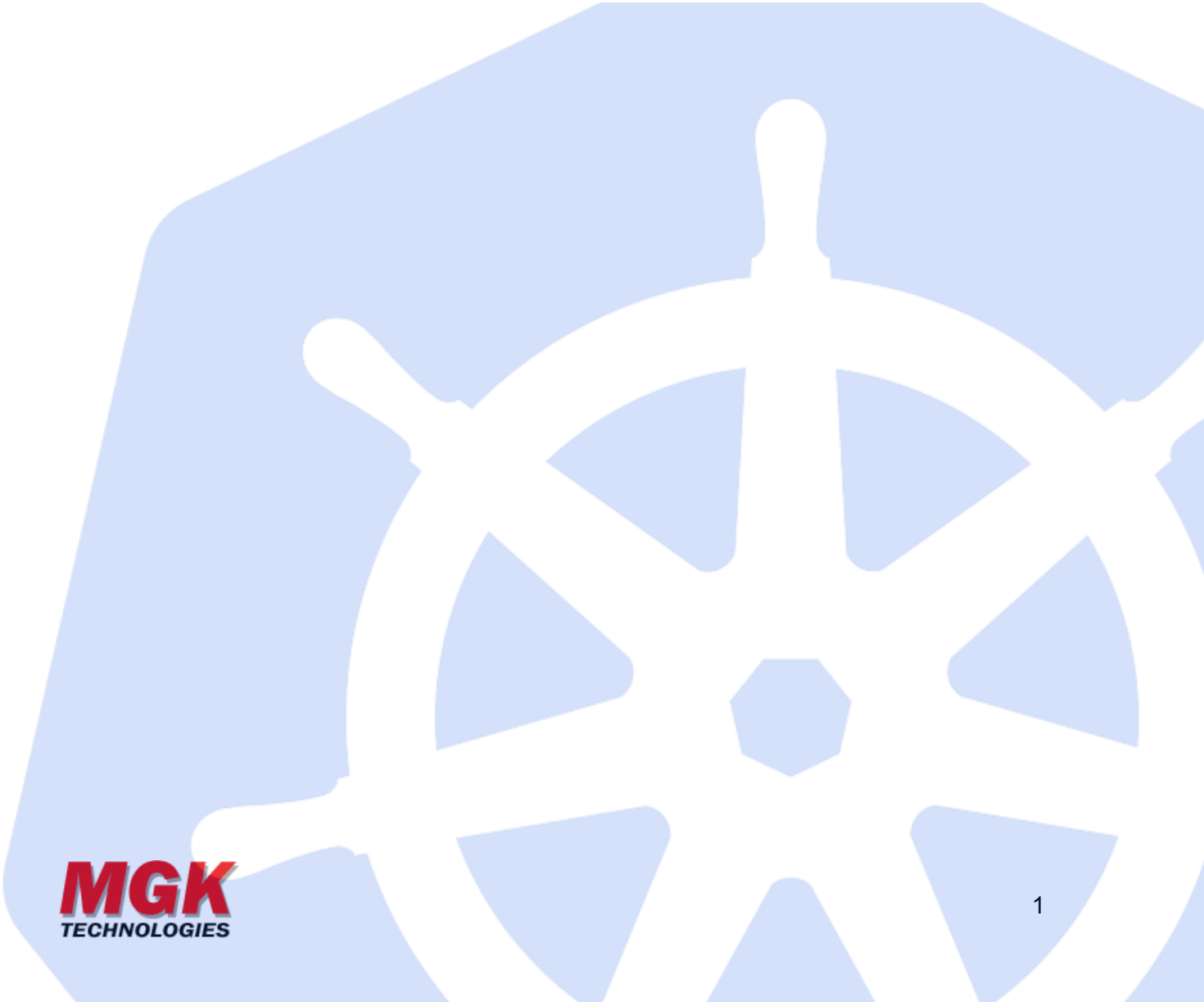
Deploy a Kubernetes cluster

Maxime Genet



This document will guide you in the installation of a kubernetes cluster (k8s) on top of a set of virtual machines. In the end, our cluster will provide a highly available MariaDB cluster and a persistent storage. The goal is to provide a highly available and flexible environment for applications.

Useful commands for Docker

Docker allows developers to launch their applications inside light units called containers. These applications run in a fully isolated environment, sharing resources from a unique underlying operating system. The lightweight structure of containers lead to highlight the microservices architecture.

A docker container is based on an underlying image made up of multiple layers configured in a build file called Dockerfile.

```
$ docker ps # List active containers

$ docker exec -it <container_name> bash #Run a bash session

$ docker run -it <image_name> bash  #Launch a container with a
bash session

$ docker build -t <image_name> . #Create a Docker image from
Dockerfile in current directory

$ docker push -t <repository> <image_name> #Export Docker image to
a remote repository (default: docker hub)

$ docker commit <container_name> <image_name> #Save container into
a docker image
```

Environment

The setup takes place through an environment of 4 Virtual machines created through Linux KVM. The first VM will serve as master in the k8s cluster. The third others will serve as workers. Workers are used to launch & run applications through pods. Each VM (guest/node) is based on Ubuntu 18.04. It is recommended to have a configuration of 8GB of RAM per node.

Our guests are configured with the following IPs:

```
master:     192.168.122.120
worker01:   192.168.122.121
worker02:   192.168.122.122
worker03:   192.168.122.123
```

Don't forget to update /etc/hosts & hostname configuration on each VM. The hosts file should be the same on each guest. You can also add these entries on host computer, as it would make easier SSH access to each guest.

Persistent volumes

We'll need to provide to our cluster a list of available disks that will be used especially for storing data that is read in the database. Each guest is provided with a 30 Gib disk. Data will be automatically synced & replicated on all of the disks. Then, the cluster would be able to failover when multiple nodes are down.

Volumes can easily be created through virt-manager. This is what you should get:

```
            master: no external storage
worker01: 30Gb external storage. Location: /dev/vdb
worker02: 30Gb external storage. Location: /dev/vdb
worker03: 30Gb external storage. Location: /dev/vdb
```

vdb is the name that will be used in this documentation. Your volumes might be called differently

Setup

All of the operations in the setup part have to be performed on all of the VMs.

Open the following ports::

```
-Kubernetes-
4149
10250
10255
10256
9099
6443
6783-6784 (TCP+UDP)
-GlusterFS-
2222
24007
24008
49152 - 49251
```

Load the following kernel modules :

```
dm_snapshot
dm_mirror
dm_thin_pool
```

Kubernetes cannot handle Swap for performance concerns. Disable it by :

```
 # swapoff -a
```

Comment all swap entries in fstab:

```
# sed -e '/swap/ s/^#*/#/' -i /etc/fstab
```

Install the necessary packages and add k8s GPG key to the system

```
$ sudo apt-get update \
  && sudo apt-get install -y apt-transport-https \
  curl \
  gnupg gnupg1 gnupg2 \
  && curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg
| sudo apt-key add -
```

Add k8s sources

```
$ echo "deb http://apt.kubernetes.io/ kubernetes-xenial main" \
  | sudo tee -a /etc/apt/sources.list.d/kubernetes.list \
  && sudo apt-get update
```

Install k8s, docker & Gluster packages

```
$ sudo apt-get update \
  && sudo apt-get install -y \
  docker.io \
  glusterfs-client \
  kubelet \
  kubeadm \
  kubernetes-cni
```

Enable docker start at boot on all of the VMs:

```
# systemctl enable docker.service
```

Prepare kubernetes cluster

Launch cluster initialization on your master VM :

```
$ sudo kubeadm init --pod-network-cidr=10.244.0.0/16
```

We need to configure a subnet range that will be needed for assign pod IPs, like 10.244.0.0 . Take note that this subnet has to be isolated from your current LAN.
If initialization is successful, you should get the kubeadm join command you have to run on each slave node

Note

Kubeadm listens by default on the first interface with a default gateway. You can explicit the network interface that k8s will be using by adding --apiserver-advertise-address <interface ip>

As indicated in the console, before joining other guests as nodes on the cluster, run the following commands on your master to get access through cluster administration (kubectl) :

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

Then, launch kubeadm join on other guests with the options as stated in the console.
Now, you can watch your nodes on the cluster using kubectl command on your master node:

```
$ kubectl get nodes
```

kubectl commands can only be run on master node.
This command allows you to see pods status in the cluster:

```
$ kubectl get pods
```

The command above doesn't allow you to see directly pods from k8s system namespace. To show the entire list :

```
$ kubectl get pods --all-namespaces
```

You need to pay attention to these kube-dns pods not in running state.

Kubernetes requires a network provider (CNI) to be installed. If not, some pods are in pending state, waiting for network features to be installed.

Several CNIs are available (Calico,Flannel,WeaveNet).
Let's install WeaveNet from its online yaml:

```
$ kubectl apply -f
"https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version |
base64 | tr -d '\n')"
```

It will create several pods on the cluster.
Now, all of your pods in kube-system namespace should be running. You can see their evolution using:

```
$ kubectl get pods --watch
```

Your cluster is now ready. Now, we need to prepare our persistent storage in order to get an operational database.

Run an application

In this part, we will run a sample deployment in order to become more familiar with the basic concepts of Kubernetes. Some useful tools like curl are included inside the container.
This deployment relies on a small ExpressJS server listening on port **8081**.

client-deploy.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-deployment
  labels:
      app: client
spec:
  replicas: 2
  selector:
```

```
      matchLabels:
      app: demo
  template:
      metadata:
      labels:
      app: demo
      spec:
      containers:
      - name: demo
      image: mushroommagnet/demo-pod:latest
      command: ["node", "app/server.js"]
      ports:
            - containerPort: 8081
```

Install this deployment:

```
$ kubectl create -f client-deploy.yaml
```

Now we have 2 “demo” pods running in parallel.

We need to add public visibility to our deployment, without having to consider which pod we need to run our requests on. As multiple pods are running simultaneously, we need to run an automated traffic routing to our pods in order to spread the workload.

Kubernetes provides different types of services for applications such as LoadBalancers,Nodeports or Ingress. In this deployment, we are using a ClusterIP service.

More information:
https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services-service-types

demo-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: demo-svc
spec:
  selector:
```

```
    app: demo
  ports:
    - protocol: TCP
    port: 80
    targetPort: 8081
```

A ClusterIP is the basic kind of Kubernetes service. Each ClusterIP is assigned with a stable DNS name and listens on a specific port, and targets a defined port on any pod of the deployment it is related to. This only works inside the cluster, which means any pod can emit requests to our service. This is useful when an application needs to get an access to a database.

Here, we want to redirect any request from port **80** of demo-svc, to port **8081** of any pod of our deployment demo.

```
$ kubectl create -f demo-svc.yaml
```

Even though we can emit requests from any pod in the cluster, it is easier to test our service from inside the deployment as the curl package is already installed on the demo pods.

Grab the name of one of the pods of the deployment:

```
$ kubectl get pods | grep demo
$ kubectl exec -it <pod_name> bash
```

Attach a bash session and launch a GET request from one of the pods shown above.

```
root@demo-deployment-6c7d968f5c-qnfgf:/# curl demo-svc
Hi Engineer !
root@demo-deployment-6c7d968f5c-qnfgf:/#
```

Persistent storage

As we are running docker containers, storage inside each pod is ephemeral. In a cluster, a distributed storage between nodes is useful when using a database. Fortunately, Kubernetes provides an API in order to provision persistent volumes when needed by pods. These volumes are directly mounted as a path inside the container. Managing volumes through k8s is based on several resources:

Persistent Volume (PV): Storage resource that is available in the cluster.

Persistent Volume Claim (PVC): A request for a storage with specific quantities (size).
A PVC can be seen as a front-end interface between the pod request (user) and the API in charge of binding the volume.

Storage Class: Controller in charge of binding a PVC with a PV. A Storage class defines a storage provisioner and volume characteristics (replicas) in order to provide a custom volume.

More information: https://kubernetes.io/docs/concepts/storage/persistent-volumes/

A large list of storage solutions is available for Kubernetes. In this documentation, we are going to focus on GlusterFS, thanks to its large documentation and options available.

Our deployment provides the Heketi framework, which is a RESTful interface for Glusterfs. Heketi's role is to expose an interface that helps managing volumes through the Glusterfs topology.

Download Glusterfs for kubernetes from Github:

```
$ git clone https://github.com/gluster/gluster-kubernetes.git
```

Navigate to deploy/ directory.

Firstly, we need to edit the topology.json.sample file as following :

topology.json:

```
{
  "clusters": [
          {
          "nodes": [
          {
          "node": {
          "hostnames": {
          "manage": [
          "worker03"
          ],
          "storage": [
          "192.168.122.123"
          ]
          },
          "zone": 1
          },
          "devices": [
          "/dev/vdb"
          ]
          },
          {
          "node": {
          "hostnames": {
          "manage": [
          "worker02"
          ],
          "storage": [
          "192.168.122.122"
          ]
          },
          "zone": 1
          },
          "devices": [
          "/dev/vdb"
          ]
          },
```

```
        {
        "node": {
        "hostnames": {
        "manage": [
        "worker01"
        ],
        "storage": [
        "192.168.122.121"
        ]
        },
        "zone": 1
        },
        "devices": [
        "/dev/vdb"
        ]
        }]
        }
  ]
}
```

GlusterFS deploys its cluster using the topology provided in this file. We need to define which nodes that will serve as volume bricks, and their devices that are available for glusterfs volumes.

Then, we can launch glusterfs deployment:

```
$ ./gk-deploy -gy
```

Note

If you need to make a full reinstall of glusterfs after a fail, you have to take into consideration that your persistent volumes have already been created and a simple delete of glusterfs pods with ./gk-deploy --abort will not be enough. Gluster volumes have to be clear of any data

Remove glusterfs pods:

```
$ ./gk-deploy -gy --abort
```

Then, on each glusterfs node (replace vdb with your persistent volume name):

```
# vgremove -ff $(sudo vgdisplay | grep -i "VG Name" | awk '{print
$3}')
# rm -rf /var/lib/heketi /etc/glusterfs /var/lib/glusterd
/var/log/glusterfs
# pvremove /dev/vdb
# wipefs -a -f /dev/vdb
```

You can now launch glusterfs deployment as stated above.

The installation will take several minutes, depending on how many storage devices are included in the topology.
After that, Glusterfs will deploy Heketi and a ClusterIP service in order to enable communication between the storage Class and Heketi interface.

When the deployment is done, go back in one of the pods in our demo deployment and verify that heketi is alive:

```
# curl heketi:8080/hello
Hello from Heketi !
```

We need to create a Storage Class that will provision persistent volumes to our applications, using Heketi API. Any persistent volume claim calls the storage class in order to get a persistent volume on the cluster.

Storage classes aren't complicated to configure. Here's an example for storage class called "data".

data.yaml:

```
apiVersion: storage.k8s.io/v1beta1
kind: StorageClass
metadata:
  name: data
provisioner: kubernetes.io/glusterfs
```

```
parameters:
  resturl: "http://heketi:8080"
  restuser: "user"
  restuserkey: "psd"
```

Install this class and mark it as you default storage class:

```
$ kubectl create -f data.yaml
$ kubectl patch storageclass data -p '{"metadata":
{"annotations":{"storageclass.kubernetes.io/is-default-class":"tru
e"}}}'
```

The storage Class is now available on the cluster:

```
$ kubectl get storageclass
```

Deploy a replicated MariaDB Cluster

This part shows you how to deploy a mariaDB cluster running master-slave replication. The goal is to provide highly available encrypted databases with automatic failover when a master crash is detected.

Get into the deployment

```
$ cd mysql/
```

Edit credentials

All credentials are stored in **secrets.yaml** . The values are base64 encoded. These secrets will be integrated into each pod as standard environment variables.

Default credentials:

Default mysql user: mgk
passwords: psd

If you're using echo when encoding a value, please make sure to disable the default trailing line :

```
$ echo password | base64 # wrong
$ echo -n password | base64 # right
```

The values are stored in the data section of the file:

```
data:
  root_password: cHNk
  slave_password: cHNk
  db_username: bWdr
  db_password: cHNk
  backup_password: cHNk
  encryption_key: ZGNmYmQ3MDIyNTc5N2I3NWE5MzQ4MmRhYTlmMjQ3ZjYK
```

Then, import your secret:

```
$ kubectl create -f secrets.yaml
```

Statefulset

Create a backup volume. This volume will be needed as the replication Master generates a backup when starting, so replication slaves can load the latest data and be synchronized with the master. The PVC is configured to use the storage class called "data"

```
$ kubectl create -f backup/pvc.yaml
```

Deploy the mariaDB configmap. A configmap is a configuration file available on the cluster that can be loaded by any pod of the deployment. This configmap contains several Mysql options needed to enable replication and encryption. The file will be stored in path /etc/mysql/conf.d/ of every container running mariaDB.

```
$ kubectl create -f cm.yaml
$ kubectl get cm # List available configmaps
```

Create a service for our database cluster. This is a headless service that enables direct access to our pods using their stable DNS name instead of using their IP. This service will not be useful from client side as we make the distinction between the readers (slaves) and the writer (master). Applications execute Read/Write access to a database using a unique host. In the next part, we will resolve this issue by splitting up reads from writes by providing the cluster a SQL proxy.

```
$ kubectl create -f svc.yaml
```

It's time to deploy mariaDB pods. The best way to deploy our cluster is to define a StatefulSet controller. A statefulSet is far more efficient than the basic deployment controller when it comes to pod order and stable pod identifiers.
The deployment generates 3 mariaDB instances and can be scaled up to 10 pods. Each pod contains a main mariaDB instance, with a "sidecar" or helper container acting as a script for replication.

In the end, we should get the following pod topology:

```
mysql-0
mysql-1
mysql-2
```

Each pod is provisioned with a unique persistent Volume.For example, mysql-0 is linked to data-mysql-0. These volumes store encrypted database data.

Pods are started one at a time by the stateful controller. Each pod runs a Init-container, which generates the configuration and loads/backups data for the main instance. Then, the pod becomes available when the two containers (mariaDB + sidecar) are started and running.

```
$ kubectl create -f statefulset.yaml
```

Cronjob backup

Database backups can be run at specific time using cronjob resource from Kubernetes. A cronjob config file is already included in the deployment. The execution time is set every 5 hour.

```
$ kubectl create -f backup/cronjob.yaml
```

We can trigger the cronjob manually:

```
$ kubectl create job --from=cronjob/mysql-backup backup-manual01
```

Deploy ProxySQL

Our current cluster cannot handle SQL writes on the slaves. We need to deploy a proxy that acts both as a load balancer and a router for writes. Applications will see the proxy as an interface between them and the cluster, without having to figure out which mariaDB instance can accept writes.

Firstly, we need to generate a configmap for ProxySQL. The configuration file defines general behavior, admin users, hostgroups for mysql servers and query rules to apply. As with mariaDB pods, sensitive information like passwords or usernames are only added when the pod is starting up by reading secret **mysql-secret**, so make sure this secret is already loaded in kubernetes.
The proxy can guess which instance accepts writes, based on the **read_only** mysql variable, which is off for the master.

Generate a configmap from the config file

```
$ kubectl create configmap proxysql-configmap
--from-file=proxy/proxysql.cnf
```

Launch the deployment:

```
$ kubectl create -f proxy/deploy.yaml
```

The file will also create a Nodeport service, which will make the proxy available on any node of the cluster.
Let's take a look at the specifications:

```
$ tail -n 20 proxy/deploy.yaml
```

The service is available on its DNS name: proxysql
Standard Mysql connections are using the default port: **3306** inside the cluster, which redirects to port **6033** of the proxySQL deployment (avoid port conflicts between mariaDB deployment and the proxy).
When connecting from a node, we need to specify port **30033**. This port is included in the default nodeport range **30000 - 32767**.

Admin access is available on port 6032 inside the cluster. From any node, it's available on port **30032**.

Deploy an Ingress application

A kubernetes Ingress is an object that allows an easier access to your application from outside the cluster. An Ingress acts as a router: it redirects the incoming traffic to a specific service by looking at the URL.

This job is done by an Ingress Controller. We chose to install Traefik, thanks to its documentation. Then, we'll deploy PhpMyAdmin as an example application.

The following steps come from Traefik user-guide for Kubernetes:
https://docs.traefik.io/user-guide/kubernetes/

Install Traefik

We need to allow Traefik to use the Kubernetes API. This is done by deploying a Role-based access control or RBAC :

```
$ kubectl create -f
https://raw.githubusercontent.com/containous/traefik/v1.7/examples
/k8s/traefik-rbac.yaml
```

Then, we can deploy Traefik pods by using a DaemonSet:

```
$ kubectl create -f
https://raw.githubusercontent.com/containous/traefik/v1.7/examples
/k8s/traefik-ds.yaml
```

Traefik pods are created in the kube-system namespace. When namespace option is not specified, we can only have a look of the default namespace.

Check the Ingress controller.

```
$ kubectl get pods -n kube-system
```

When the controller is up and running, we can expose the web-ui via an Ingress:

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

An Ingress acts as a router to a service, but it cannot redirect inbound traffic to the deployment alone. That's why we need to add a service.
The last thing to do is to add the ingress entry to the client /etc/hosts.
By default, pods are not scheduled on the master, so we cannot add the master's IP to the file.

For example, using worker01's IP:

Line added to /etc/hosts

```
192.168.122.121 admin.traefik
```

You should now be able to access the Traefik dashboard at admin.traefik

Example application: PhpMyAdmin

Create the deployment:

phpmyadmin.yaml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: phpmyadmin-deployment
```

```
  labels:
      app: phpmyadmin
spec:
  replicas: 1
  selector:
      matchLabels:
      app: phpmyadmin
  template:
      metadata:
      labels:
      app: phpmyadmin
      spec:
      containers:
      - name: phpmyadmin
            image: phpmyadmin/phpmyadmin
            ports:
            - containerPort: 80
            env:
            - name: PMA_HOST
            value: proxysql
            - name: PMA_PORT
            value: "3306"
            - name: MYSQL_ROOT_PASSWORD
            valueFrom:
                secretKeyRef:
                   name: mysql-secret
                   key: root_password
```

In a production environment, a good practice would be to enable TLS encryption over the Ingress.

More information:
https://kubernetes.io/fr/docs/concepts/services-networking/ingress/#tls

Now, let's create a ClusterIP service and its Ingress:

php-ingress.yaml

```
---
apiVersion: v1
kind: Service
metadata:
  name: phpmyadmin-service
spec:
  selector:
      app: phpmyadmin
  ports:
  - protocol: TCP
      port: 80
      targetPort: 80
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-phpmyadmin
  namespace: default
  annotations:
      kubernetes.io/ingress.class: traefik
spec:
  rules:
  - host: phpmyadmin.traefik
      http:
      paths:
      - path: /
      backend:
            serviceName: phpmyadmin-service
            servicePort: 80
```

Check the Ingress state from the Dashboard. The deployment should be available at phpmyadmin.traefik. Don't forget to add this entry in /etc/hosts.

Extra tools

Kubernetes Dashboard

```
$ kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0-beta
1/aio/deploy/recommended.yaml
```

```
$ kubectl proxy
```

The dashboard is now available on the master
In our configuration, we need to initiate a ssh tunnel from the main host in order to get an access from any web browser to the dashboard.

```
$ ssh -L 8001:127.0.0.1:8001 user@master
```

The Web UI is available at:
http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/

Ansible

Ansible is a powerful automation tool that can be useful when managing multiple machines.Like Kubernetes, Ansible can be configured via Yaml files called “playbooks’.

Installation procedure:

Use the PPA on Ubuntu:

```
$ sudo apt update
$ sudo apt install software-properties-common
$ sudo apt-add-repository --yes --update ppa:ansible/ansible
$ sudo apt install ansible python
```

Ansible communicates through ssh. Export your ssh public key to your hosts (~/.ssh/authorized_keys).

In order to set up ansible host groups, edit the file /etc/ansible/host.
Some examples are given in the commented section of the file.

```
[workers]
10.1.99.121
10.1.99.122
10.1.99.123
[master]
10.1.99.120 ansible_connection=local
```

If one node ssh user is different from the current master user, add this option next to the node's IP: **ansible_ssh_user=<username>**

Example playbook: Update nodes

Ansible owns a set of default actions such as updating or rebooting. As APT requires root permissions, this script gain root access on all nodes using the password provided in order to update apt packages.

update.yaml

```
- hosts: all
  vars:
      ansible_become_pass: <password>
  tasks:
      - name: Update and upgrade apt packages
      become: yes
      apt:
      upgrade: yes
      update_cache: yes
```

```
$ ansible-playbook update.yaml
```

Example playbook: Join a cluster

We assume here that the master node is already initialized. In the case of multiple worker nodes to join, doing the same command on every node will become a running score. The playbook below runs a shell task with arguments :

join.yaml

```
---
- name: Join nodes
  hosts: workers
  ignore_errors: yes
  become: true
  vars:
      ansible_become_pass: psd
      _token: "{{ token }}"
      _cert: "{{ cert }}"
  tasks:
      - name: Add nodes
      shell: kubeadm join 10.1.99.120:6443 --discovery-token {{
_token }} --discovery-token-ca-cert-hash {{ _cert }}
```

Replace <value> with the token and cert values from the output of the init command on the master node.

```
$ ansible-playbook join.yaml --extra-vars "token=<value>
cert=<value>"
```