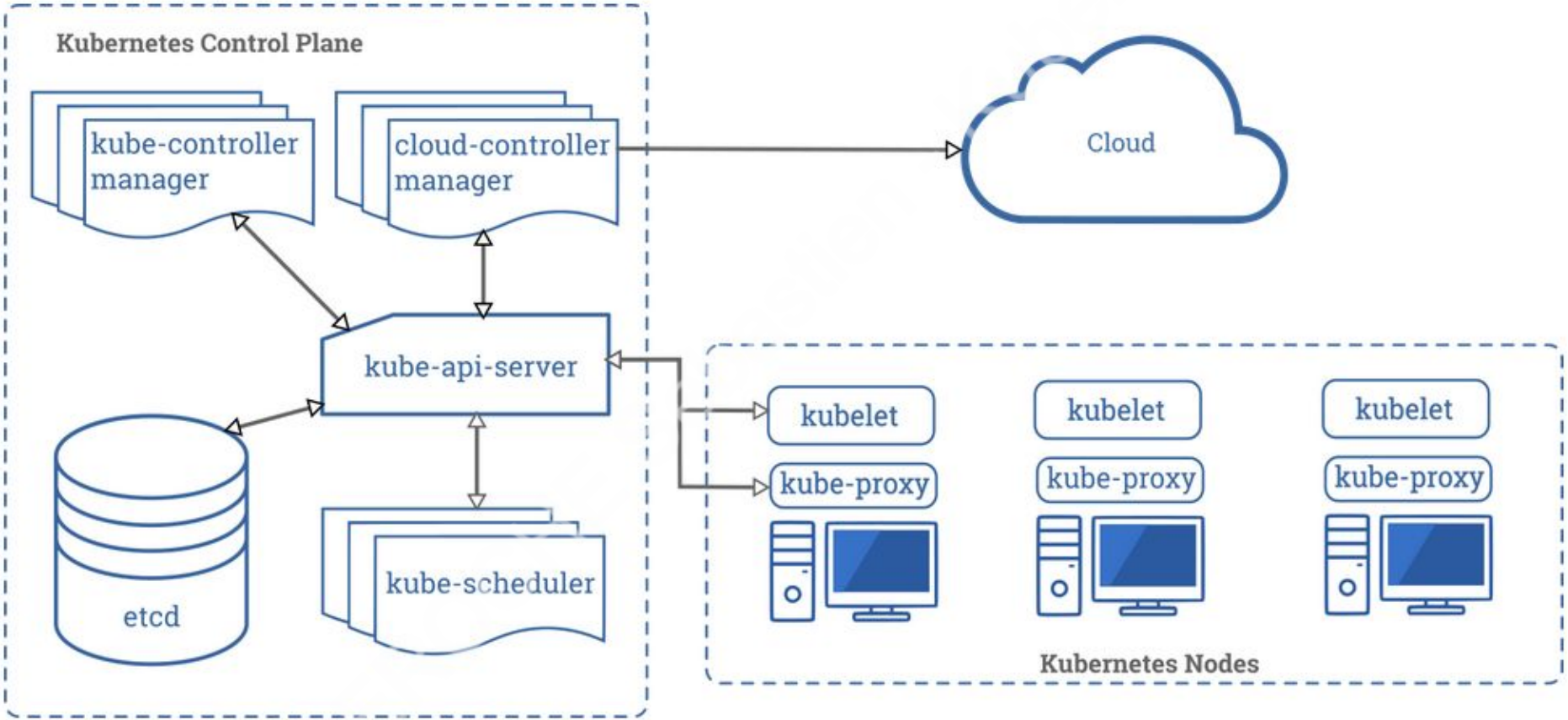
```
apt-get install bash-completion
echo "source <(kubectl completion bash)" >> ~/.bashrc
```

- Remote access:

```
mkdir ~/.kube
ssh user@master "sudo cat /etc/kubernetes/admin.conf" >.kube/config
```

OverView

K8s Architecture



K8s control plane components

Those components take the globals decisions of the cluster.

They also detect and respond to cluster Events.

They can be run from Any machine, but usually we run them on one that will not have any user container. (MASTER)

In high availability mode we have multiple masters. It’s the price to pay to have one or more nodes with nothing else that K8s components on it.

Components:

List of the components of the Control plane:

- Kube-apiserver: Expose the k8s API
- Etcd: Highly available Key-Value Store to keep cluster data
- Kube-scheduler: Watch created pods and assign node to them following rules (resources, labels, policy, affinity...)
- Kube-controller-manager: Runs all the controller processes
    - Node: Notification on Node is Down
    - Replication: Maintain desired replicas
    - Endpoints: populates Endpoint (Services and Pods)
    - ServiceAccounts & Tokens: Create default SA for new namespaces

Cloud controller manager:

This component is not present on on-premise cluster. This will allow you to link your cluster to the CloudProvider's API. This also separate the cluster resource controller from the Cloud resource controller.

Those controllers have dependency to it:

- Node: Check if node Is deleted by CloudProvider when unreachable
- Route: Setting up route in the cloud underlying infrastructure
- Service: Creating, deleting & updating cloud LB.

Node components:

List of components present on Nodes:

- Kubelet: It's an agent that make sure that containers runs in a pod and are healthy.
- Kube-proxy: Maintain the network rules on nodes, and allow communications from inside or outside the cluster.
- ContainerRuntime: Software that runs the containers (Docker, CRI-O, Containerd ...)

Some more Addons are available you can found them here:
https://kubernetes.io/docs/concepts/overview/components/#addons

Deployment

First deployment

- Deployment:
  - Logical representation of pods

- Service:
  - Entrypoint wrapping IP / port

Let’s create a deployment for nginx with:

```
Kubectl create deployment nginx --image nginx
```

```
root@kubmaster:~# kubectl describe pod nginx-f89759699-b9bkk
Name:         nginx-f89759699-b9bkk
Namespace:    default
Priority:     0
Node:         kubnode/10.0.2.15
Start Time:   Thu, 06 Aug 2020 20:54:09 +0000
Labels:       app=nginx
              pod-template-hash=f89759699
Annotations:  cni.projectcalico.org/podIP: 192.168.153.197/32
Status:       Running
IP:           192.168.153.197
IPs:
  IP:           192.168.153.197
Controlled By:  ReplicaSet/nginx-f89759699
Containers:
  nginx:
    Container ID:   docker://321fe07c8db24700f64aa268b73e64410f4fbd6a10ee86ce90d70f4c1c998507
    Image:          nginx
    Image ID:       docker-pullable://nginx@sha256:36b74457bccb56fbf8b05f79c85569501b721d4db813b684391d63e02287c0b2
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Thu, 06 Aug 2020 20:54:19 +0000
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-w8df7 (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-w8df7:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-w8df7
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason     Age          From               Message
  ----    ------     ----         ----               -------
  Normal  Scheduled  <unknown>    default-scheduler  Successfully assigned default/nginx-f89759699-b9bkk to kubnode
  Normal  Pulling    44m          kubelet, kubnode   Pulling image "nginx"
  Normal  Pulled     44m          kubelet, kubnode   Successfully pulled image "nginx"
  Normal  Created    44m          kubelet, kubnode   Created container nginx
  Normal  Started    44m          kubelet, kubnode   Started container nginx
```

- We have a running instance of Nginx but no way to access it.

- So let's expose our POD!

Services

First Service

Let’s create a service for our Nginx pod:

```
Kubectl create service nodeport nginx --tcp=8888:80
```

Note that we can change the type of service depending our needs:

- NodePort = Expose our pod through a port <NodeIP>:<NodePort>
- LoadBalancer = Expose with a cloud provider LB
- ClusterIP = Expose internally to the cluster
- ExternalName = Expose through a CNAME record

We can also use `kubectl expose deployment` but I strongly recommend to use the K8s service notion.

NodePort

When this type ...000-32767).
Each node will ...

If you choose y... ...ision and use
one who is insi...

Using a NodeP... ...load
balancing solu... ...fully
supported by K... ...e nodes' IPs
directly.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
    - name: http
      protocol: TCP
      port: 8888
      targetPort: 80
      nodePort: 30007 #OPTIONNAL
    - name: https
      protocol: TCP
      port: 9999
      targetPort: 443
```

LoadBalancer exposition

On cloud providers which support external load
provisions a load balancer for your Service.

See provider specificity for using LoadBalancerIP
creation.

See https://kubernetes.io/docs/concepts/services-networking/
for specific configurations and features from the

Connexions draining, Proxy, Partial TLS support ..

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  annotations:
        cloud.google.com/
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 8888
      targetPort: 80
  clusterIP: 10.0.171.239
  type: LoadBalancer
status:
  loadBalancer:
```

ClusterI

Default moc
or for interna

development,

So you can c
expose your

ration who will

You'll then n
manually ex

ou want to
at this point).

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 8888
      targetPort: 80
  clusterIP: 10.0.171.239
  type: ClusterIP
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  namespace: prod
spec:
  type: ExternalName
  externalName: nginx-service.company.org
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - name: http
      protocol: TCP
      port: 8888
      targetPort: 80
  externalIPs:
    - 80.11.12.10
#this pod will be accessible via http://80.11.12.10:8888
```

Network Models

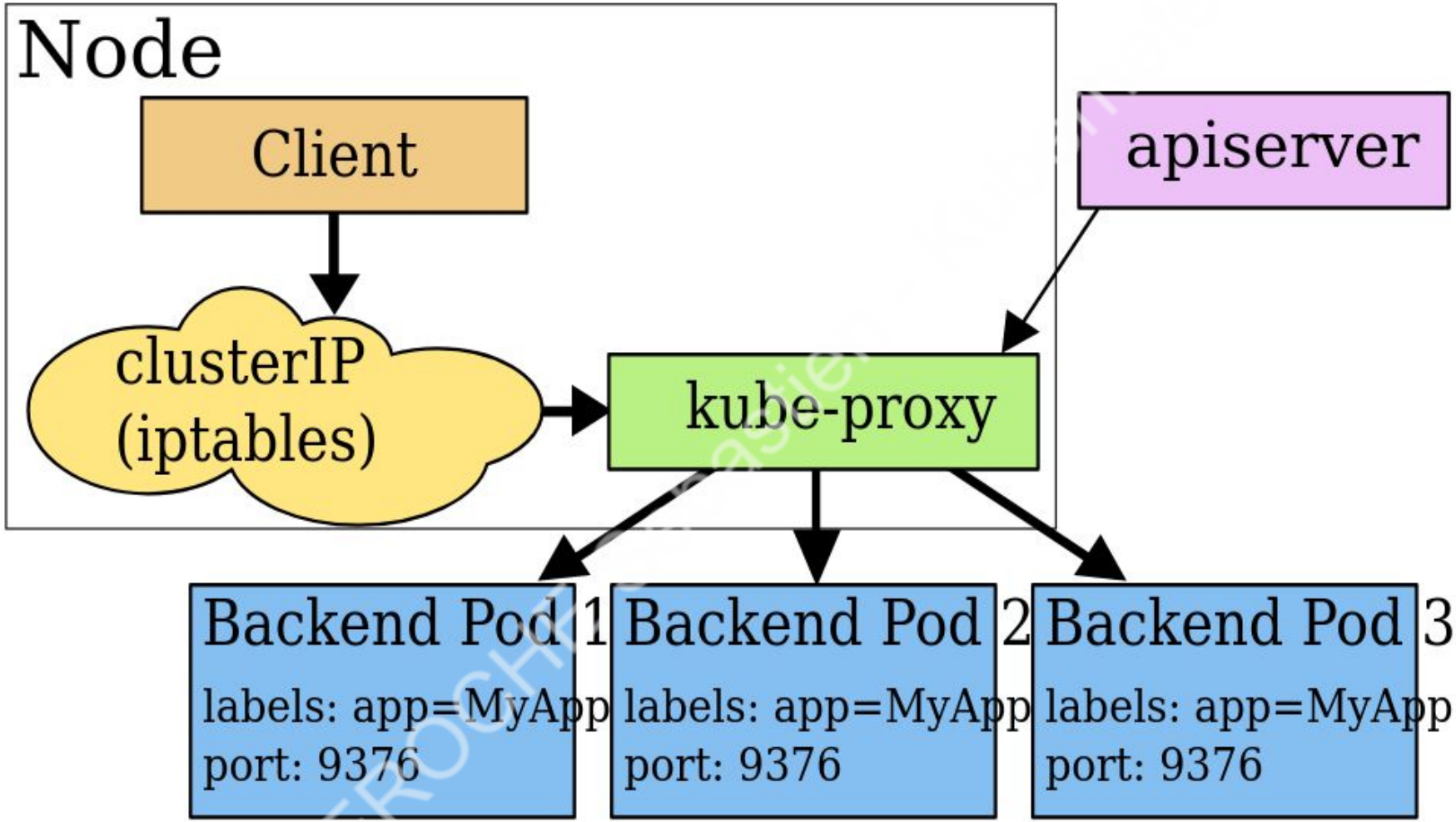
Virtual IPs and Proxy Service

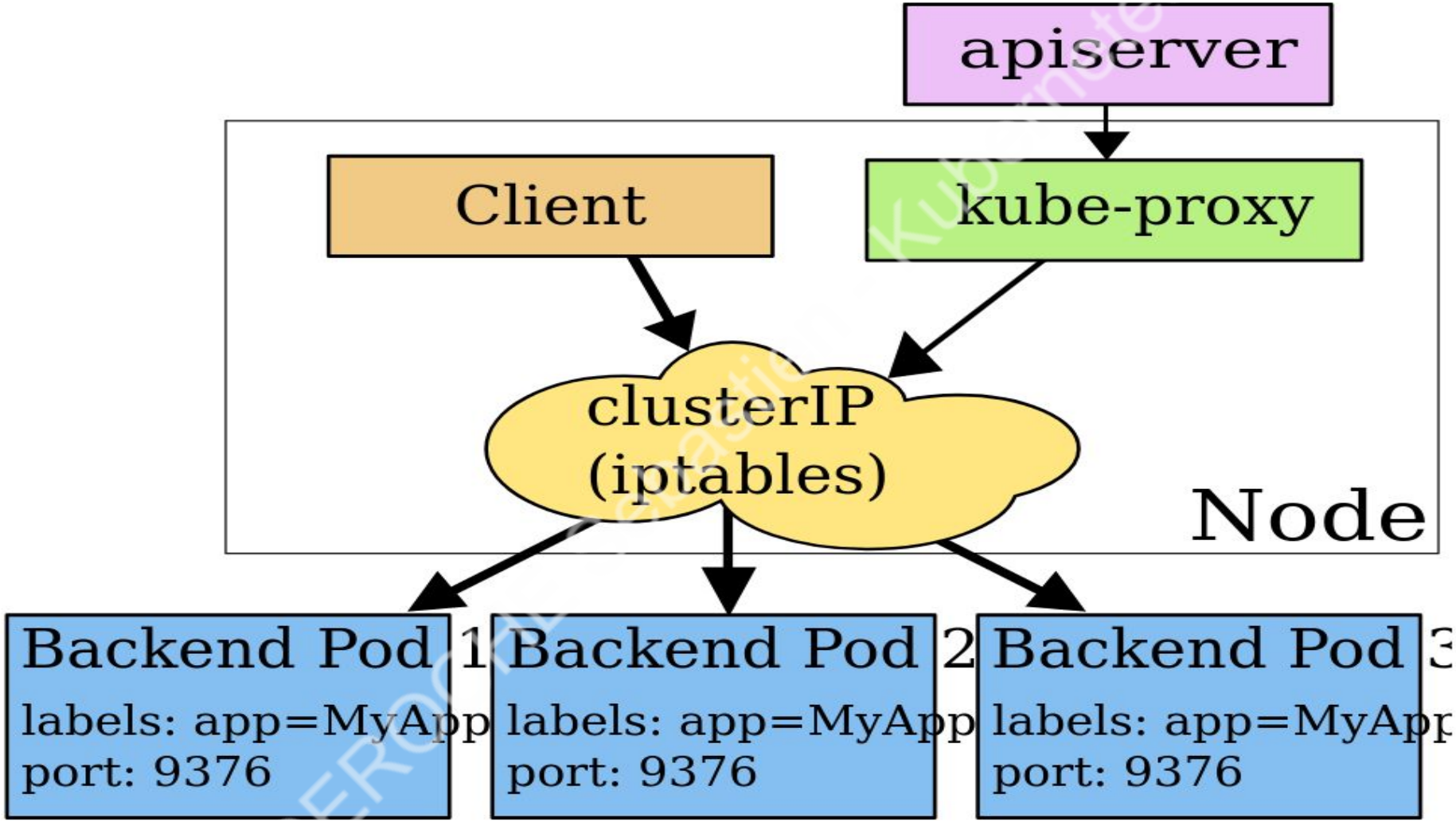
Each node of a K8s cluster runs a `kube-proxy` which is responsible of implementing a virtual IP for the services that are not External Name type.

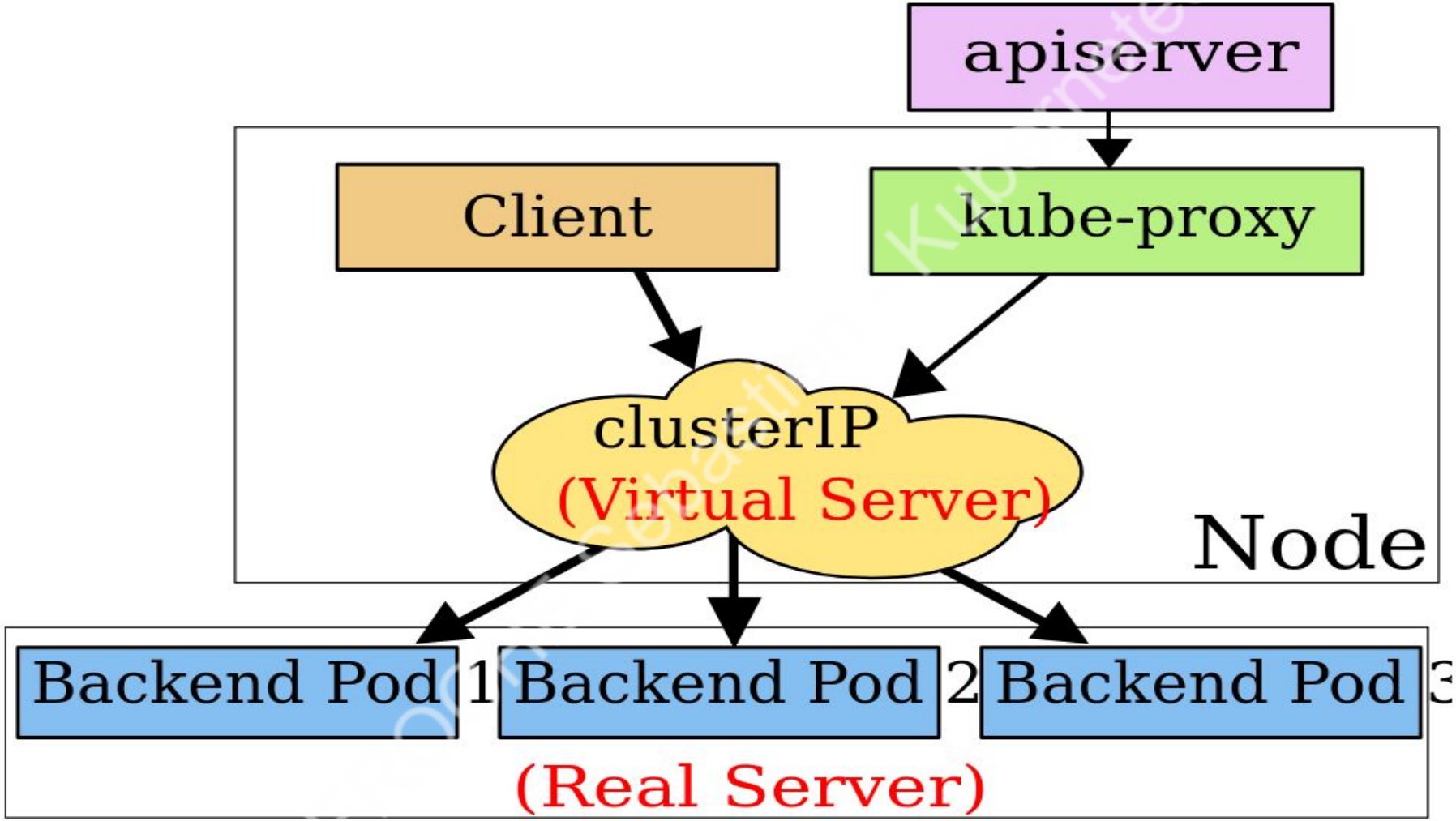
Default mode is IPtable mode, tends to be replaced by IPVS for now according to the documentation it's still IPtable.

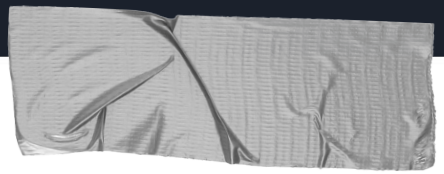
Fallback if something is not supported:

IPVS > IPTable > User Space


Node
Client
apiserver
clusterIP
(iptables)
kube-proxy
Backend Pod 1
labels: app=MyApp
port: 9376
Backend Pod 2
labels: app=MyApp
port: 9376
Backend Pod 3
labels: app=MyApp
port: 9376


apiserver
Client
kube-proxy
clusterIP
(iptables)
Node
Backend Pod 1
labels: app=MyApp
port: 9376
Backend Pod 2
labels: app=MyApp
port: 9376
Backend Pod 3
labels: app=MyApp
port: 9376


apiserver
Client
kube-proxy
clusterIP
(Virtual Server)
Node
Backend Pod 1
Backend Pod 2
Backend Pod 3
(Real Server)



QUIZZ TIME!

➔ Is it possible to execute locally kubectl commands that targets a remote cluster?

Yes, by having a .kubeconfig file

➔ What are the 4 types of services?

NodePort, ClusterIP, LoadBalancer & ExternalName

➔ What is the default kube-proxy mode?

IPtable

Scaling

First scaling

Creation of an instance in order to split the charge. So let's continue with our Nginx.

First we are going to make it display the instance number:

```
kubectl exec -ti nginx-f89759699-plt5j -- /bin/bash
```

And now let's horizontal scale it:

```
kubectl scale deployment nginx --replicas=2
```

And make it display his instance number as well.

Autoscaling

Auto scaling

We can now let K8s auto scale our pod to handle the charge

```
kubectl autoscale deployment nginx --min 2 --max 5
```

The controller fetches the metrics from the resource metrics API for each pod. Depending on the type of target specified the controller will calculate a ratio to produce the required number of replicas.

You can create custom pod metrics, and define a target value.
See: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/instrumentation/custom-metrics-api.md

desiredReplicas = ceil[currentReplicas * (currentMetricValue / desiredMetricValue)]

Scaling Policy

You can access the scaling policy with:

```
kubectl [get/describe] hpa
```

You can attribute behaviours for scaling(UP/DOWN) for example avoid brutal scale down, a Yo-Yo effect etc ...

You can also freeze your cluster (Block scaling) in order to perform Rolling update or maintenance for example.

Configurable from API under the `behavior` section

Resource: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/
Create small intensive PHP script and autoscale from custom + Base metrics

Debug and Logs

Getting more information

In order to gather more information we can use `-o wide`:

```
kubectl get [node/pods] -o wide
```

We can also use the `describe` keyword to have deeper information on the resource.

Check the components/daemonsets status with:

```
kubectl get componentstatuses
Kubectl get daemonsets -n kube-system
```

Complete list -> `kubectl get all`

Accessing the logs

Get the logs of a pod:

```
kubectl logs -f nginx-xxxx
```

Get the events linked to a pod:

```
kubectl describe pod nginx-xxxx
```

Accessing a pod:

```
kubectl exec [-ti] nginx-xxxx <cmd>
```

List resources

Important resources

Get all the resources from the API:

```
kubectl api-resources
```

Get all the resources version from the API:

```
kubectl api-versions
```

Manual for a resource:

```
kubectl explain service
```

Export /Import resources

YAML for the win

Command line is cool but when the cluster becomes bigger and automation kicks in you want to have descriptive YAML files.

- Versionable (GIT)
- Easier to maintain when config become complex
- To go further See **HELM**

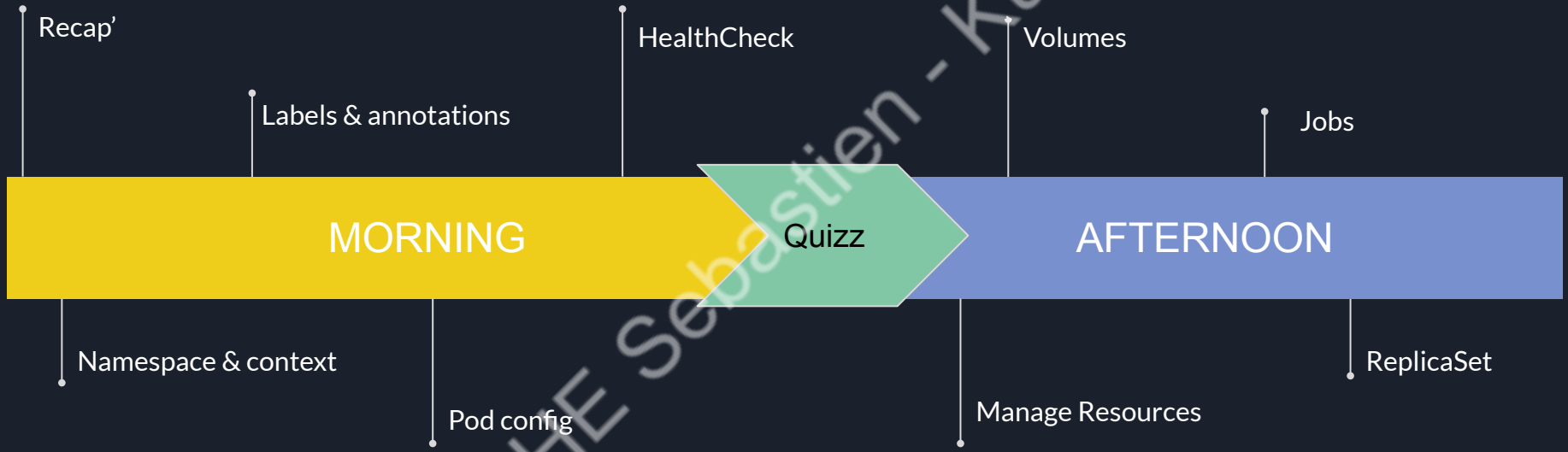
Extract a resource config to YAML form:

```
kubectl get deployment nginx -o yaml
```

Create resource from YAML file:

```
kubectl apply -f config.yaml
```

DAY 2:



Recap’

Namespaces

Usages:

Namespaces are used to partition things. (Rights, applications, rules, usage ...)

Can be also used to start the same pod on different environments, keep in order and to secure application.

Example you can have `dev`, `staging` and `prod` on the same cluster.
List namespaces:

```
kubectl get namespace
```

Usages II:

List resources in a namespace use: `-n <namespace>`

```
kubectl get services -n kube-system
```

Note that when you create a service it will create a private DNS entry like so:

<service>.<namespace>.svc.cluster.local

Note also that not all resources are in namespace (nodes, persistent volumes, ...)

```
kubectl api-resources --namespaced=false
```

Contexts

Usages:

Context is used to define WHO are we and WHERE are we.

To see the actual config (depending your .kubeconfig):

```
kubectl config view
```

List contexts:

```
kubectl config get-contexts | current-context
```

Let’s try:

Let’s create a namespace:

```
kubectl create namespace demo
```

And this time let’s deploy our Nginx in our brand new namespace:

```
kubectl create deployment nginx --image nginx -n demo
```

Let’s create our context and use it:

```
kubectl config set-context demo-context --namespace demo --user kubernetes-admin --cluster kubernetes
kubectl config use-context demo-context
```

Labels & annotations

Usage:

Labels and annotations can be applied to any resource. It helps to order, identify or manage your cluster.

Labels are more destined to internal usage(selectors) and annotations for external/user. It helps you put “**stickers**” to categorize your resources.

Let’s create some Nginx deployments with labels:

```
kubectl create deployment nginx[1,2,3] --image nginx --labels=”env=[x,y,z],usage=Web”
kubectl get pods --show-labels
```

Usage II:

Let's update one (Careful we are modifying deployment here):

```
kubectl label --overwrite deployment nginx3 “usage=Intranet”
```

Applying on a pod will trigger a creation of a new pod.

Why?

Because initially we created a deployment with the value “Web”, this value isn't here anymore K8s will converge to the wanted state. I.e 1 pod with the “Web” label. We got some relationship at creation time but not afterwards.

Usage III:

Let’s delete a label:

```
kubectl label --overwrite deployment nginx3 “usage-”
```

Let’s use a selector to fetch some resources:

```
kubectl get pods --selector “env=prod” --show-label
kubectl get pods --selector “env in (dev,staging)” --show-label
```

Annotation works exactly the same with “annotation” instead. Allows you to keep track of change, store B64 data, urls etc...

Pod Configuration

Pod config:

We are going to focus more on manifest declaration way.

It’s better to use more higher level features such as deployment to benefit from convergence, orchestration, etc...

For pod replication we use controllers such as “Statefulset”(storage), “Deployment”(scaling) and “Daemonset”(monitoring)

Those controllers uses templates of pod.

Inside a pod, the containers have the same network (IP/Port), the same volumes and can communicate through “localhost”.

Minimal configuration:

- Api version
- Kind of the resource
- A metadata name
- At least one container spec

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: nginx
```

Applying our configuration:

- Add namespace
- Add label / annotation

Inside the container spec you can for example override the entrypoint with

`command: [“echo”, “TEST”]`

See also ports configuration from earlier.

And use `apply -f` to create the pod.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  namespace: demo
  label:
    env: staging
spec:
  containers:
  - name: nginx
    image: nginx
```

Health Check

Liveness & Readiness:

Sometimes applications need to be restarted for any reason. Also we can prevent containers in bad shape to join the pool and start receive queries or job. Allows to have 0 downtime during update or deployment (SLA).

Liveness => Restart pod if check fail
Readiness => Take out pod from LB

Both are complementary!

How to check:

3 modes available:

- Command -> exec
- Http -> httpGet
- TCP -> tcpSocket

Configuration:

- initialDelaySeconds -> When do we start ?
- periodSeconds -> Frequency ?
- timeoutSeconds -> Timeout ?
- successThreshold -> How many success ?
- failureThreshold -> How many failures ?

Configure

Configuration:

- Host
- Schem
- Path
- HttpHe
- Port

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  namespace: demo
  label:
    env: staging
spec:
  containers:
  - name: nginx
    image: nginx
    livenessProbe:
      httpGet:
        host: my.nginx.test.company.com
        scheme: HTTPS
        path: /healthcheck
        port: 9999
        httpHeaders:
        - name: Authorization
          value: Bearer Test123
      initialDelaySeconds: 20
      periodSeconds: 3
```

Configure

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  namespace: demo
  label:
    env: staging
spec:
  containers:
  - name: nginx
    image: nginx
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

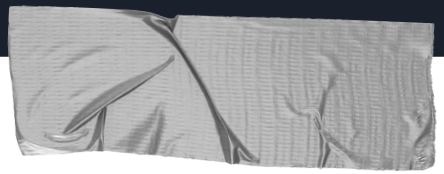
Configure

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  namespace: demo
  label:
    env: staging
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - name: liveness-port
      containerPort: 8080
      hostPort: 8080
    livenessProbe:
      tcpSocket:
        port: liveness-port
      initialDelaySeconds: 5
      periodSeconds: 5
```

Exercise:

Some custom probes to practice:

https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/



QUIZZ TIME!

➔ What is the main usage of a namespace?

Partition things in a cluster

➔ What are the 4 types of services?

➔ What are the usages of volumes?

Managing resources

Resource management:

In order to manage the pod resources you can request(minimal) or limit(maximal).

Request is useful for scheduling and spread pods across the nodes.
Limit is useful for the health of the pods.

Careful to memory leak for example you will have a great number of restart of your pod you you have some.

Those metrics are gathered through the `metrics-server` daemonset.

2 types or resource management CPU and Memory

Configuration:

Resource management is defined under the `spec` key.

In controllers, we will use ResourceRequirements. (Deploy, Daemonset, StatefulSet)

K8s config > Docker config > LINUX Kernel

In order to have those metrics, you will need to install metrics-server, which will provide those to the kubelet API.

Installation of metrics-server:

Apply manifest from github release:

```
kubectl apply -f
https://github.com/kubernetes-sigs/metrics-server/releases/download/v0.3.7/components.yaml
```

See more: https://github.com/kubernetes-sigs/metrics-server

Check install:

```
kubectl top node
kubectl get apiservices | grep metrics
```

Resources:

Manage Memory resource :

Can be expressed in MebiBytes (Mi) -> 50Mi ~ 5%
Or in MegaBytes (500M, 2G ...).

Manage CPU resource :

Can be expressed in m of CPU -> 200m = 0.2 CPU
Or in Number of CPU (1, 0.5 ...)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: busy-pod
spec:
  containers:
  - name: liveness
    image: busybox
    args:
    - /bin/sh
    - -c
    - sleep 600
    resources:
      requests:
        cpu: 0.1
        memory: 200M
      limits:
        cpu: 1
        memory: 500M
```

LimitRange

LimitRange:

A resource is available in K8s world is limitRange. This allow us to define default behaviour, and/or min-max configuration for resource management in a namespace.

From k8s documentation:

- Enforce minimum and maximum compute resources usage per Pod or Container in a namespace.
- Enforce minimum and maximum storage request per PersistentVolumeClaim in a namespace.
- Enforce a ratio between request and limit for a resource in a namespace.
- Set default request/limit for compute resources in a namespace and automatically inject them to Containers at runtime.

LimitRange II:

Using LimitRange implies some constraints such as:

- The LimitRanger controller enforce default if no resource requirements is set, and track usage.
- If you create a resource that violates a LR, API server will fail with a 403 Forbidden
- If a LR is activated in a Namespace, you must specify those or system may reject creation
- LR works at Pod admission stage not already running Pods

LimitRange

Let's create a Nam

Let's try to create

Let's try to crate or

And forEach create no request, and exceed range.

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: test-lr
  namespace: dev
spec:
  limits:
  - default:
      memory: 100Mi
      cpu: 100m
    defaultRequest:
      memory: 50Mi
      cpu: 50m
    max:                    #Acceptable max
      memory: 512Mi
      cpu: 500m
    min:                    #Acceptable min
      memory: 50Mi
      cpu: 50m
    type: Container
```

Quota

Quota:

So you can also define quotas for your namespaces, especially when teams shares a cluster. So you can define a limit of resources allocated to a namespace, a quantity limit of objects that can be created or the total amount of consumed resources.

Same as LR, if someone creates a resources that violates the Quota constraints, he will receive a 403. If you have overlapping configurations, it's first come first served. As the LR this will not apply to already running resources.

Quota II:

You can define quotas on the following resources(Across all Pods in non-terminal state):

- limits.cpu: Maximum sum of CPU limits
- limits.memory: Maximum sum of Mem limits
- requests.cpu: Maximum sum of CPU requests
- requests.memory: Maximum sum of Mem requests

You can also define extended resources such as GPU and limit them by prefixing with `requests.` or `limits.`

Ex: *requests.nvidia.com/gpu: 4*

More: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#extended-resources

Quota III:

You can of cou olumeclaims ...

If you want you K8s Object.

Examples:

- cou
- cou
- ...

And even subty

- ser

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: mem-cpu-demo
spec:
  hard:
    requests.cpu: 1
    requests.memory: 1Gi
    limits.cpu: 2
    limits.memory: 2Gi
    pods: 3
```

Quo

Finally, , and cc sed on the

Let's cr urcesQu or that m

create prio

```yaml
apiVersion: v1
kind: List
items:
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-premium
  spec:
    hard:
      cpu: 1
      memory: 500M
      pods: 3
    scopeSelector:
      matchExpressions:
      - operator : In
        scopeName: PriorityClass
        values: ["premium"]
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-normal
  spec:
    hard:
      cpu: 200m
      memory: 100M
      pods: 3
    scopeSelector:
      matchExpressions:
      - operator : In
        scopeName: PriorityClass
        values: ["normal"]
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: busy-pod
spec:
  containers:
  - name: liveness
    image: busybox
    args:
    - /bin/sh
    - -c
    - sleep 600
    resources:
      requests:
        cpu: 200m
        memory: 100M
      limits:
        cpu: 1
        memory: 500M
  priorityClassName: premium
```

Volumes

What for?

At application conception level, we need to keep as low as possible the storage needs.

For example you can externalise logs (ELK, logserver etc...), you can replace configuration files by ConfigMaps but the rest you will need to store data somewhere.

Files, database ... need some volume to work but some can be ephemeral some others need to be persistent.

K8s Volume types:

Kubernetes have a few volume types:

- hostPath: Shared between Host and Pod (depend of the node)
- emptyDir: Not persistent but can share between pods/containers
- persistentVolumeClaim: For example Google cloud disk
- external: NFS, Volumeclaim, GlusterFS, Vsphere ...

And so many others like secret, local, configmap... list and description can be found here: https://kubernetes.io/docs/concepts/storage/volumes/

HostPath

Quite similar a … e host but **careful,** we ar … heduled on another one s … e is for daemonset m … dedicated node or on all …

Types of volur …

- Direct …
- FileOr …
- Socke …
- CharD …
- Block … or Create) 755

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 8888
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: my-hp-volume
  volumes:
  - name: my-hp-volume
    hostPath:
      path: /tmp/testVolume
      type: DirectoryOrCreate
```

EmptyDir:

You can share a ainers of a pod.

Everytime the p d.

They can be mo need I/O)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 8888
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: my-ed-volume
  - name: busybox
    image: busybox
    args:
    - /bin/sh
    - -c
    - sleep 600
    volumeMounts:
    - mountPath: /tmp/test
      name: my-ed-volume
  volumes:
  - name: my-ed-volume
    emptyDir: {}
      # medium: Memory  # <- RAM DISK
```

PersistentVolume

PersistentVolume:

System of volume provisioning, it's separated in 2 kinds:

- persistentVolume: Provisioning of the volume
- persistentVolumeClaim: Consume the volume

PV => PVC => POD
Provisioning => Claim => Usage

A loop in the master binds and watches new PVCs. Once bound, PVCs are Exclusive, no matter how they were bound. PV have to match the claim or higher. You cannot have multiple PV to match one PVC.

PersistentVolume II:

PersistentVolume types are implemented as plugins, see the full list here:
https://kubernetes.io/docs/concepts/storage/persistent-volumes/#types-of-persistent-volumes

Types of accessMode:

- ReadWriteOnce: Mounted on a single Pod (RWO)
- ReadOnlyMany: Read-only mount on multiple Pods (ROX)
- ReadWriteMany: Read and Write on multiple Pods (RWX)

Get persistent Volumes with:

```
kubectl get pv
```

PersistentVolume III:

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  volumeMode: Filesystem #can be Block -> App need to handle Raw block support
  storageClassName: manual
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  hostPath:
    path: "/tmp/testVolume"
```

PersistentVolumeClaim

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  storageClassName: manual
  resources:
    requests:
      storage: 4Gi
  # selector:
  #   matchLabels:
  #     release: "stable"
  #   matchExpressions:
  #     - {key: environment, operator: In, values: [dev]}
```

PersistentVolumeClaim II:

Now let's use c

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 8888
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: my-pvc-volume
  volumes:
  - name: my-pvc-volume
    persistentVolumeClaim:
      claimName: my-pvc
```

Some important points:

In order to have a portable configuration:

- Include PVC in your config Bundle
- Do not include PV in your config
- Give the user the possibility to provide the storageClass
- Watch for PVCs that are not bound, in case the cluster don't have the dynamic storage support.

Dynamic Provisioning

Dynamic Provisioning:

Instead of creating PV, a cluster Admin can create StorageClass objects to expose some flavours of Storage specifying a volume provisioner (plugin).

This is not supported in local mode.

See more here:
https://kubernetes.io/docs/concepts/storage/storage-classes/#provisioner

A persistent Disk will be provisioned, when the claim is deleted, the volume is destroyed.

Jobs

Jobs:

When you need to run Tasks (cron, CI/CD, Side-Process) you can use a k8s Job. It will run one or more pod(s) and ensure they terminate successfully.

When reaching a number of successful completions, Job is complete.

If you delete the job, the pods will be cleaned up.

You can run Pods in parallel.

Writing a Job:

In the `spec` section, you need to provide a pod Template. Same schema as a Pod without `kind` and `apiVersion`.

You can define restartPolicy with `Never` or `OnFailure`

You can use `spec.completions` to define how much completions you want and `spec.parallelism` to define how much job you want to run at the same time.

Ex: completions: 8
paralellism: 2

CronJob

CronJobs:

Basically it's a job with an embedded schedule. Don't forget to make them idempotent, they benefit of the same RestartingPolicy. You can declare the schedule with a Cron format string or for example `@hourly.

You can specify a `spec.startingDeadlineSeconds` in case the job misses the Scheduled time and you don't want it to run if it's too late.

If this field is set, the controller will count how many missed schedule occured.

CronJ

You can c

`spec.suc
fields are

Define ho

ew one

mit`

```yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *" #@hourly
  startingDeadlineSeconds: 10
  concurrencyPolicy: Allow
  successfulJobsHistoryLimit: 1
  failedJobsHistoryLimit: 5
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

ReplicaSet

What is it:

This allow you to duplicate pods, can be used in Deployment, and it's a fixed number of instances (This is not scaling).

We have two ways of describing it:

- Attached: The ReplicaSet have Pod template
- Detached: The ReplicaSet target distant Pod

Get replicaSet:

```
kubectl get rs
```

Example:

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
spec:
  replicas: 2
  selector:
    matchLabels:
      app: front
  #   matchExpressions:
  #     - {key: environment, operator: In, values: [dev]}
  template: #Template of the pod
    metadata:
      labels:
        app: front
    spec:
      containers:
      - name: nginx
        image: nginx
```

ReplicaSet HPA

Horizontal Pod Autoscaler

HPA:

So this allows us to
complementary w
be use
s, Openshift...)

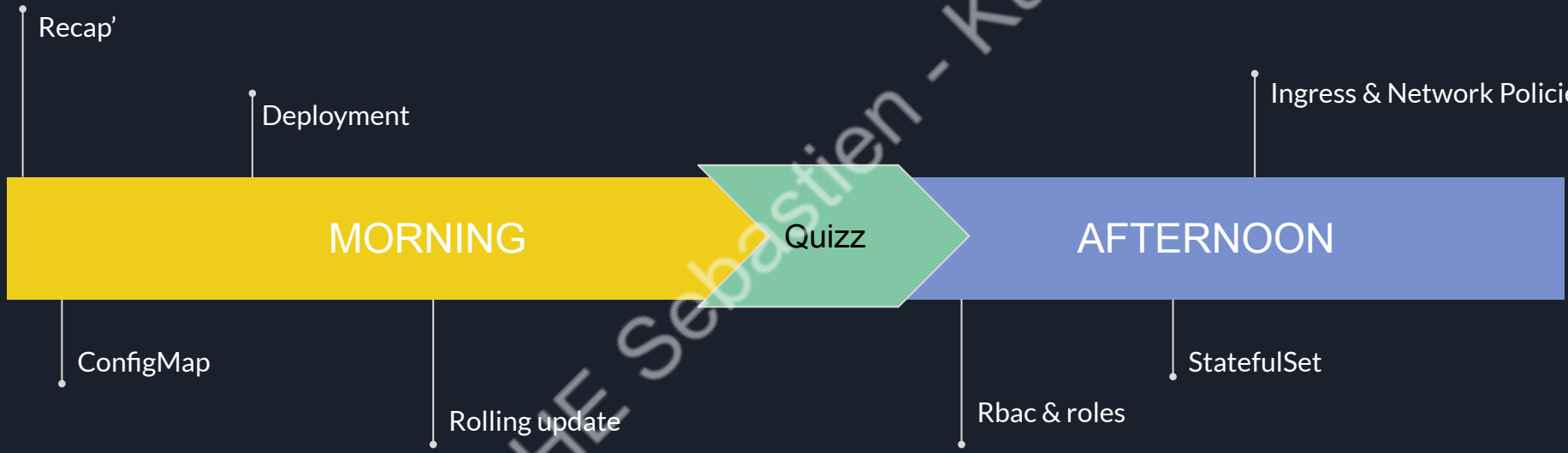
We can multiply th
d a max, and set a
trigger threshold.

It's better to use w
aling without
metrics!

We already used it
orm.

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: my-hpa
spec:
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: nginx
  minReplicas: 2
  maxReplicas: 5
  # targetCPUUtilizationPercentage: 20
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
  - type: Resource
    resource:
      name: memory
      target:
        type: AverageValue
        averageValue: 50
```

DAY 3:



ConfigMap

ConfigMaps usage:

ConfigMaps are useful to centralize, improve and manage configurations.

Instead of having files on a server, we now have a kind of “Database” we can access from our cluster.

You can isolate them, secure them, manipulate them, share them.

For the secrets, it’s encoded in Base64.

Careful for security, secrets can be manipulated through manifests.

ConfigMaps creation:

Create a simple configmap/secret:

```
kubectl create cm test --from-literal=KEY=Value
kubectl create secret generic testsecret --from-literal=KEY=Value
```

Retrieve configmap/secret:

```
kubectl get cm
kubectl get secret
```

Create from file:

```
kubectl create cm testfile --from-file=config.yaml
```

ConfigMaps edit:

Replace (Security issue, need to reboot pods):

```
kubectl replace -f newcm.yaml
```

Edit (No need reboot):

```
kubectl edit cm test
```

Delete:

```
kubectl delete cm test
```

ConfigMaps manifest way:

Let’s create our CM with a manifest:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-cm
data:
  key1: value1
  test: echo
  multiple: |
    pod.val=nginx
    deploy.replicas=3
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 8888
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: myCMVolume
volumes:
- name: myCMVolume
  configMap:
    name: my-cm
    items: # if this is not precised it will be a dir if you create a cm from multiple files
    - key: multiple
      path: index.html
```

How to use our CM:

Let’s create a cm that contains multiple files:

```
kubectl create cm cm-dir --from-file=index.html --from-file=test.html
```

NodeSelector

Assign pod to node:

You can constraint a Pod to be able to only run on a specific Node.
It's recommended to use the labels Selector to do so. Generally you don't need such constraint, as the scheduler should chose wisely.

But sometimes you want a SSD or target a specific node Full of RAM ...

So we can use `spec.nodeSelector` it specifies a map of Key-Value pairs.

The node must have each one of those as labels.

Assign pod to node II :

Let’s add a label to a Node:

```
kubectl label nodes <nodeName> key=value
Kubectl get nodes --show-labels
```

And then let’s create our pod with the nodeSelector value.

Node affinity:

This is similar to nodeSelector, but in a softer way.

There is two types of affinity:

- requiredDuringSchedulingIgnoredDuringExecution: This is the “hard” way that the requirements have to be met.
- preferredDuringSchedulingIgnoredDuringExecution: This is the “soft” way

If the rule doesn’t match at runtime the pod won’t be evicted.

Deployment

Deploying easy peasy lemon squeezy:

This will allow us to manage at a larger scale the pods and replicaSet.

Having a better and smoother way to upgrade, update, manage our application.

Having fallback/rollback strategies.

All of this in one single tool.

Deploying types:

There is 2 ways of deploying:

- RollingUpdate: Implement a strategy to progressively replace the old stack.
- Recreate: Destroy v1, create v2 (Brutal)

There is 2 advanced deployment methods you can build on top:

- Blue/Green: Duplicate stack, test it, switch the service to the new one
- Canary Deployment: Sending a % of the traffic to the new version, when validated do the roll-out. (ISTIO)

Let's

You car
configu

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.17 # next is 1.19
        imagePullPolicy: always #Force check Hash and pull if necessary
        name: nginx
        ports:
        - containerPort: 8888
        readinessProbe:
          httpGet:
            path: /
            port: 8888
          initialDelaySeconds: 10
          periodSeconds: 10
          successThreshold: 1
```


rting

d

in to

is set

Failures:

Here is some possible failure factors that can happen during a deployment:

- Insufficient Quota
- Readiness probe Failures
- Image pull errors
- Insufficient permissions
- Limit Ranges
- Application Runtime Misconfiguration

If you didn’t provide a progressDeadlineSeconds value, the deployment will in ProgressDeadlineExceeded after 10 minutes.

Failures II:

All actions that apply to a complete Deployment also apply to a failed Deployment.

You can scale it up/down, roll back to a previous revision, or even pause it if you need to apply multiple tweaks in the Deployment Pod template.

You can set `spec.revisionHistoryLimit` to specify how many old replicaSets you want to retain.

Rolling Update

Recreate:

Let’s add a strategy, recreate is brutal and not really the most used one but here is the configuration:

```
    matchLabels:
      app: nginx
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: nginx
    spec:
```

Rolling Update:

It’s the default strategy of a deployment. You can customize the rolling update using those values:

- maxUnavailable: How many pods can be done during the process *(25%)*
- maxSurge: How many pods we can create above the requested replicas *(25%)*

Rollout

Rollout:

Rollout will allow us to manage and keep track of the deployments (versioning) and a quick access to fallback/rollback without file editing.

We are going to use annotations for that:

```
template:
  metadata:
    labels:
      app: nginx
    annotations:
      kubernetes.io/change-cause: "bump Nginx 1.17"
  spec:
    containers:
    - image: nginx:1.17 # next is 1.19
```

Rollout:

Check rollout status:

```
kubectl rollout status deploy nginx-deployment
```

Pause/Resume rollout:

```
kubectl rollout pause/resume deploy nginx-deployment
```

Get rollout history:

```
kubectl rollout history deploy nginx-deployment [--revision X]
```

Rollback:

```
kubectl rollout undo deploy nginx-deployment [--to-revision X]
```

Affinity

Base Labels:

By default K8s create some labels (depending also of the CloudProvider), that allows you to use some NodeSelector/Affinity:

- kubernetes.io/hostname: Hostname
- topology.kubernetes.io/zone: Cloud zone
- topology.kubernetes.io/region: Cloud region
  This is automatically defined by the CP. If it makes sense to your cluster you should add them too. The automatic Pod spreading behaviour is modified to reduce impact of zone failures. Same for volume behaviour as they cannot be attached across zones.

- node.kubernetes.io/instance-type: Set from the CP. If you want to target specific machines.
- kubernetes.io/os: OS of the node (Useful if you mix OS)
- kubernetes.io/arch: Architecture of the node (Useful if you mix different architectures)

Inter-pod Affinity:

So you can define some preference of a pod to be scheduled or not where an existing pod is already there or not.

=> Let's create a pod that have a Label `friendly: high`
=> Let's create a pod that MUST be on the same node
=> Let's create a pod that WANT to be on the same node
=> Let's create a pod that MUST be on another node
=> Let's create a pod that WANT to be on another node

This affinity can be even more useful, when used with a Deployment, StatefulSet... to colocate a set of pods.

Practice:

Let’s create a redis server and a web server that will be scheduled on each node and by Pair.

So we will obtain 1 redis-cache and 1 Web server on each node

Taints & Tolerations

Definition:

Affinity was a way to attract Pods to some Nodes, taints are the opposite: How to repel Pods from a node.

Tolerations applies to Pods and allow them to schedule on Nodes with matching Taints.

Taints and Tolerations works together, To be sure that we DO NOT schedule on inappropriate Nodes.

Practice:

Let’s add a taint to one of our Node:

```
kubectl taint nodes kubnode maintenance=true:NoSchedule
```

We can remove it with:

```
kubectl taint nodes kubnode maintenance:NoSchedule-
```

Let’s create a Pod that don’t have the toleration, then one that does.

Taints:

Here is a list of effects:

- NoSchedule: Don’t schedule on that node without Toleration
- PreferNoSchedule: Try not to place Pod on that node without Toleration
- NoExecute: Immediately evict all Pods without Toleration

=> Let’s create a Deployment with a few pods that will be on multiple nodes. Then we will add a Taint to evict the pods.

Headless Services

Usage:

Sometimes you might want a single Service IP and no load-balancing. So you can create a **“Headless”** Service by specifying `None` to the clusterIP.

A clusterIP is not allocated and K8s does not handle theses, no LB and no Proxy. DNS is automatically configured depending on whether or not we defined the selectors:

- With: Creation of an Endpoints record in the API, return record that point to Pod using this service
- Without: No endpoint created in the API but DNS looks for and configure CNAME records for ExternalName types and records of any Endpoint that share a name with the service.

StatefulSet

Usages:

This is used when you want to deploy stateful Apps (Databases For example)

This gives us some particularities:

- Launch order
- Keep in memory the volumes
- Unique network identifiers

This way we can guarantee the status of the stateful App after deployment.

Limitations:

This also confront us to some limitations:

- The storage need to be pre-provisioned by admin or requested using a Persistent volume Provider and matching the StorageClassName (*https://github.com/kubernetes/examples/blob/master/staging/persistent-volume-provisioning/README.md*)
- Deleting/scaling does not delete volumes
- Require a Headless service (You need to create it)
- No guarantee of pods termination when it is deleted
- Default rolling update can create broken state

Try it:

=> Let’s create directory to pretend we have a replicated storage:

We need 4 directories /tmp/pv0| 1 | 2 | 3 on each Nodes

Create 4 index.html files containing the name of the PV in it
Then create 4 PVs using those.

Try it:

We will start by creating some PV:

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: stateful-pv1
spec:
  storageClassName: manual
  capacity:
    storage: 200Mi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/tmp/pv1"
```

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: stateful-pv2
spec:
  storageClassName: manual
  capacity:
    storage: 200Mi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/tmp/pv2"
```

...

Try it:

Then we will create the Statefulset:

We can see that our pods are created 1 by 1
In order {0-N}

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: my-sts
spec:
  serviceName: nginx
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      storageClassName: manual
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 100M
```

Try it:

We also need to add the Headless Service. This will allow us to have the pods in the same statefulSet to communicate.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
```

Service details:

Now pods attached to the headless service are now reachable with the following pattern:

<PodName>.<ServiceName>.<Namespace>.svc.cluster.local

If you don’t precise the podname, you‘ll be load balanced as expected behaviour.

Identity:

So now our pods have their identity, they'll claim the same PV, they have an index and they also have a DNS record that allows us to reach each pod in the cluster.

```
kubectl run -i --tty --image busybox:1.28 dns-test --restart=Never --rm
nslookup my-sts-0.nginx
wget my-sts-0.nginx.default.svc.cluster.local
```

Delete the pods:

```
kubectl delete pods -l app=nginx
Kubectl get pvc -l app=nginx
```

Scaling :

Let’s scale up our STS by modifying our manifest.

When we scale up we have a new Pod with the index = N+1

When we scale down the pods are removed one by one from {N-N-X}

Let’s watch again our PVC.

Our PVCs are still bound to PVs and are ready to be reclaimed Anytime.

Rolling update :

Let’s update our STS file by implementing our Rolling update strategy.

The STS controller goes in reverse ordinal Order, terminates a pod and wait to transition to Running and ready before moving to the next one.

If a pod fails in the chain, it won’t go to the next one and restore the failed pod to his current version. The controller will try to keep the application healthy and the update consistent.

Rolling update staging:

We can stage an update by using the `partition` configuration.

Basically we can define a range of pods between N and `partition` that need to be updated.

Let's put half of our pods to the partition update.

And let's get their images:

```
for p in 0 1 2 3; do kubectl get pod "my-sts-$p" --template '{{range $i, $c := .spec.containers}}{{$c.image}}{{end}}'; echo; done
```

We can roll out our canary Deployment by decrementing our partition until 0.

Deleting StatefulSets:

If you want to delete the STS but keep the pods, you can use the `--cascade=false` flag to the command.

```
kubectl delete statefulset my-sts --cascade=false
```

Then delete a pod.

Then recreate the STS.

Then finally let's delete our STS with cascading.

We can force parallel behavior by using the `spec.podManagementPolicy: Parallel`

Daemonset

Daemonset:

A daemonset ensure that all nodes run a Copy of a pod. Or at least to all the one we requested. As soon as a new node is added to the cluster an instance of this pod will run on it. When nodes are removed, thoses pods are garbage collected.

Typical uses of a DaemonSet:

- Running a cluster storage Daemon
- Running a logs collection Daemon
- Running a node Monitoring Daemon

Daemonset II:

If you want to filter the nodes that runs the pod of the daemonset, you need to specify a `spec.template.spec.nodeSelector` or a `spec.template.spec.affinity`, if you don't it will be created on all nodes.

Daemon pods respect taints and tolerations, the following tolerations are added automatically:

- node.kubernetes.io/not-ready: NoExecute
- node.kubernetes.io/unreachable: NoExecute
- node.kubernetes.io/disk-pressure: NoSchedule
- node.kubernetes.io/memory-pressure: NoSchedule
- node.kubernetes.io/unschedulable: NoSchedule
- node.kubernetes.io/network-unavailable: NoSchedule

Daemonset III:

Here is some patterns on how to communicate with Pods in a DaemonSet:

- Push: Pods are configured to send updates to another service
- NodeIP/Port: They can use hostPort, so the Pods are reachable via nodeIPs
- DNS: Create a Headless servicem then discover DaemonSets using the endpoints resource
- Service: Create a service with the same pod selector and reach a daemon on a random node.

Daemonset IV:

Let’s create a DaemonSet using this template:

https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/controllers/daemonset.yaml

Now we are going to add on it a RollingUpdate strategy, and perform a version upgrade.

=> Add the strategy, and bump to version 2.6.0

Ingress

Ingress:

Ingress is an entrypoint to expose HTTP and HTTPS route outside the cluster.

It’s usually used without a serviceType NodePort or LoadBalancer.

Ingress is a resource in Beta, if you want to use it you will need to deploy and Ingress controller.

https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

Configuration:

Let’s see how to configure it:

- Host: Optional, if none then this apply to all incoming requests
- Path: List of path to associate to a backend
- Backend: Defined by ServiceName and ServicePort

You can also specify a `spec.tls` entry to specify a list of hosts and the associated secret that contains the CRT and KEY

Exercise:

Let’s do the following tutorial:

https://kubernetes.io/docs/tasks/access-application-cluster/ingress-minikube/

Security Context

Configuration:

In order to specify Security Context for a Pod, we need to add the `spec.securityContext` field in the configuration.

This Policy will apply to all the containers in the Pod or you can specify for each container.

https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.18/#securitycontext-v1-core

Advanced features:

Here is a list of some access control settings:

- allowPrivilegeEscalation: Is a process is able to gain more privilege than its parent. Always true if you run as privileged
- capabilities: add/drop POSIX capabilities
- privileged: Equivalent to root on the host (false)
- procMount: Type of procMount to use for the containers
- readOnlyRootFilesystem: Is this container have a ReadOnly root filesystem (false)
- runAsGroup: GID to run the entrypoint
- runAsNonRoot: Must run as non root. Kubelet validate at runtime the non-usage of UID 0
- runAsUser: the UID to run the entrypoint
-

More to see in the PodSecurity Context Documentation

Exercise:

Let’s run a pod that will force all of his 2 busybox containers to run:

Have an EmptyDir volume.

User ID -> 1000
User group -> 3000

And one of his containers does Not allow privilege escalation.

Then modify the Manifest to force the second container to run as User ID 2000.

Pod security standards

PodSecurityPolicy:

It’s possible to create a Policy at cluster level to ensure security guidelines are followed.

There is 3 profiles that are available and customizable:

- Privileged: Unrestricted, widest possible level of permissions
  https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/policy/privileged-psp.yaml
- Baseline/Default: Prevent known privilege escalations
  https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/policy/baseline-psp.yaml
- Restricted: Heavily restricted policy, pod hardening best practices.
  https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/policy/restricted-psp.yaml

PodSecurityPolicy:

Security context will apply policy at runtime, they are part of the pod definition.

Security policies are control plane to enforce settings in the Security Context. Other tools and initiatives are developed (OPA GateKeeper)

Pod security context have no effect on Windows.

RBAC

Definitions:

RBAC allow us to control users and applications using RoleBasedAccessControl:

- Cluster Role: (Or Role for a specific namespace)Role and profile allowing some accesses/Actions/Resources in the cluster
- Service Account: Applicative User
- Cluster Role Binding: Association of a role and a Service Account

Define Roles > Create SAs > Bind Roles to SAs

ClusterRole

Cluster Role:

ClusterRole defines the rules to apply to the group (Note Role is used for a specific namespace).

Rules are defined by the resource you want to match and the verbs (actions) you want to apply:

- Create: POST
- Get, List, Watch: GET
- Update: PUT
- Patch: PATCH
- Delete, Deletecollection: DELETE

Exercise:

Let’s create a Role in the default namespace that can be used to :

- Have read access to Pods

Let’s create a ClusterRole that can be used to :

- Have read access to Secrets

Cluster Role Binding

Usage:

A RoleBinding or a ClusterRoleBinding allows us to join users,groups or Service account on one side and permissions on the other side.

You can:

- Reference any Role from a RoleBinding
- Reference a ClusterRole from a RoleBinding
- Reference a ClusterRole in a ClusterRoleBinding

Exercise:

=> We want now to create a RoleBinding that grants the “pod-reader” Role to your user (sderoche for me) in the default namespace

=> Now let’s add to our user the ability to read secrets in the default namespace using the previous ClusterRole we made “secret-reader”

=> Now let’s create a ClusterRoleBinding that give the ability to read secrets everywhere if you are in the group “admin”

More information:

After you bound a User,Group or SA to a Role/ClusterRole, you cannot change this one.

To modify you need to remove the binding object and create a new one.

-> If you grant someone the update ability on this resource make this person able to manage the list of subjects but not role itself

-> Binding to a different Role/ClusterRole is fundamentally different. By forcing delete/create we ensure that it is intended this way.

Be precise:

You can be more specific/precise in your Role/ClusterRole declaration by delimit the subresource.

Example to allow someone to read Pods and the logs of the pod you can define as follow:

Resources: [“pods”, “pods/log”]

You can also specify a resourceNames list to allow:

Resources: [“configmaps”]
resourceNames: [“cm-1”, “cm-2”]

Aggregate:

You can aggregate ClusterRoles into one, by setting an AggregationRule.

This rule defines a label selector to match the ClusterRoles that should be combined.

https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.18/#aggregationrule-v1beta1-rbac-authorization-k8s-io

You can use the label rbac.example.com/aggregate-to-XXX to either aggregate to existing role (admin, edit, view) or a custom one you created

Exercise:

Let’s create an AggregatedClusterRoleRule that will match our label

Let’s try by creating 2 ClusterRoles with the label:

One that can read “secrets” the other that can read "services", "endpoints", "pods"

`rbac.example.com/aggregate-to-monitoring`

Service Account

Service Account:

Let’s create a Service Account!

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sderoche
```

Users can be authenticated through different Authproviders (LDAP, OAuth2, SAML, OpenID...)

See https://kubernetes.io/docs/reference/access-authn-authz/authentication/

A

Let
Bir

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: clusterRole1
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  verbs: ["get", "list", "watch", "patch"]
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["describe"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRolebinding
metadata:
  name: clusterRoleBinding1
subjects:
- kind: ServiceAccount #can be User, Group
  name: sderoche #system:serviceaccounts:dev -> All SA in namespace "dev"
  namespace: default
roleRef:
  kind: ClusterRole
  name: clusterRole1
  apiGroup: rbac.authorization.k8s.io
```

Token:

When we describe our SA, we can see a secret has been created

Let's retrieve it:

```
kubectl get secrets XXX -o yaml
```

So now we have our SA token to communicate with K8s API. We will use this later with a custom Pod.

Usage

Using It:

Let’s create a Pod, that have our secret Token in volume, and call k8s API.

```
kubectl get secrets XXX -o yaml
```

Specs of the pod:

-> Alpine, using our SA, entrypoint sleep 1000, secret mounted in “/var/run/secrets/kubesecrets”

=> Then connect in the pod, install curl

```
apk add curl
```

Using It:

Retrieve the Token value in a VAR:

```
TOKEN=$(cat /var/run/secrets/kubesecrets/token)
```

Test connectivity to our API :

```
Curl -H “Authorization: Bearer $TOKEN” https://kubernetes/api --insecure
```

Try to request Pods

```
Curl -H “Authorization: Bearer $TOKEN” https://kubernetes/api/v1/pods --insecure
```

Try to request Configmaps

```
Curl -H “Authorization: Bearer $TOKEN” https://kubernetes/api/v1/configmaps --insecure
```

Using It:

Let’s update our Role to add the “list” verb to allow GET.

And curl again the API.

Custom Resources

Custom Resource:

A custom Resource is an extension of the K8s API, that is not necessarily available in the default K8s installation.

Many of the core K8s functions are built using CRs to make it more modular.

So you can add or remove them through a dynamic registration, once installed users can use them(with the required access Rights) just like any other resource.

In order to know if you should or not add CR to your cluster and how (API aggregation or standalone API) check this:
https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#should-i-add-a-custom-resource-to-my-kubernetes-cluster

2 ways to reach the goal:

You can go through using:

- CustomResourceDefinitions: Specify a name and a Schema, this way you need to write your own API, but you have less flexibility.
- API Server Aggregation: you can use the aggregation layer, and provide specialized implementations by writing your own code. The main API delegate requests to you making them available to all clients.

CRDs	Aggregated API
Do not require programming. Users can choose any language for a CRD controller.	Requires programming in Go and building binary and image.
No additional service to run; CRDs are handled by API server.	An additional service to create and that could fail.
No ongoing support once the CRD is created. Any bug fixes are picked up as part of normal Kubernetes Master upgrades.	May need to periodically pickup bug fixes from upstream and rebuild and update the Aggregated API server.
No need to handle multiple versions of your API; for example, when you control the client for this resource, you can upgrade it in sync with the API.	You need to handle multiple versions of your API; for example, when developing an extension to share with the world.

Authorization:

CRDs benefits from the same rules as your k8s API Server and can rely on RBAC.
By default RBAC roles won't have the access rights to your new resource. (Except cluster-admin or * roles)

You need to explicitly grant access (Using the label to aggregate to already existing Roles for example.)

For aggregated API you can or not use the k8s API server Authentication, Authorization and Auditing

Create a CRD:

Let’s do the Tutorial on:

https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/

General Good Practices

Some tips /conf:

Before ending this session, let's see some good practice you might want to apply.

Configuration:

- Use the latest stable API
- Store in a version control before applied to the cluster (Also helps cluster restore and recreation)
- Group related resources manifests in the same file
- You can apply commands on a directory rather than on a file (-f)
- Don't put all the fields in your config if you use default value
- Use annotations to describe your resource

Some tips /Pods:

Except when testing don’t use naked pods, they are bound to nothing and won’t be rescheduled if a node fails.

Use deployment instead and if you expect the pod to disappear in a such scenario use `restartPolicy: Never`.

Some tips /service:

Create services before the associated backends, this way k8s will populate the containers with some environment variables:

- NAME_SERVICE_HOST
- NAME_SERVICE_PORT

Don’t precise hostPort in your service if not necessary. Each <hostIP, hostPort, protocol> combination must be unique.

You can use the `kubectl port-forward` for debug to avoid permanent creation of a hostPort.

Same for hostNetwork, for the same reasons as hostPort.

Use HEADLESS services if you don’t need kube-proxy LB.

Some tips /labels /images:

Use Labels, eat labels, sleep labels. They are really important, and it become more efficient to use them when you have a large amount of resources running in a big cluster.

https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/#using-labels-effectively

Use the imagePullPolicy according to your needs, be careful to the usage of :latest and no tags.

More resources:

For REALLY large clusters:
https://kubernetes.io/docs/setup/best-practices/cluster-large/

Backup and restore K8s cluster:
https://medium.com/velotio-perspectives/the-ultimate-guide-to-disaster-recovery-for-your-kubernetes-clusters-94143fcc8c1e

Multi-zone Cluster:
https://kubernetes.io/docs/setup/best-practices/multiple-zones/

That’s all folks

Thank you :D