Technische Universität München

Cloud Computing Course

WS - 2017

Exercise - 4

“Kubernetes: Container Orchestration”

Authors:
Prof. M. Gerndt,
Prof. S. Benedict,
Anshul Jindal

Table of Contents

I. Introduction

1.1 Purpose of this document

This document provides guidance and material to assist the relevant person, in understanding the various container orchestration engines. This includes following:

- Installation and configuration of Kubernetes
- App deployment using Kubernetes
- Run and deploy the application

1.2 Prerequisites

Following requirements are mandatory:

- Account on LRZ
- Previous exercises Completed

Following requirements are recommended (not mandatory):

- Basic knowledge of Linux console commands.

1.3 Background

The Docker platform and surrounding ecosystem contain many tools to manage the lifecycle of a container. Like, Docker Command Line Interface (CLI) supports the following container activities:

- Pulling an image from a repository.
- Running the container and optionally attaching a terminal to it.
- Committing the container to a new image.
- Uploading the image to the registry.
- Terminating a running container.

While the CLI meets the needs of managing one container on one host, it falls short when it comes to managing multiple containers deployed on multiple hosts. To go beyond the management of individual containers, we must turn to orchestration tools.

Orchestration tools extend lifecycle management capabilities to complex, multi-container workloads deployed on a cluster of machines. By abstracting the host infrastructure, orchestration tools allow users to treat the entire cluster as a single deployment target.

Baseline Features: The process of orchestration typically involves the automation of all aspects of application management from initial placement, scheduling and deployment to steady-state activities such as update, deployment, and health monitoring functions that support scaling and failover. These capabilities have come to characterize some of the core features offered by modern container orchestration tools.

Declarative Configuration: Orchestration tools provide an option for DevOps teams to declare the blueprint for an application workload and its configuration in a standard schema, using languages such as YAML or JSON. These definitions also carry crucial information about the repositories, networking (ports), storage (volumes) and logs that support the workload. This approach allows orchestration tools to apply the same configuration multiple times and always yield the same result on the target system. It also allows the tools to accept different configurations for the same application during the various stages of development, testing and production for different target environments.

Rules and Constraints: Workloads often have special policies or requirements for host placement, performance, and high availability. For example, it's pointless to provision the master and slave database container on the same host; it defeats the purpose. Similarly, it may be a good idea to place in-memory cache on the same host as the web server. Orchestration tools support mechanisms for defining the affinity and constraints of container placement.

Provisioning: Provisioning, or scheduling, deals with negotiating the placement of containers within the cluster and launching them. This process involves selecting an appropriate host based on the configuration. Apart from a container-provisioning API, orchestration tools will invoke the infrastructure APIs specific to the host environment.

Discovery: In a distributed deployment consisting of containers running on multiple hosts, container discovery becomes critical. Web servers need to dynamically discover the database servers, and load balancers need to discover and register web servers. Orchestration tools provide, or expect, a distributed key-value store, a lightweight DNS, or some other mechanism to enable the discovery of containers.

Health Monitoring: Since orchestration tools are aware of the desired configuration of the system, they are uniquely able to track and monitor the health of the system's containers and hosts. In the event of host failure, the tools can relocate the container. Similarly, when a container crashes, orchestration tools can launch a replacement. Orchestration tools ensure that the deployment always matches the desired state declared by the developer or operator.

1.3.1 Kubernetes

The name **Kubernetes** originates from Greek, meaning "helmsman" or "pilot", and is the root of "governor" and "cybernetic". **K8s** is an abbreviation derived by replacing the 8 letters "ubernete" with 8. The Kubernetes project was started by Google in 2014

It is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts, providing container-centric infrastructure. With Kubernetes, you can:

- Deploy your applications quickly and predictably.
- Scale your applications on the fly.
- Seamlessly roll out new features.
- Optimize use of your hardware by using only the resources you need.

Kubernetes' architecture is based on a master server with multiple minions. The command line tool, called **kubectl**, connects to the API endpoint of the master to manage and orchestrate the minions. Below is the definition of each component that runs within the Kubernetes environment:

- **Master:** The server that runs the Kubernetes management processes, including the API service, replication controller and scheduler.
- **Kube-apiserver:** kube-apiserver exposes the Kubernetes API; it is the front-end for the Kubernetes control plane. It is designed to scale horizontally (i.e., one scales it by running more of them– Building High-Availability Clusters).
- **etcd:** etcd is used as Kubernetes' backing store. All cluster data is stored here. Proper administration of a Kubernetes cluster includes a backup plan for etcd's data
- **Kube-controller-manager:** kube-controller-manager is a binary that runs controllers, which are the background threads that handle routine tasks in the cluster. Logically, each controller is a separate process, but to reduce the number of moving pieces in the system, they are all compiled into a single binary and run in a single process. These controllers include:
    - **Node Controller:** Responsible for noticing & responding when nodes go down.
    - **Replication Controller:** Responsible for maintaining the correct number of pods for every replication controller object in the system.
    - **Endpoints Controller:** Populates the Endpoints object (i.e., join Services & Pods).
    - **Service Account & Token Controllers:** Create default accounts and API access tokens for new namespaces.
    - … and others

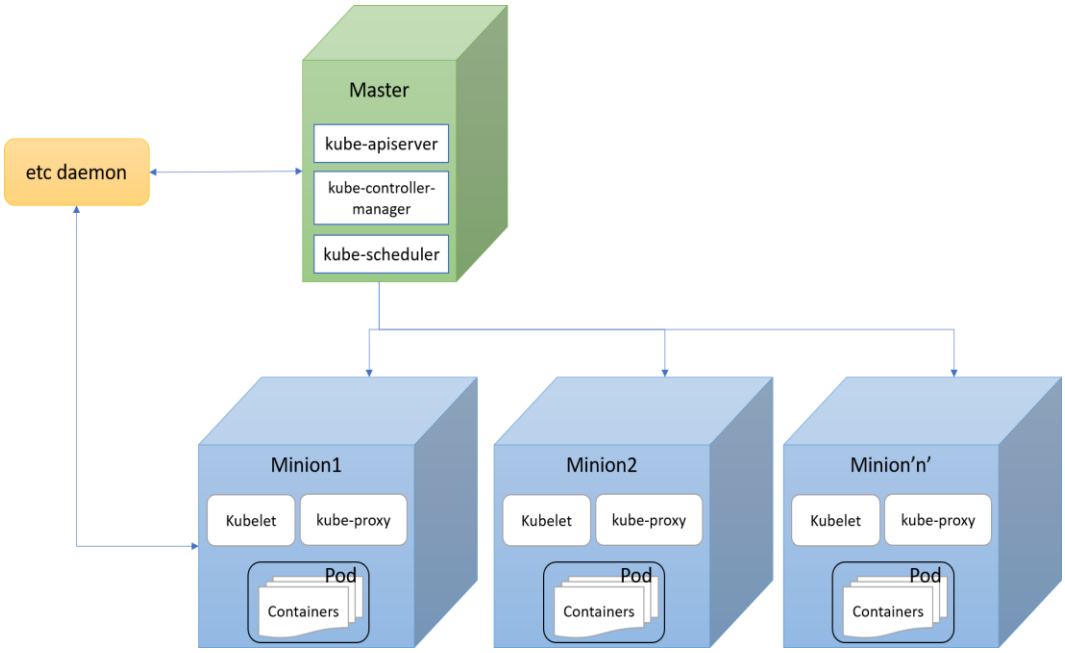


Figure 1: Kubernetes Architecture

- **kube-scheduler:** kube-scheduler watches newly created pods that have no node assigned, and selects a node for them to run on.
- **Minion:** The host that runs the Kubelet service and the Docker Engine. Minions receive commands from the master.
- **Kubelet:** The node-level manager in Kubernetes; it runs on a minion.
- **Kube-proxy:** kube-proxy enables the Kubernetes service abstraction by maintaining network rules on the host and performing connection forwarding
- **Pod:** The collection of containers deployed on the same minion.

The service definition, along with the rules and constraints, is described in a JSON file.

For service discovery, Kubernetes provides a stable IP address and DNS name that corresponds to a dynamic set of pods. When a container running in a Kubernetes pod connects to this address, the connection is forwarded by a local agent (**called the kube-proxy**) running on the source machine to one of the corresponding backend containers.

Since Kubernetes operates at the application level rather than at just the hardware level, it provides some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, logging, monitoring, etc. However, Kubernetes is not monolithic, and these default solutions are optional and pluggable.

Additionally, Kubernetes is not a mere “orchestration system”; it eliminates the need for orchestration. The technical definition of “orchestration” is execution of a defined workflow: do A, then B, then C. In contrast, Kubernetes is comprised of a set of independent, composable control processes that continuously drive current state towards the provided desired state. It shouldn’t matter how you get from A to C: make it so. Centralized control is also not required; the approach is more akin to “choreography”. This results in a system that is easier to use and more powerful, robust, resilient, and extensible.

II. Exercise Steps

2.1 Installation and Setup

1. Install the docker from official repository as mentioned in the 2nd exercise using the following commands:

```
sudo apt-get remove docker docker-engine docker.io
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
sudo apt-get update
sudo apt-get install docker-ce
```

2. Once the docker is installed start it by running the following command:

```
sudo systemctl start docker.service
```

3. We will be installing Kubernetes on Linux machine with the help of **kubeadm**. Firstly, install the necessary packages required for running Kubernetes.

```
sudo apt-get update && apt-get install -y apt-transport-https
sudo curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
sudo cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
sudo apt-get update
sudo apt-get install -y kubelet kubeadm kubernetes-cni
```

you can test the installation by running the kubectl command, it will display something like this:

```
kubectl controls the Kubernetes cluster manager.

Find more information at https://github.com/kubernetes/kubernetes.

Basic Commands (Beginner):
  create         Create a resource by filename or stdin
  expose         Take a replication controller, service, deployment or pod and expose it as a new Kubernetes Service
  run            Run a particular image on the cluster
  set            Set specific features on objects

Basic Commands (Intermediate):
  get            Display one or many resources
  explain        Documentation of resources
  edit           Edit a resource on the server
  delete         Delete resources by filenames, stdin, resources and names, or by resources and label selector

Deploy Commands:
  rollout        Manage a deployment rollout
  rolling-update Perform a rolling update of the given ReplicationController
  scale          Set a new size for a Deployment, ReplicaSet, Replication Controller, or Job
  autoscale      Auto-scale a Deployment, ReplicaSet, or ReplicationController

Cluster Management Commands:
  certificate    Modify certificate resources.
  cluster-info   Display cluster info
  top            Display Resource (CPU/Memory/Storage) usage.
  cordon         Mark node as unschedulable
  uncordon       Mark node as schedulable
  drain          Drain node in preparation for maintenance
  taint          Update the taints on one or more nodes
```

Note:

- **kubeadm** works for only the Ubuntu versions after 16.04.
- If you already have docker installed on your machine, then you can skip the docker installation command.
- Recommend you have sufficient diskspace before proceeding further.
- Run the following command on the VM to enable network through all ports

iptables -I INPUT -j ACCEPT

2.2 Configuring Kubernetes

2.2.1 Initializing the Master Node

The master is the machine where the "control plane" components run, including etcd (the cluster database) and the API server (which the kubectl CLI communicates with). These components run in pods started by **Kubelet.**

To initialize the master run the following command on one of the machines (**can only be run as root, because we set up Kubernetes on a node which will effectively act as root on the node**):

```
kubeadm init
```

Note:
This will autodetect the network interface to advertise the master on as the interface with the default gateway. If you want to use a different interface, specify **--api-advertise-addresses <ip-address>** argument to kubeadm init.

Output of the above command will look something like this:

```
[etcd] Wrote Static Pod manifest for a local etcd instance to "/etc/kubernetes/manifests/et
[init] Waiting for the kubelet to boot up the control plane as Static Pods from directory "
[init] This might take a minute or longer if the control plane images have to be pulled.
[apiclient] All control plane components are healthy after 48.504239 seconds
[uploadconfig] Storing the configuration used in ConfigMap "kubeadm-config" in the "kube-sy
[markmaster] Will mark node vm-10-155-208-213 as master by adding a label and a taint
[markmaster] Master vm-10-155-208-213 tainted and labelled with key/value: node-role.kubern
[bootstraptoken] Using token: e1e847.e8e8b3eda94f8587
[bootstraptoken] Configured RBAC rules to allow Node Bootstrap tokens to post CSRs in order
[bootstraptoken] Configured RBAC rules to allow the csrapprover controller automatically ap
[bootstraptoken] Configured RBAC rules to allow certificate rotation for all node client ce
[bootstraptoken] Creating the "cluster-info" ConfigMap in the "kube-public" namespace
[addons] Applied essential addon: kube-dns
[addons] Applied essential addon: kube-proxy

Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

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

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

You can now join any number of machines by running the following on each node
as root:

  kubeadm join --token e1e847.e8e8b3eda94f8587 10.155.208.213:6443 --discovery-token-ca-cer
c307fa17285eaeedaa556d4
```

Make a record of the kubeadm join command that kubeadm init outputs. This is required to add nodes to the kubernetes cluster.

Note:

- **Because of the limitation of number of VMs on LRZ, the exercise will not involve attaching of slave instances to Master.**

Before going forward, you should create a new user (as described in exercise-1), add it to sudoers and run the following commands on it:

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

Now you can use that user to create cluster. You can verify whether everything is running or not by running the following command from the **normal user account**:

kubectl get nodes

```
ansjin@vm-10-155-208-213:/$ sudo kubectl get nodes
NAME                   STATUS     ROLES     AGE       VERSION
vm-10-155-208-213   NotReady   master    14m       v1.9.0
ansjin@vm-10-155-208-213:/$
```

2.2.2 Installing the Pod Network

You must install a pod network[1] add-on so that your pods can communicate with each other.

It is necessary to do this before you try to deploy any applications to your cluster, and before kube-dns will start up. Note also that kubeadm only supports CNI based networks and therefore kubenet based networks will not work. Several projects provide Kubernetes pod networks using CNI, some of which also support Network Policy. See the add-ons page for a complete list of available network add-ons.

1. You can install a pod network add-on with the following command from the normal-user account:

kubectl apply -f <add-on.yaml>

We will be installing weave net, which provides networking and network policy.

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

```
ansjin@vm-10-155-208-213:/$ kubectl apply -f "https://cloud.weave.works/k8s/net?
serviceaccount "weave-net" created
clusterrole "weave-net" created
clusterrolebinding "weave-net" created
role "weave-net" created
rolebinding "weave-net" created
daemonset "weave-net" created
ansjin@vm-10-155-208-213:/$
```

NOTE:

[1] Kubernetes assumes that pods can communicate with other pods, regardless of which host they land on. We give every pod its own IP address, so you do not need to explicitly create links between pods and you almost never need to deal with mapping container ports to host ports. This creates a clean, backwards-compatible model where pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration. For more detail refer here.

- Please check here to confirm the updated version of the URL.
- You can install **only one** pod network per cluster.

2. To check the status of pods run the following command:

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

```
ansjin@vm-10-155-208-213:/$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                        READY     STATUS    RESTARTS   AGE
kube-system   etcd-vm-10-155-208-213                      1/1       Running   0          35s
kube-system   kube-apiserver-vm-10-155-208-213            1/1       Running   0          39s
kube-system   kube-controller-manager-vm-10-155-208-213   1/1       Running   0          39s
kube-system   kube-dns-6f4fd4bdf-t67wg                    0/3       Pending   0          1m
kube-system   kube-proxy-jz4s5                            1/1       Running   0          1m
kube-system   kube-scheduler-vm-10-155-208-213            1/1       Running   0          34s
kube-system   weave-net-7w48x                             2/2       Running   0          42s
```

3. To access the dashboard of kubernetes, firstly access control permissions need to be provided. These access permissions can be studied here. But for the exercise we will be providing full admin access control by using the **`dashboard-admin.yaml`** `file (provided in the source code)` to access dashboard. This file looks like this:

```yaml
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: kubernetes-dashboard
  labels:
    k8s-app: kubernetes-dashboard
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: kubernetes-dashboard
  namespace: kube-system
```

Now, run the following command from inside the source directory:

```
kubectl create -f dashboard-admin.yaml
```

4. You can now install the dashboard add on by running the following command:

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/master/src/deploy/recommended/kubernetes-dashboard.yaml
```

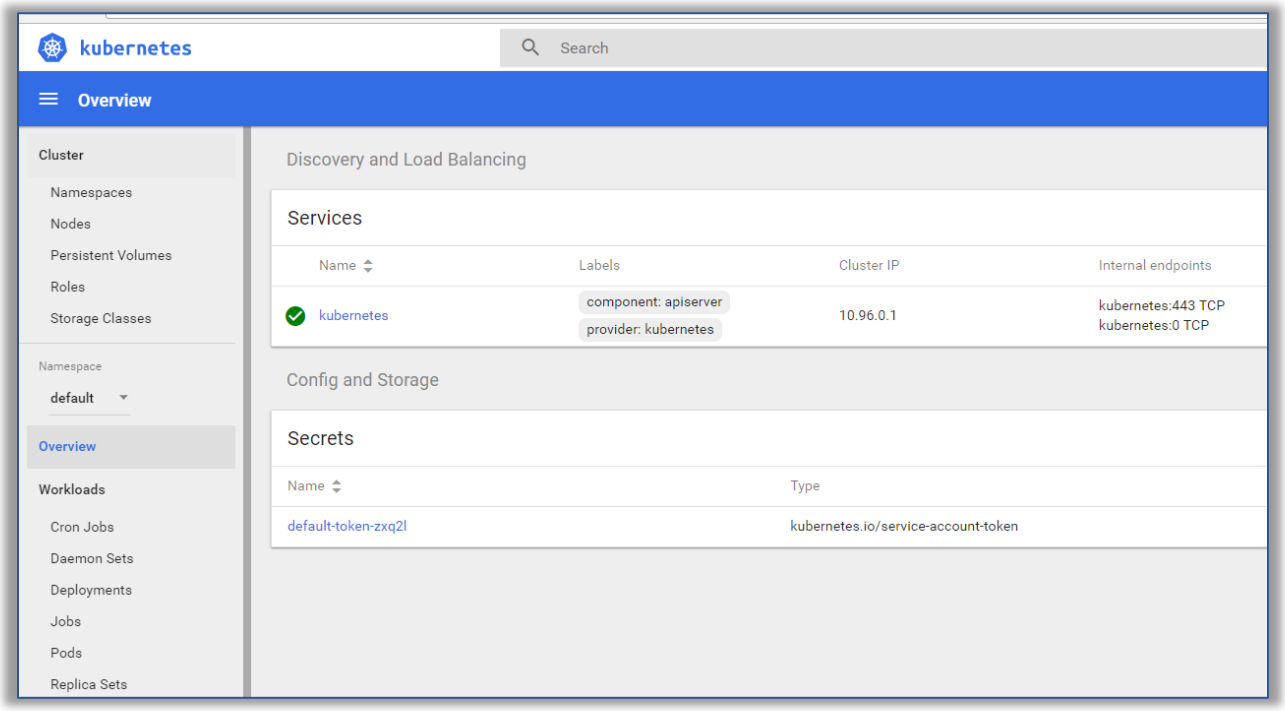
The easiest way to access Dashboard is to use kubectl. Run the following command:

```
sudo kubectl proxy --address='0.0.0.0' --port=8001 --accept-hosts='^*$'&
```

kubectl will handle authentication with apiserver and make Dashboard available at

http://YOUR_VM_IP:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/.

Click on the skip as we are have provided admin access control.



5. All the status of the pods can be checked from this command:

kubectl get pods --all-namespaces

```
ansjin@vm-10-155-208-213:/$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                          READY     STATUS    RESTARTS   AGE
kube-system   etcd-vm-10-155-208-213                        1/1       Running   0          3m
kube-system   kube-apiserver-vm-10-155-208-213              1/1       Running   0          3m
kube-system   kube-controller-manager-vm-10-155-208-213     1/1       Running   0          3m
kube-system   kube-dns-6f4fd4bdf-vfhmd                      2/3       Running   1          4m
kube-system   kube-proxy-pmvj9                              1/1       Running   0          4m
kube-system   kube-scheduler-vm-10-155-208-213              1/1       Running   0          3m
kube-system   kubernetes-dashboard-7b7b5cd79b-5ms9b         1/1       Running   0          26s
kube-system   weave-net-d9tbr                               2/2       Running   0          2m
ansjin@vm-10-155-208-213:/$
```

NOTE:

You may have trouble in the configuration if you see any of the three statuses **RunContainerError** and **CrashLoopBackOff** and **ContainerCreating.** These are very common. To help diagnose what happened, you can use the following command to check what is in the logs:

kubectl describe -n kube-system po {YOUR_POD_NAME}

You can then search Google for the error messages, which may help you find a solution. And once the kube-dns pod is up and running, you can continue to next steps.

2.2.3 Joining your Nodes

1. By default, your cluster will not schedule pods on the master for security reasons. If you want to be able to schedule pods on the master, e.g. a single-machine Kubernetes cluster for development, run the following command on master:

   kubectl taint nodes --all node-role.kubernetes.io/master-

2. The nodes are where your workloads (containers and pods, etc.) run. If you want to add any new machines as nodes to your cluster, for each machine:
   - Install the kubeadm, kubectl, docker.io, Kubelet and kubernetes-cni as explained in here
   - SSH to that machine, become root (e.g. sudo su -) and
   - run the command that was output by **kubeadm init**.
     Run the following command on the master node, you will see the nodes attached to the master.

     kubectl get nodes

2.3 Running your containerized image

Now we have our cluster, the next task is to deploy our application on it.

1. We already have images of our microservices from the last exercise. We will be using those services and deploy using kubernetes. To deploy the service run the following command:

```
kubectl run SERVICENAME --image=docker.io/IMAGEADDRESS --port=PORT
```

Running our microservices will look like this:

```
ansjin@vm-10-155-208-213:/$ kubectl run hello-world-service --image=docker.io/ansjin/microservice:hello --port=
9001
deployment "hello-world-service" created
ansjin@vm-10-155-208-213:/$ kubectl run product-descp-service --image=docker.io/ansjin/microservice:productdesc
p --port=9002
deployment "product-descp-service" created
ansjin@vm-10-155-208-213:/$ kubectl run product-price-service --image=docker.io/ansjin/microservice:productpric
e --port=9003
deployment "product-price-service" created
ansjin@vm-10-155-208-213:/$ kubectl run server --image=docker.io/ansjin/microservice:server --port=8080
deployment "server" created
ansjin@vm-10-155-208-213:/$
```

You could also deploy by creating yaml file for each of the service, but the above process is the easiest one.

2. You can check the status of all the deployments by running the command:

```
kubectl get deployments --all-namespaces
```

```
ansjin@vm-10-155-208-213:/$ kubectl get deployments --all-namespaces
NAMESPACE     NAME                   DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
default       hello-world-service    1         1         1            1           4m
default       product-descp-service  1         1         1            1           4m
default       product-price-service  1         1         1            1           4m
default       server                 1         1         1            1           3m
kube-system   kube-dns               1         1         1            1           1h
kube-system   kubernetes-dashboard   1         1         1            1           1h
```

3. You can also check the information about the deployment with the following command:

```
kubectl describe deployment <name of the deployment>
```

Here it is:

```
kubectl describe deployment hello-world-service
```

```
ansjin@vm-10-155-208-213:/$ kubectl describe deployment hello-world-service
Name:                   hello-world-service
Namespace:              default
CreationTimestamp:      Sun, 17 Dec 2017 19:05:40 +0100
Labels:                 run=hello-world-service
Annotations:            deployment.kubernetes.io/revision=1
Selector:               run=hello-world-service
Replicas:               1 desired | 1 updated | 1 total | 1 available | 0 unavaila
ble
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
Pod Template:
  Labels:  run=hello-world-service
  Containers:
   hello-world-service:
    Image:        docker.io/ansjin/microservice:hello
    Port:         9001/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type            Status  Reason
  ----            ------  ------
  Available       True    MinimumReplicasAvailable
```

4. As all the services are running in different pods so we need to create services for each of them to complete the interaction. This can be done by following command:

   kubectl expose deployment <deploymentName> --port=<portNumber>

   By default, the type is Cluster IP, so the service will be exposed inside the Cluster.

   kubectl expose deployment hello-world-service --port=9001

```
ansjin@vm-10-155-208-213:/$ kubectl expose deployment  product-descp-service --port=9002
service "product-descp-service" exposed
ansjin@vm-10-155-208-213:/$ kubectl get svc --all-namespaces
NAMESPACE     NAME                   TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
default       hello-world-service    ClusterIP      10.101.168.134   <none>        9001/TCP         3m
default       kubernetes             ClusterIP      10.96.0.1        <none>        443/TCP          3h
default       product-descp-service  ClusterIP      10.99.204.194    <none>        9002/TCP         5s
default       server                 LoadBalancer   10.104.32.226    <pending>     8080:32400/TCP   2h
kube-system   kube-dns               ClusterIP      10.96.0.10       <none>        53/UDP,53/TCP    3h
kube-system   kubernetes-dashboard   ClusterIP      10.102.132.97    <none>        443/TCP          3h
ansjin@vm-10-155-208-213:/$ kubectl expose deployment  product-price-service --port=9003
```

5. We want the server service to be accessed from outside the Cluster. So that need to be exposed to Internet. It can exposed as a service to the internet with the following command:

   kubectl expose deployment <deploymentName> --port=<portNumber> --type=LoadBalancer

   Here we are using type as loadbalncer, so that kubernetes automatically distributes load on the service between its slave nodes.

Here it is:

kubectl expose deployment server --port=8080 --target-port=8080 --type=LoadBalancer

you can view the services by running the command

kubectl get services –all-namespaces

```
ansjin@vm-10-155-208-213:/$ kubectl get svc --all-namespaces
NAMESPACE     NAME                   TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)
default       hello-world-service    ClusterIP      10.101.168.134   <none>        9001/TCP
default       kubernetes             ClusterIP      10.96.0.1        <none>        443/TCP
default       product-descp-service   ClusterIP      10.99.204.194    <none>        9002/TCP
default       product-price-service   ClusterIP      10.105.146.127   <none>        9003/TCP
default       server                 LoadBalancer   10.104.32.226    <pending>     8080:32400/TCP
kube-system   kube-dns               ClusterIP      10.96.0.10       <none>        53/UDP,53/TCP
kube-system   kubernetes-dashboard   ClusterIP      10.102.132.97    <none>        443/TCP
```

Here the external-IP is your **VM public IP** and the port number is **32400**.
If you visit the address http://VM_IP:PORTNUMBER/
/exercises/exercise3?name=CCS&productId=3 you will get the following page:


```
{"hello":"Welcome CCS","product_id":"3","productURL":"http://meetup.com/neque/vestibulum/eget/vulputate.png?
malesuada=cras&in=in&imperdiet=purus&et=eu&commodo=magna&vulputate=vulputate&justo=luctus&in=cum&blandit=sociis&ul
us&interdum=vestibulum&mauris=sagittis&non=sapien&ligula=cum&pellentesque=sociis&ultrices=natoque&phasellus=penati
drerit=vestibulum&at=rutrum&vulputate=rutrum&vitae=neque&nisl=aenean&aenean=auctor&lectus=gravida&pellentesque=sem
=ut&massa=odio&volutpat=cras&convallis=mi&morbi=pede&odio=malesuada&odio=in&elementum=imperdiet&eu=et&interdum=com
```


6. You can scale up or down your deployment by running the command:

   kubectl scale deployment <deployment_Name> --replicas=<replicaNumber>

   Here it is:

   kubectl scale deployment server --replicas=2

```
ansjin@vm-10-155-208-213:/$ kubectl get deployment server
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
server    2         2         2            2           2h
```

7. To update and deploy, you can run the following command:

   kubectl edit deployment <deployment_Name>

   Here it is:

   kubectl edit deployment server

```
"
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  creationTimestamp: 2017-12-17T18:06:27Z
  generation: 2
  labels:
    run: server
  name: server
  namespace: default
  resourceVersion: "20365"
  selfLink: /apis/extensions/v1beta1/namespaces/default/d
  uid: 0010afc8-e355-11e7-9f7d-020000f80025
spec:
  replicas: 3
  selector:
    matchLabels:
      run: server
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        run: server
```

Here I modified the number of replicas. After saving, it automatically creates 3 replicas

```
ansjin@vm-10-155-208-213:/$ kubectl edit deployment server
deployment "server" edited
ansjin@vm-10-155-208-213:/$ kubectl get deployment server
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
server    3         3         3            3           2h
```

8. To delete the service and deployment you can run the following command:

   kubectl delete service,deployment <deployment_Name>

9. Kubernetes offer many more possibilities:
   - creating namespaces to isolate users of a cluster, defining quotas, …
   - creating a secure proxy from your localhost to a pod

     kubectl port-forward POD_NAME PORT:LOCAL_PORT

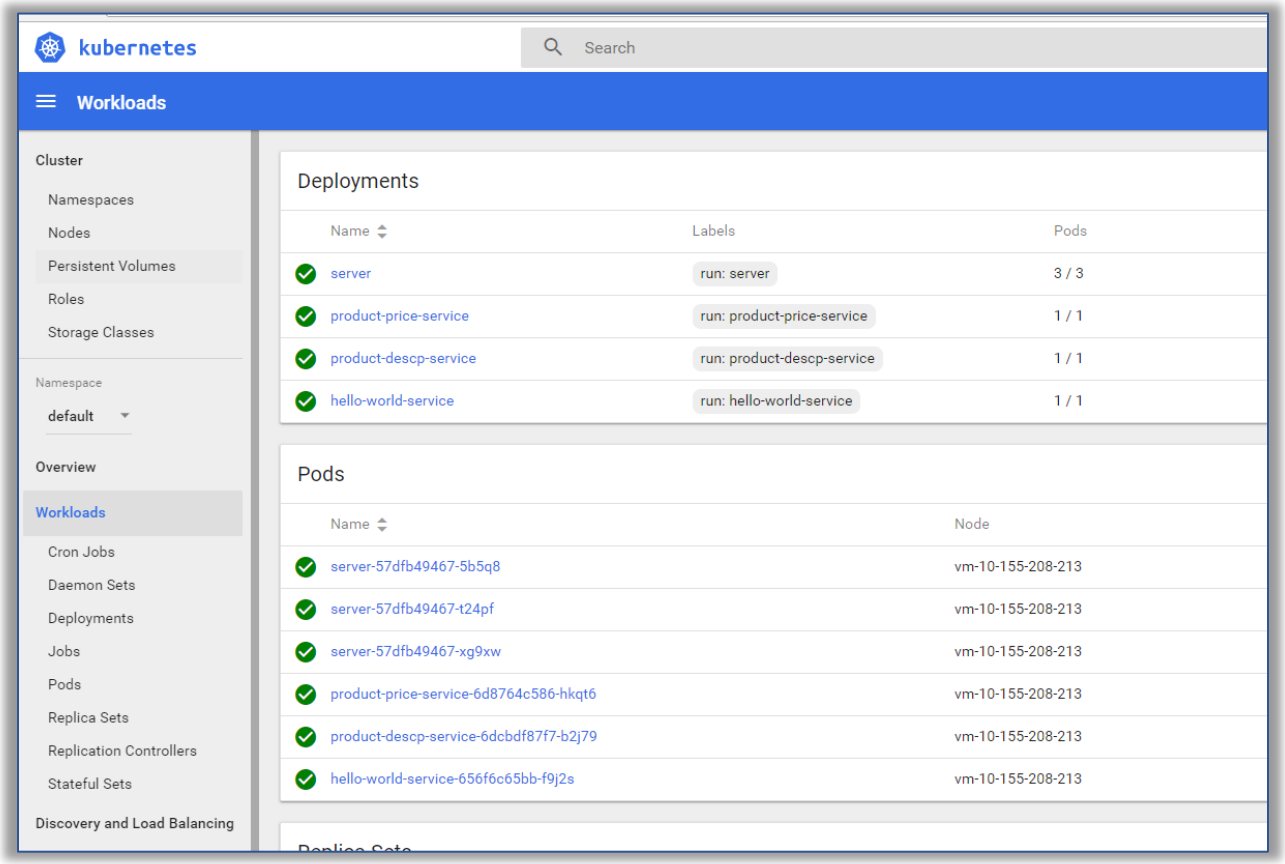
   - auto scaling

Vary the number of pods between 2 to 10 based upon the target average CPU utilization at 80%

kubectl autoscale deployment DEPLOYMENT_NAME --min=2 --max=10 --cpu-percent=80

- For using Autoscaling, Heapster need to be deployed(will be discussed in the next exercise)
- You can also update the deployment by creating/editing a new YAML file.

10. You can also do everything from the user-interface.



2.4 Tear Down of your nodes

To undo what kubeadm did, you should first drain the node and make sure that the node is empty before shutting it down. Run the following command on the node with the appropriate credentials:

kubectl delete node <node name>

You can get the node name by running kubectl get nodes

Then, on the node being removed, reset all kubeadm installed state:

kubeadm reset

Note:

1. These are only some of the things that can be done with Kubernetes, you can find more information and command list over here.
2. Also, we have used kubeadm to deploy service, you can even use the commands by directly cloning the Kubernetes repository and running the various commands.

2.5 Tasks to be completed

As part of the exercise4 you will have to deploy the exercise3 (microservices) using Kubernetes along with some other tasks. The following tasks need to be completed:

1. Install docker on the VM.
2. Install Kubernetes as explained above.
3. Run the Kubernetes Cluster
4. Install the Pod network
5. Enable pod Scheduling on Master as explained in **section 2.2.3.**
6. Add a service in your Server Microservice with the name
    a. **/exercises/exercise4:** Send a message "group 'GroupNumber' application deployed using Kubernetes".
7. Run all 4 Microservices inside Kubernetes
8. Create services for all the 4 Microservices
    a. Expose **3 Microservices** (Hello_service, Product_Descp, Product_Price) inside the Cluster as explained in **section 2.3 4th Point.**
    b. Expose **4th Microservice** (Server), so that it can be accessed from outside the Cluster.
9. Scale your **Server Microservice** to have **2 Replicas**.
10. Run the following commands to expose the Kubernetes APi
    a. sudo kubectl proxy --address='0.0.0.0' --port=8001 --accept-hosts='^*$'&
    b. Enable Port **8001** in IPTables so that it can be accessed from outside.
11. Run the following command to check the port number of the Server Service on which it is running and afterwards **enable that port number in IP tables**.
    a. kubectl get services --all-namespaces
12. Visit the URL to test whether the Application is running or not

http://VM_IP:PORTNUMBER/ /exercises/exercise3?name=CCS&productId=3

3. Result Delivery

You already know how this works 😊. To submit your application results you need to follow this:

1. Open the cloud Class server url
2. Login with your username and password.
3. After logging in, you will find the button for exercise3
4. Click on it and a form will come up where you must provide your VM ip and the port number of the Server application is running.

   Example:

   10.0.23.1
   32456
5. Then click submit.

Important points to Note:

1. Make sure your VM and your application is running after following all the steps mentioned in this manual.
2. We will grade you based upon the number of exercises completed by you.
3. You will get to see, what your application has submitted to the server.
4. You can submit as many times until the deadline of exercise.
5. Multiple correct submissions will overwrite the previous results.

REFERNCES

1. **LRZ Supercomputing Center**
   https://www.lrz.de/english/

2. **Node.js official website**
   https://nodejs.org/en/

3. **Node.js Tutorial**
   https://www.tutorialspoint.com/nodejs/

4. **Node Package Manager**
   https://www.npmjs.com/

5. **Representational state transfer protocol.**
   https://en.wikipedia.org/wiki/Representational_state_transfer

6. **JSON format.**
   https://en.wikipedia.org/wiki/JSON

7. **Express js framework.**
   http://expressjs.com/

8. **Docker.**
   https://docs.docker.com/

9. **Docker Hub.**
   https://hub.docker.com/

10. **Kubernetes**
    https://kubernetes.io