Kubernetes Operator Training

Labs

Prerequites

The labs will be done remotely on provided virtual machines.

Trainees should have access to remote virtual machines, either:

- with ssh
- with a web navigator using [Apache Guacamole]

Trainees will have to run everything as root (`sudo su`).

The trainer will provide required information.

Each trainee will be attributed 4 virtual machines.

Lab 1

1.1 Container runtime

The goal of this first lab is to install the container runtime on your server and check that it's working fine.

1.1.1 Docker

Docker is pre-installed on all the node

- Check that Docker is installed correctly and running:

```
docker container run hello-world
```

- You should see this:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

1.1.2 Containerd & co (Optional)

As an alternative to Docker, you can use Kubernetes with containerd. Containerd is a minimal container runtime, as such it relies on other tools for many functionalities:

- runc: CLI tools for spawning and running containers
- CNI: Container Network Interface, to configure containers network interfaces
- crictl: CLI tool to interact with CRI (Container Runtime Interface) compatible runtimes

To use containerd, just see the following steps:

- Download containerd and dependencies:

```
curl -LO https://github.com/containerd/containerd/releases/download/v1.2.2/contai
curl -LO https://github.com/opencontainers/runc/releases/download/v1.0.0-rc6/runc
curl -LO https://github.com/containernetworking/plugins/releases/download/v0.7.4/
curl -LO https://github.com/kubernetes-sigs/cri-tools/releases/download/v1.13.0/c
```

- Install containerd binaries to path:

```
tar xf containerd-1.2.2.linux-amd64.tar.gz -C /usr/local
```

- Install runc binary to path:

```
cp runc.amd64 /usr/local/bin/runc
chmod +x /usr/local/bin/runc
```

- Install cni-plugins to path:

```
mkdir -p /opt/cni/bin
tar xf ~/cni-plugins-amd64-v0.7.4.tgz -C /opt/cni/bin
```

- Create containerd configuration:

```
cat << EOF > /etc/containerd/config.toml
[plugins]
  [plugins.cri.containerd]
    snapshotter = "overlayfs"
    [plugins.cri.containerd.default_runtime]
      runtime_type = "io.containerd.runtime.v1.linux"
      runtime_engine = "/usr/local/bin/runc"
      runtime_root = ""
    [plugins.cri.containerd.untrusted_workload_runtime]
      runtime_type = "io.containerd.runtime.v1.linux"
      runtime_engine = ""
      runtime_root = ""
EOF
```

- Configure CNI plugin:

```
mkdir -p /etc/cni/net.d/
cat <<EOF > /etc/cni/net.d/10-bridge.conf
{
    "cniVersion": "0.3.1",
    "name": "bridge",
    "type": "bridge",
    "bridge": "cnio0",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "ranges": [
          [{"subnet": "10.240.0.0/24"}]
        ],
```

```
        "routes": [{"dst": "0.0.0.0/0"}]
    }
}
EOF

cat <<EOF > /etc/cni/net.d/99-loopback.conf
{
    "cniVersion": "0.3.1",
    "type": "loopback"
}
EOF

# Disable ipv6 as it has known problems with cni plugins
echo 1 > /proc/sys/net/ipv6/conf/all/disable_ipv6
```

- Declare containerd systemd service by creating the following file:
  - The file is available in your workspace as `./Lab1/containerd.service`

```
cat <<EOF > /etc/systemd/system/containerd.service
[Unit]
Description=containerd container runtime
Documentation=https://containerd.io
After=network.target

[Service]
ExecStartPre=/sbin/modprobe overlay
ExecStart=/usr/local/bin/containerd
Restart=always
RestartSec=5
Delegate=yes
KillMode=process
OOMScoreAdjust=-999
LimitNOFILE=1048576
LimitNPROC=infinity
LimitCORE=infinity

[Install]
WantedBy=multi-user.target
EOF
```

- Install crictl:

```
tar xf crictl-v1.13.0-linux-amd64.tar.gz -C /usr/local/bin/
```

- Configure crictl:

```
cat <<EOF > tee /etc/crictl.yaml
runtime-endpoint: unix:///var/run/containerd/containerd.sock
image-endpoint: unix:///var/run/containerd/containerd.sock
timeout: 10
EOF
```

- Reload `systemctl` daemons and start containerd.service:

```
systemctl daemon-reload
systemctl start containerd
```

- Test containerd:

```
# Download an image
crictl pull busybox

# Create a pod (busybox-pod.yaml is in ./Lab1/busybox-pod.yaml)
crictl runp busybox-pod.yaml

# Create a container in the pod (busybox-container.yaml is in ./Lab1/busybox-cont
crictl create $(crictl pods -q) busybox-container.yaml busybox-pod.yaml

# Start the container
crictl start $(crictl ps -aq)

# Execute a command in the container
crictl exec $(crictl ps -q) ps aux

# Stop and remove the pod
crictl stopp $(crictl pods -q)
crictl rmp $(crictl pods -q)
```

1.2 Kubelet

Install kubelet

Kubernetes binaries are distributed:

- As distribution packages available on specific repositories
- As binaries joined to Release Notes, and grouped by scope:
    - Client binaries
    - Server binaries
    - Node binaries
- Directly, as single binaries
    - https://storage.googleapis.com/kubernetes-release/release/${RELEASE}/bin/linux/${ARCH}/kubelet
    - https://storage.googleapis.com/kubernetes-release/release/${RELEASE}/bin/linux/${ARCH}/kubectl
- Even though they are not configured and running, kubelet, kubectl and kubeadm are already installed on your nodes
- Check the kubelet command parameters: `kubelet --help`

Many configuration options of the kubelet can be set by command line parameters but this is not the recommended way anymore. You should see a `--config` parameter.

Example `kubelet-config.yaml` :

```
kind: KubeletConfiguration
apiVersion: kubelet.config.k8s.io/v1beta1
staticPodPath: /etc/kubernetes/manifests
```

Documentation on these parameters is available here and here

- Start the kubelet with the command `kubelet &`
- Does it work?
- Check kubelet status with:

```
curl -k https://localhost:10250/healthz
```

- Stop the kubelet with `killall kubelet`
- Define a static pod with the following descriptor, by placing it in `/etc/kubernetes/manifests` :
    - The file is available in your workspace as `./Lab1/busybox-pod.yaml`

```
mkdir -p /etc/kubernetes/manifests
```

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
spec:
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: busybox
  restartPolicy: Always
```

- Start kubelet with the command:

```
kubelet --pod-manifest-path=/etc/kubernetes/manifests &
```

- Check created containers with: `docker container ls`
- What do you see?
- Remove the busybox container with: `docker container rm -f <CONTAINER_ID>`
- What happens?

- Stop the kubelet: `killall kubelet`
- List running containers: `docker container ls`
- What do you see?
- Remove the containers: `docker rm -f <CONTAINERS_IDS>`
- List running containers: `docker container ls`
- What do you see?
- Stop the kubelet: `killall kubelet`
- Remove the created directories: `rm -Rf /var/lib/kubelet`

Lab 2

2.1: Certificates

Communications in a Kubernetes cluster should be encrypted and peers generally use TLS mutual authentication. In order to set this up, there are many certificates to generate.

There are various methods available to generate these certificates:

- With openssl and many commands...
- Using CloudFlare SSL (cfssl) command and config files
- Using kubeadm

We'll use this last method as it's way simpler.

Kubeadm

- View which certs will be generated with command:

```
kubeadm init phase certs
```

- All certificates will be created by default in `/etc/kubernetes/pki`
- Check available options with: `kubeadm init phase certs all --help`
  - You can change default destination with: `--cert-dir`
  - You can adapt the API Server exposed certificate with `--apiserver-advertise-address` and `--apiserver-cert-extra-sans`
  - If some certificates/keys already exist in the directory they won't be overwritten. So if you want to use an existing CA for your PKI, just put the `ca.key` and `ca.crt` in the directory and launch the command.
- Generate all certs:

```
kubeadm init phase certs all
```

2.2: Installation

We'll use `kubeadm` to spawn our cluster. Our cluster will be composed of:

- 1 Control-plane node
- 3 Worker nodes

Steps:

- Check `kubeadm` arguments with: `kubeadm init --help`
- Check what will be done by launching the command `kubeadm init phase --help`
- ⚠ Don't spawn the full control plane now! We'll do it step by step.
- Check node requirements:

```
kubeadm init phase preflight
```

Control plane

With kubeadm, control-plane components are spawned as static pods on the node's kubelet. So the first step is to start the kubelet with the right config on this node:

```
kubeadm init phase kubelet-start
```

You should see:

```
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kub
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.ya
[kubelet-start] Activating the kubelet service
```

To connect to the cluster, clients (cluster components, users, ...) must use a kubeconfig file which holds their credentials and information on the cluster (cluster API server endpoint, CA certificate to validate the https exposed certificate, ...). kubeadm will generate these files for us using the already generated certificates.

```
# See which kubeconfig files will be generated
kubeadm init phase kubeconfig
# Generate them all
kubeadm init phase kubeconfig all
```

We will now use kubeadm to create the static pods manifests for control-plane components:

```
# See what will be generated
kubeadm init phase control-plane
# Generate them all
kubeadm init phase control-plane all
```

- Check that control-plane components containers are running:

```
docker container ls
```

- What is happening to the kube-apiserver container? Why?
- Deploy etcd:

```
kubeadm init phase etcd local
```

> Note: when deploying a HA cluster, the command `kubeadm join [...] --experimental-control-plane` take care of making the new control-plane node joining the cluster

- Check that control-plane components and etcd containers are running:

```
docker container ls
```

- What is happening to the kube-apiserver container? Why?

- Configure kubectl for cluster admin:

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

- At this point, you should be able to use `kubectl` to interact with the cluster.

- Check cluster state with `kubectl get nodes`

- Check cluster components pods with `kubectl get pods -n kube-system`

- Finish the control-plane set up, by

  - Marking this node as control-plane (users workloads won't be scheduled on it)

  - Deploying kube-proxy, which handle service to pods traffic distribution

  - Uploading kubelet and kubeadm as ConfigMaps in the cluster

```
kubeadm init phase mark-control-plane
kubeadm init phase addon kube-proxy
kubeadm init phase upload-config all
```

Workers

With Kubernetes 1.12, Bootstrap TLS went GA (General Availability). Using Bootstrap TLS, workers use a token to contact the cluster control-plane and a client certificate is generated on the fly. This certificate will be used by the kubelet to authenticate the node for further communications. The token authentication information only allow to set up this certificate request.

- Generate a bootstrap token which will be used by workers to join the cluster

```
kubeadm init phase bootstrap-token
```

- The last command generated a bootstrap token, but the following one is even simpler as it prints the command which we'll use to join the cluster:

```
kubeadm token create --print-join-command
```

- You can check the validity of tokens with: `kubeadm token list`

To deploy worker nodes, just execute the join command on each of them. When this is done, go back to the first node and check the cluster state:

```
kubectl get nodes
```

2.3: Network solution

At this moment, the cluster isn't in a usable state. You can check it with: `kubectl get nodes`. It still lacks a network addon to enable cross cluster communication. There are many availables, you can check it [there](). We will deploy *Weave Net*.

- Deploy a network solution on the cluster

```
sysctl net.bridge.bridge-nf-call-iptables=1
K8S_VERSION=$(kubectl version | base64 | tr -d '\n')
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=${K8S_VERSION}"
```

- Check that nodes are now ready with:

```
kubectl get nodes -w
```

- Create 2 pods from these descriptors `nginx-pod.yaml` and `shell-pod.yaml`
- Get the IP of nginx pod: `kubectl get pods -o wide`
- Check that the 2 pods can communicate: `kubectl exec shell -- wget -O- <NGINX_IP>`

2.4: DNS

At this point, we've enabled cross cluster pod communication. However, our cluster doesn't have dns service discovery. To be able to contact services by name, we should deploy a cluster DNS. For older versions of Kubernetes, we used `kube-dns` but now you should use `coredns`.

- Deploy CoreDNS on the cluster with kubeadm: `kubeadm init phase addon coredns`
- Create a service pointing to nginx with descriptor: `nginx-svc.yaml`
- Check that the service name is resolved with: `kubectl exec shell -- wget -O- nginx`

2.5: Storage

Setting up a full blown storage provisioner is beyond the scope of this training. To discover the mechanism behind a storage provisioner, we'll use [hostpath-provisioner]().

- Review and deploy the storage provisioner with the provided descriptors:
    - `hostpath-provisioner-deployment.yaml`
    - `hostpath-provisioner-rbac.yaml`
    - `hostpath-provisioner-storageclass.yaml`

- Review and deploy the `test-claim.yaml` and `test-pod.yaml` to the cluster
- Check the pod to know on which node it was deployed: `kubectl get pods -o wide`
- Connect to the node and check that the hostpath was created: `ls /var/kubernetes`

Lab 3

3.1: EFK Deployment

- Check logs manually from various sources (kubelet, container runtime, ...)
- Deploy EFK with provided descriptors: `kubectl apply -f efk/`
- Check EFK deployment state: `kubectl get pods -n kube-system`
- Label nodes with:

```
kubectl label node node-{0,1,2,3} beta.kubernetes.io/fluentd-ds-ready=true
```

- Find Kibana exposed NodePort: `kubectl get svc kibana-logging -n kube-system`
- Connect to Kibana check logs
  - Click on `Discover`
  - Fill index pattern with value: `logstash-*` and click `Next step`
  - Select `@timestamp` for time filter field name and click `Create index pattern`
  - Click on `Discover` again
- Deploy this container producing logs `date-printer-pod.yaml`:

```
kind: Pod
apiVersion: v1
metadata:
  name: date-printer
  labels:
    name: date-printer
spec:
  containers:
  - name: date-printer
    image: busybox
    command: [ "sh", "-c", "while [ true ]; do date; sleep 1; done"]
```

- Check the container logs on kibana

3.2: Prometheus Deployment

- Deploy the example Pod and Service `go-metrics-pod.yaml`

The pod exposes metrics on port 80 and path /metrics

- Get the `go-metrics` pod IP

- From the `shell` pod verify that metrics are exposed on: `http://go-metrics:80/metrics`

Now we will deploy prometheus on our cluster to scrape metrics.

- Use descriptors in `Lab3/prometheus/`
- Check prometheus scraped metrics by going to `http://<NODE_IP>:30090/`
  - Then go to *Status* > *Targets*
- You should see that many targets are already defined and scraped

However, to add the `go-metrics` service to scraped targets it should be annotated with `prometheus.io/scrape=true`

- Annotate `go-metrics` service
- Wait, and verify that the new target shows up in prometheus
- Check that your metrics are available in prometheus by going to *Graph*
  - Type `myapp_elapsed_seconds`
- Check that cluster and node metrics are also available
  - Example cluster metric: `kubelet_running_pod_count`
  - Example node metric: `node_memory_MemFree_bytes`

3.3: Backup/Restore

To make a backup of the etcd store, you must be able to access it. Access to etcd is authenticated with `apiserver-etcd-client.crt` and `apiserver-etcd-client.key` .

- We'll begin to create a secret with the needed certificates:

```
kubectl create secret generic etcd-creds \
 --from-file=/etc/kubernetes/pki/etcd/ca.crt \
 --from-file=/etc/kubernetes/pki/apiserver-etcd-client.crt \
 --from-file=/etc/kubernetes/pki/apiserver-etcd-client.key

kubectl describe secret etcd-creds
```

- Extract etcd listen address:

```
export ENDPOINT="$(ip -4 addr show eth0 | grep -oP '(?<=inet\s)\d+(\.\d+){3}')"
```

- Deploy a simple pod that will be used to access etcd, with the descriptor template `Lab3/etcd-connect-pod.yaml` :

```
cat Lab3/etcd-connect-pod.yaml | envsubst | kubectl apply -f -
```

- Connect inside the connect-etcd pod:

```
kubectl exec -it connect-etcd -- sh
```

- Perform a backup:

```
etcdctl $SSL_OPTS --endpoints https://$ENDPOINT:2379 snapshot save snapshotdb
```

- Check backup:

```
etcdctl --write-out=table snapshot status /snapshotdb
```

3.4: Capacity planning

3.4.1: Cluster metrics

To see current usage of resources on a Node

- Try to get nodes and pods metrics with:
  - `kubectl top nodes`
  - `kubectl top pods`
- What do you see?
- Deploy the metrics server with descriptors available in `Lab3/metrics-server`
- Check metrics-server deployment state with `kubectl get pods -n kube-system`
- Try to get nodes and pods metrics again

3.4.2: Limits/Requests

- Check current declared usage of resources on one of the nodes with: `kubectl describe node node-0`
- Create a pod with the descriptor `Lab3/qos/qos-pod.yaml`
- Check pod assigned QoS with `kubectl describe pod qos-pod`
- Delete `qos-pod` : `kubectl delete po qos-pod`
- Modify the Pod manifest to add `limits` :
  - cpu: 500m
  - mem: 250Mi
- Create the Pod
- Check `resources` values for created pod with: `kubectl get pod qos-pod -o yaml`
- What do you see?

- Check pod assigned QoS with `kubectl describe pod qos-pod`
- Delete `qos-pod` : `kubectl delete po qos-pod`
- Modify the Pod manifest to add:
  - `limits` :
    - cpu: 500m
    - mem: 250Mi
  - `requests` :
    - cpu: 500m
    - mem: 128Mi
- Create the Pod
- Check pod assigned QoS with `kubectl describe pod qos-pod`
- Find the Node on which is deployed the pod `qos-pod`
- Check current declared usage of resources on one of the nodes with: `kubectl describe node <QOS_POD_NODE>`

3.5: Garbage collection

- Connect to worker node-3
- Edit `/var/lib/kubelet/config.yaml`
  - Change: `imagefs.available: 15%` to `imagefs.available: 60%`
  - Change: `evictionPressureTransitionPeriod: 5m0s` to `evictionPressureTransitionPeriod: 10s`
- Download big images, for example:
  - `docker image pull jenkins`
  - `docker image pull python`
- List available images on the node
  - `docker image ls`
- Restart `kubelet` : `systemctl restart kubelet`
- List available images on the node
  - What do you see?
- Edit `/var/lib/kubelet/config.yaml` again

    - Change: `imagefs.available` to `95%`
- Restart `kubelet`: `systemctl restart kubelet`
- List available images on the node
    - What do you see?
- ⚠ Take care `evictionHard` might evict running pods!
- Edit `/var/lib/kubelet/config.yaml`
    - Change back: `imagefs.available: 95%` to `imagefs.available: 15%`
    - Change back: `evictionPressureTransitionPeriod: 10s` to `evictionPressureTransitionPeriod: 5m0s`
- Restart `kubelet`: `systemctl restart kubelet`

3.6: Ingress

Ingress Controller

- Deploy traefik by applying `Lab3/ingress/traefik-rbac.yaml` and `Lab3/ingress/traefik-ds.yaml`
- Check that *pods* are being created

Ingress traefik-ui

- Check `Lab3/ingress/traefik-webui.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: traefik-web-ui
  namespace: kube-system
spec:
  selector:
    k8s-app: traefik-ingress-lb
  ports:
  - name: web
    port: 80
    targetPort: 8080
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: traefik-web-ui
  namespace: kube-system
spec:
  rules:
  - host: traefik.<NODE_IP>.nip.io
```

```
    http:
      paths:
      - path: /
        backend:
          serviceName: traefik-web-ui
          servicePort: web
```

- Modify it, to replace `<NODE_IP>` by the IP address of a *Node*
    - Use your lab description file or launch `curl ifconfig.me` on a *Node* to know it
- Apply it and check that the ingress is created
- Test it by opening `http://traefik.<NODE_IP>.nip.io/dashboard/#/` in the navigator

Ingress whoami

- Deploy `whoami` deployment and service from `Lab3/whoami-pod.yaml`
- Using the previous example, create an ingress exposing the `whoami` service
    - It should be accessible on `whoami.<NODE_IP>.nip.io`
- Check that the ingress is well configured: `kubectl get ingress`
- Check the configuration on the traefik dashboard
    - You should see one or many pods associated to Host:whoami.<NODE_IP>.nip.io
- Check `http://whoami.<NODE_IP>.nip.io/` and make sure you end on different pods of the service

3.8: Affinity

Affinity

In this lab, we will deploy wordpress on our cluster. We'll use descriptors in `Lab3/affinity/`, this deployments will have a unique *Pod* each and we want them to run on the same *Node*. We'll use `affinity.podAffinity.requiredDuringSchedulingIgnoredDuringExecution`

- Create the secret for the mysql database password: `kubectl create secret generic mysql-pass --from-literal=password=PASSWORD`
- Create the deployments
- Check that `wordpress` and `wordpress-mysql` are not scheduled on the same *Node*: `kubectl get pods -o wide`
- Modify the descriptors to make sure the *Pods* are scheduled on the same *Node*

Anti-affinity

- We will re-use the pods deployed in the previous example

- Adapt the description in `Lab/anti-affinity/mysql-backup.yaml`
  - Add anti-affinity to make sure the *Pod* will be created on another *Node* than the previous ones
- Verify it with: `kubectl get pods -o wide`

Clean up

- Clean up all wordpress deployment with: `kubectl delete all -l app=wordpress`

3.9: Taints & Tolerations

- Create a pod using the descriptor in `Lab3/taints/shell-pod.yaml` :
  - You might need to delete the existing *Pod*: `kubectl delete po shell`
  - Notice the `nodeSelector` , to make sure this *Pod* is scheduled on `node-0`

```
kind: Pod
apiVersion: v1
metadata:
  name: shell
  labels:
    name: shell
spec:
  nodeSelector:
    kubernetes.io/hostname: node-0
  containers:
  - name: shell
    image: ubuntu
    command:
      - sh
      - -c
      - "sleep 3600"
```

- Check *Pod* state with `kubectl get pods`
  - What is the *Pod* state? Why?
- Modify the *Pod* descriptor to make it running on `node-0`
  - You can check *Node* taints with: `kubectl describe node node-0`

3.10: Schedulers

- Check `Lab3/scheduler/shell-pod.yaml` :

```
kind: Pod
apiVersion: v1
metadata:
  name: shell
  labels:
```

```
    name: shell
spec:
  schedulerName: default-scheduler
  containers:
  - name: shell
    image: ubuntu
    command: ["sh", "-c", "sleep 3600"]
```

- This *Pod* explicitly use `default-scheduler`
  - Change it to use a new scheduler named: `my-scheduler`
- Check *Pod* state with `kubectl get pod`
  - The `shell` *Pod* should be in pending state. No scheduler is picking this *Pod* to assign a *Node* to it
- Deploy the scheduler configuration `Lab3/scheduler/scheduler-config.yaml`

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-scheduler-config-map
  namespace: kube-system
  labels:
    app: my-scheduler
data:
  kube-scheduler-config.yaml: |-
    algorithmSource:
      provider: DefaultProvider
    apiVersion: kubescheduler.config.k8s.io/v1alpha1
    bindTimeoutSeconds: 600
    clientConnection:
      acceptContentTypes: ""
      burst: 100
      contentType: application/vnd.kubernetes.protobuf
      kubeconfig: "/etc/kubernetes/scheduler.conf"
      qps: 50
    disablePreemption: false
    enableContentionProfiling: false
    enableProfiling: false
    failureDomains: kubernetes.io/hostname,failure-domain.beta.kubernetes.io/zone
    hardPodAffinitySymmetricWeight: 1
    healthzBindAddress: 0.0.0.0:10251
    kind: KubeSchedulerConfiguration
    leaderElection:
      leaderElect: false
    metricsBindAddress: 0.0.0.0:10251
    percentageOfNodesToScore: 50
    schedulerName: my-scheduler
    policyConfigMap: my-scheduler-policy-cm
---
apiVersion: v1
kind: ConfigMap
metadata:
```

```yaml
  name: my-scheduler-policy-cm
  namespace: kube-system
  labels:
    app: my-scheduler
data:
  scheduler-policy.yaml: |
    kind: Policy
    apiVersion: v1
    predicates:
    - name: PodFitsPorts
    - name: PodFitsResources
    - name: NoDiskConflict
    - name: MatchNodeSelector
    - name: HostName
    priorities:
    - name: LeastRequestedPriority
      weight: 1
    - name: BalancedResourceAllocation
      weight: 1
    - name: ServiceSpreadingPriority
      weight: 1
    - name: EqualPriority
      weight: 1
```

- Deploy the scheduler Pod: `Lab3/scheduler/scheduler-pod.yaml`

```yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    scheduler.alpha.kubernetes.io/critical-pod: ""
  creationTimestamp: null
  labels:
    component: kube-scheduler
    tier: control-plane
    app: my-scheduler
  name: my-kube-scheduler
  namespace: kube-system
spec:
  nodeSelector:
    kubernetes.io/hostname: node-0
  tolerations:
  - key: "node-role.kubernetes.io/master"
    operator: "Equal"
    effect: "NoSchedule"
  containers:
  - command:
    - kube-scheduler
    - --config=/etc/kubernetes/scheduler/kube-scheduler-config.yaml
    image: k8s.gcr.io/kube-scheduler:v1.13.2
    imagePullPolicy: IfNotPresent
    livenessProbe:
      failureThreshold: 8
      httpGet:
        host: 127.0.0.1
```

```
        path: /healthz
        port: 10251
        scheme: HTTP
      initialDelaySeconds: 15
      timeoutSeconds: 15
    name: kube-scheduler
    resources:
      requests:
        cpu: 100m
    volumeMounts:
    - mountPath: /etc/kubernetes/scheduler.conf
      name: kubeconfig
      readOnly: true
    - mountPath: /etc/kubernetes/scheduler
      name: kube-scheduler-configmap
  hostNetwork: false
  priorityClassName: system-cluster-critical
  volumes:
  - hostPath:
      path: /etc/kubernetes/scheduler.conf
      type: FileOrCreate
    name: kubeconfig
  - name: kube-scheduler-configmap
    configMap:
      name: my-scheduler-config-map
```

- The `shell` *Pod* should now be scheduled

Clean up

- Delete the shell pod: `kubectl delete po shell`
- Delete the custom scheduler and its configuration: `kubectl delete po,cm -l app=my-scheduler -n kube-system`

3.11: Troubleshooting

- Connect to worker `node-2`
- Execute:

```
sed -i s/config\\.yaml/NOPconfig.yaml/g \
  /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
systemctl daemon-reload
systemctl restart kubelet
```

- Connect to `node-0`
- Check *Nodes* state with `kubectl get nodes -w`
  - The previous *Node* will become `NotReady`
- Try to think how you could see what's wrong

- Check `kubelet` systemd service status with:
  - `systemctl status kubelet`
  - `journalctl -u kubelet -f`
- Fix the problem
- Check that the *Node* state is back to normal on the master

Lab 4

TP 4.1 : Kubeadm upgrade

Initialize an upgradable cluster

Connect to old nodes and initialize an old cluster (1.12)

- Launch `kubeadm init` on `old-0`
- Run the suggested commands to start using your cluster
- When the cluster is up, install the network on it:

```
sysctl net.bridge.bridge-nf-call-iptables=1
K8S_VERSION=$(kubectl version | base64 | tr -d '\n')
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=${K8S_VERSION}"
```

- Connect to `old-1` and `old-2` and issue the `kubeadm join` command
- Check *Nodes* status with `kubectl get nodes -w`
  - Wait for all to be *Ready*

Deploy some workload on the cluster

- Deploy `whoami-pod.yaml` on the cluster

Upgrade the cluster control plane

- On `old-0` install new `kubeadm` version:
  - `apt install kubeadm=1.13*
- Check upgrade plan: `kubeadm upgrade plan`
- Apply proposed upgrade command: `kubeadm upgrade apply v1.13.X`
  - Note that the control plane is partially unreachable during this upgrade
- Upgrade `kubelet` and `kubectl` on `old-0`
  - `apt install kubelet=1.13* kubectl=1.13*`

Rolling update of the nodes

For each *Node*, follow these steps:

- Prepare the node for upgrade:
  - On `old-0` launch: `kubectl cordon <NODE>` to stop *Pods* to be scheduled on it

    - And `kubectl drain <NODE>` to evict *Pods* before the upgrade
    - Add `--ignore-daemonsets` to the command
- Do the upgrade
    - On the node launch: `apt install kubelet=1.13*`
- Wait for the *Node* to be ready again then mark the *Node* as upgraded:
    - `kubectl uncordon <NODE>`

Do the same for each *Node* and you're done!

Lab 5

5.1: Network policies

- Create the *Namespace* `networkpolicy` with `kubectl create ns networkpolicy`
- Create the *Pod* from `Lab5/network-policy/whoami-pod.yaml`
- Create the *Pod* from `Lab5/network-policy/shell-pod.yaml`
- Create the *Pod* from `Lab5/network-policy/other-shell-pod.yaml`
- Create the *NetworkPolicy* from `Lab5/network-policy/network-policy.yaml`

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: networkpolicy
spec:
  podSelector:
    matchLabels:
      name: whoami
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          lab: networkpolicy
    - podSelector:
        matchLabels:
          name: whoami
    ports:
    - protocol: TCP
      port: 80
```

- From the `shell` *Pod*, try to access `whoami` *Service*: `kubectl -n networkpolicy exec -it shell -- wget -O- http://whoami`
- What happends? Why?
- Modify the *NetworkPolicy* so that the `shell` *Pod* is able to access `whoami` *Service* but not the `other-shell` *Pod*
    - Either create a new *NetworkPolicy* or modify the existing one
- Validate your configuration with:
    - `kubectl -n networkpolicy exec -it shell -- wget -O- http://whoami`

        - Should be ok
    - `kubectl -n networkpolicy exec -it other-shell -- wget -O- http://whoami`
        - Should fail with timeout
- When you're done, clean up with: `kubectl delete ns networkpolicy`

5.2: Admission controllers

- List default activated admission controllers for your Kubernetes version
- The command to do it is: `kube-apiserver -h | grep enable-admission-plugins`
- However when you setup a cluster, the `kube-apiserver` is spawn as static *Pod*. So you must execute this command in the *Pod*
- Find where the `kube-apiserver` static *Pod* is configured
- Check if there are activated or deactivated admission plugins on your cluster
- Try to create `whoami` *Pod* and *Service* in *Namespace* `whoami` with the descriptor `Lab5/admission/whoami-pod.yaml` but don't create the *Namespace*!

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: whoami
  namespace: whoami
spec:
  selector:
    matchLabels:
      app: whoami
  replicas: 3
  template:
    metadata:
      labels:
        app: whoami
    spec:
      containers:
      - name: whoami
        image: containous/whoami
        ports:
        - containerPort: 80
---
kind: Service
apiVersion: v1
metadata:
  name: whoami
  namespace: whoami
spec:
  selector:
    name: whoami
  ports:
```

```
  - port: 80
    targetPort: 80
```

- You should get an error:

```
Error from server (NotFound): namespaces "whoami" not found
```

- Add the admission plugin `NamespaceAutoProvision`
  - On `node-0`, edit `/etc/kubernetes/manifests/kube-apiserver.yaml`
  - To the line `- --enabled-admission-plugins=...` add `NamespaceAutoProvision`
- Wait for the `kube-apiserver` to be available again
- Try to create `whoami` *Pod* and *Service* again
  - The *Namespace* should be created automatically
- Delete the `whoami` *Namespace*: `kubectl delete ns whoami`
  - It might take a while as the *Namespace* resources are also deleted

5.3: LimitRanges

- Create a new *Namespace* `limitrange`
- Create a *LimitRange* from `Lab5/limitrange/limitrange-1.yaml`

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-limit-range
  namespace: limitrange
spec:
  limits:
  - default:
      memory: 64Mi
      cpu: 500m
    defaultRequest:
      memory: 32Mi
      cpu: 500m
    type: Container
```

- Create a new *Pod* from `Lab5/limitrange/whoami-pod.yaml`
- Check the newly created *Pod* `limits` / `requests`: `kubectl -n limitrange describe pod whoami`
- Create the *LimitRange* from `Lab5/limitrange/limitrange-2.yaml`
- Apply `Lab5/limitrange/shell-pod.yaml`

- What happens? Why?

- Modify *LimitRange* `Lab5/limitrange/limitrange-2.yaml` to make creation of `Lab5/limitrange/shell-pod.yaml` possible

- Delete *Namespace* `limitrange`

5.4: Quotas

- Create a new *Namespace* `quotas`

- In this *Namespace* create the *ResourceQuota* `Lab5/quotas/quotas.yaml`

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pods-high
  namespace: quotas
spec:
  hard:
    cpu: "4"
    memory: 2Gi
    pods: "4"
```

- Create the *Deployment* `Lab5/quotas/whoami-deploy.yaml`

- Check that the deployment is ok: `kubectl -n quotas get pods`

- Scale the *Deployment*: `kubectl -n quotas scale deploy whoami --replicas=5`

- Check the *Pods* creation: `kubectl -n quotas get pods`

- What happens? Why?

- Check *Deployment* `whoami` : `kubectl -n quotas describe deploy whoami`

- Check *ReplicaSet* `whoami` : `kubectl -n quotas describe rs -l name=whoami`

- Modify *ResourceQuota* to allow scaling of *Deployment* `whoami` to 6

- Delete *Namespace* `quotas`

5.5: Security context

- Create a *Pod* from `Lab5/security/shell-security-pod.yaml`

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: shell-security
  labels:
    name: shell-security
```

```yaml
spec:
  securityContext:
    runAsUser: 37
    supplementalGroups: [11, 18, 19, 100]
  containers:
  - name: shell-security
    image: busybox
    command: ["sh", "-c", "sleep 3600"]
```

- Go into the created *Pod*: `kubectl exec -it shell-security sh`
- Check current user with: `whoami`
- Check that the id matches the one in `/etc/passwd`
- Check current groups with: `groups`
- Check that the ids match the ones in `/etc/group`
- Exit the *Pod*
- Delete it, and modify the manifest so that new files are created with `users` group ownership
- Create a file with: `touch /tmp/myfile`
- Check file ownership: `ls -al /tmp/myfile`
- Exit from the *Pod* and delete it

5.6: PodSecurityPolicy (Optional)

- Modify `kube-apiserver` configuration to add the `PodSecurityPolicy` admission plugin.
  - On `node-0`, edit `/etc/kubernetes/manifests/kube-apiserver.yaml`
  - To the line `- --enabled-admission-plugins=...` add `PodSecurityPolicy`
- Wait for the `kube-apiserver` to be available again
- Create *Namespace* `psp`
- In this *Namespace*, create *PodSecurityPolicy* `Lab5/psp/restricted-psp.yaml`

```yaml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: 'docker/default'
    apparmor.security.beta.kubernetes.io/allowedProfileNames: 'runtime/default'
    seccomp.security.alpha.kubernetes.io/defaultProfileName:  'docker/default'
    apparmor.security.beta.kubernetes.io/defaultProfileName:  'runtime/default'
spec:
  privileged: false
```

```yaml
  # Required to prevent escalations to root.
  allowPrivilegeEscalation: false
  # This is redundant with non-root + disallow privilege escalation,
  # but we can provide it for defense in depth.
  requiredDropCapabilities:
    - ALL
  # Allow core volume types.
  volumes:
    - 'configMap'
    - 'emptyDir'
    - 'projected'
    - 'secret'
    - 'downwardAPI'
    # Assume that persistentVolumes set up by the cluster admin are safe to use.
    - 'persistentVolumeClaim'
  hostNetwork: false
  hostIPC: false
  hostPID: false
  runAsUser:
    # Require the container to run without root privileges.
    rule: 'MustRunAsNonRoot'
  seLinux:
    # This policy assumes the nodes are using AppArmor rather than SELinux.
    rule: 'RunAsAny'
  supplementalGroups:
    rule: 'MustRunAs'
    ranges:
      # Forbid adding the root group.
      - min: 1
        max: 65535
  fsGroup:
    rule: 'MustRunAs'
    ranges:
      # Forbid adding the root group.
      - min: 1
        max: 65535
  readOnlyRootFilesystem: false
```

- Create the *Pod* `Lab5/psp/shell-pod.yaml`
- Check its status with: `kubectl get pods -w`
- What do you see?
- Check the error with `kubectl describe po shell`
- Modify the *Pod* manifest to make it work
- Check that it's ok with: `kubectl get pods -w`
- Check what's the `uid` of the *Pod* with `kubectl exec -it shell -- id -u`
- Delete *Namespace* `psp`

5.7: RBAC

- Create a new *Namespace* `rbac`
- In this *Namespace*:
  - Create the *Pod* `Lab5/rbac/shell-kubectl-pod.yaml`
  - Create the *Role* and *RoleBinding* `Lab5/rbac/role.yaml`

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: rbac
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
---
# This role binding allows "system:serviceaccount:rbac:default" to read pods in t
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: rbac
subjects:
- kind: User
  name: system:serviceaccount:rbac:default
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role #this must be Role or ClusterRole
  name: pod-reader # this must match the name of the Role or ClusterRole you wish
  apiGroup: rbac.authorization.k8s.io
```

- Go in the `shell-kubectl` *Pod*: `kubectl -n rbac exec -it shell-kubectl sh`
- Check files in `/run/secrets/kubernetes.io/serviceaccount`
  - `ca.crt` is the certificate used to trust the API server certificate
  - `namespace` is the name of the current *Namespace*
  - `token` contains a JWT (Json Web Token) with credentials for the *Pod* service account
- Check the content of the JWT with `base64 -d token`
  - The token is divided in three parts:
    - Header: algorithm and token type
    - Payload: json data
    - Token signature verification

- When you use `kubectl` inside this *Pod*, it will use these credentials
- Try to get *Pod* list from within the *Pod*: `kubectl get pods`
  - What happens? Why?
- Try to get *Pod* list of all namespaces from with the *Pod*: `kubectl get pods --all-namespaces`
  - What happens? Why?
- Add a *ClusterRole* and a *ClusterRoleBinding* to allow the service account `system:serviceaccount:rbac:default` listing *Pods* at cluster scope
- Check that it's working by listing *Pods* of all namespaces again
- Delete *Namespace* `rbac`

Lab 6

6.1: Custom Resource Definition

- Check and create a CRD (Custom Resource Definition) from `Lab6/crd/crd.yaml` :

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form: <plural>.<group>
  name: nodeapps.zenika.com
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: zenika.com
  # list of versions supported by this CustomResourceDefinition
  versions:
    - name: v1
      # Each version can be enabled/disabled by Served flag.
      served: true
      # One and only one version must be marked as the storage version.
      storage: true
  # either Namespaced or Cluster
  scope: Namespaced
  names:
    # plural name to be used in the URL: /apis/<group>/<version>/<plural>
    plural: nodeapps
    # singular name to be used as an alias on the CLI and for display
    singular: nodeapp
    # kind is normally the CamelCased singular type. Your resource manifests use
    kind: NodeApp
    # shortNames allow shorter string to match your resource on the CLI
    shortNames:
    - noa
  validation:
   # openAPIV3Schema is the schema for validating custom objects.
    openAPIV3Schema:
      properties:
        spec:
          properties:
            replicas:
              type: integer
              minimum: 1
              maximum: 10
            image:
              type: string
```

- Check that the CRD was created with `kubectl get crd`

- Check available api-resources on your cluster with `kubectl api-resources`

    - You'll see the previous declared characteristics: Name, ShortName, ApiGroup, Namespaced, Kind

- Create an instance of this CRD with `Lab6/crd/nodeapp.yaml`:

```
kind: NodeApp
apiVersion: "zenika.com/v1"
metadata:
  name: myapp
spec:
  replicas: 3
  image: myapp
```

- Check that the creation is Ok: `kubectl get noa`
    - You can also get it as `yaml` or `json` with:
        - `kubectl get noa -o yaml`
        - `kubectl get noa -o json`
- Delete the *NodeApp* instance: `kubectl delete noa myapp`
- Delete the *CRD*: `kubectl delete crd nodeapps.zenika.com`

6.2: Custom controller

- Check https://github.com/yaronha/kube-crd

6.3: Aggregation layer

- The aggregation layer is already active on your cluster
- Check this by finding the following flags on the `kube-apiserver` configuration:
    - `--proxy-client-cert-file=/etc/kubernetes/pki/front-proxy-client.crt`
    - `--proxy-client-key-file=/etc/kubernetes/pki/front-proxy-client.key`
    - `--requestheader-allowed-names=front-proxy-`
    - `--requestheader-client-ca-file=/etc/kubernetes/pki/front-proxy-ca.crt`
    - `--requestheader-extra-headers-prefix=X-Remote-Extra-`
    - `--requestheader-group-headers=X-Remote-Group`
    - `--requestheader-username-headers=X-Remote-User`
- We already deployed an extension API Server on our cluster in Lab3 with the `metrics-server`
- The new API Service was declared with this resource `Lab3/metrics-server/metrics-apiservice.yaml`

```yaml
apiVersion: apiregistration.k8s.io/v1beta1
kind: APIService
metadata:
  name: v1beta1.metrics.k8s.io
spec:
  service:
    name: metrics-server
    namespace: kube-system
  group: metrics.k8s.io
  version: v1beta1
  insecureSkipTLSVerify: true
  groupPriorityMinimum: 100
  versionPriority: 100
```

- Launch `kubectl api-resources | grep metrics`
  - You should see the two metrics exposed by the `metric-server` via the APIService

Lab 7