Cloud Infrastructure HS 21

Kubernetes

by Dejan Jovicic and Thomas Kleb

Table of Contents

1 Introduction

The goal of this lab is to setup a Kubernetes cluster and load a simple Kubernetes application on it. We were given 3 nodes: master, worker-1 and worker-2 with pre-set IPs to connect to. The lab is structured to first install the prerequisites and tools for a Kubernetes cluster to work. After setting it up we made different verification and control checks to make sure everything works. After then setting up a storage system for the cluster we could deploy the application and troubleshooting problems.

All of the created, edited and used files can be found on our GitLab.

1.1 IP Addresses

To work with the Kubernetes cluster and connect to the different nodes we were given IP addresses which were pre-set on the nodes:

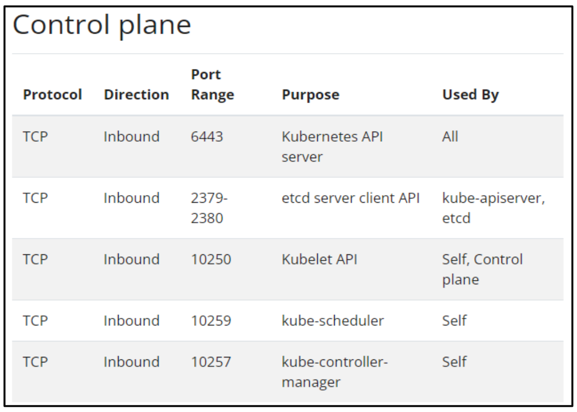
K8s VIP	master	worker-1	worker-2
10.18.9.7	10.18.10.110	10.18.10.111	10.18.10.112

2 Building a Kubernetes Cluster

2.1 Installation of Kubernetes

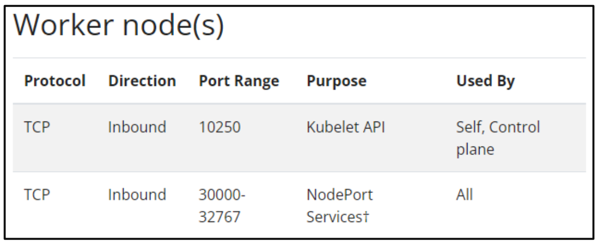
2.1.1 Checking Prerequisites

To make Kubernetes work, we installed kubeadm. Before installing it, following the guide, we had to make sure to check some things. The hardware specific requirements were given. We only had to check for connectivity, the uniqueness of the hostname, MAC address and product_uuid and if swap is disabled. The recommendations also show to open the ports for the nodes, but this was already done for us.



Control plane

Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	6443	Kubernetes API server	All
TCP	Inbound	2379-2380	etcd server client API	kube-apiserver, etcd
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	10259	kube-scheduler	Self
TCP	Inbound	10257	kube-controller-manager	Self



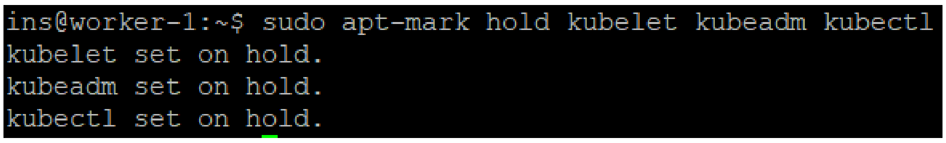
Worker node(s)

Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	30000-32767	NodePort Services†	All

Figure 1 Ports used by master and worker nodes

2.1.2 Installing

To install kubeadm, kubelet and kubectl we followed the provided guide[1]. The answers for the two questions can be found in chapter 2.1.7 and 2.1.8 respectively.



```
ins@worker-1:~$ sudo apt-mark hold kubelet kubeadm kubectl
kubelet set on hold.
kubeadm set on hold.
kubectl set on hold.
```

Figure 2 Output after running last command of setup

2.1.3 Container Runtime

The container runtime is the software responsible for running the containers as the name suggests. The Kubernetes website gave us three options: Containerd, CRI-O and Docker. Since the Docker runtime (Dockershim) is deprecated[2] and might be removed in a future release, we decided to use containerd out of the remaining options. To install the software, we followed again the guide provided by the lab document[3]. As in the lab mentioned we had to only install the containerd.io file, ignoring the docker installation guide.

2.1.4 Helm

Helm is a packet manager that allows us the installation and management of different Kubernetes applications. Helm or Helm Charts uses YAML templates for application configuration with separate “values.yaml” file to store all the values which are injected into the template YAML at the time of installation.

Installing it was rather simple just following the guide provided by the lab document[4].

2.1.5 CNI Plugin

The Container Network Interface (CNI) started as an initiative to define a standardized common interface between container execution and the networking layer. The CNI project is part of Cloud Native Computing Foundation (CNCF) and all the plugins must conform to the standard defined by the CNI specifications. In total there are four: Cilium, Calico, Flannel and Weave but the lab only gave us the first three as options. From these three, Flannel shouldn’t be chosen because of the limitations around network policies which would be a problem when we start adding them to the application. While Calico is probably the most popular CNI we chose to go with Cilium. One of the reasons we chose it over Calico is, that it allows us to add more fine-

[1] https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/
[2] https://kubernetes.io/blog/2020/12/02/dockershim-faq/
[3] https://kubernetes.io/docs/setup/production-environment/container-runtimes/
[4] https://helm.sh/docs/intro/install/

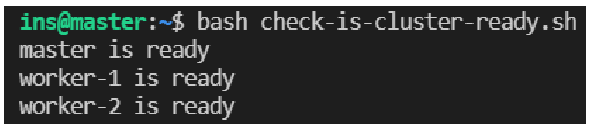
grained security controls, and if the Kubernetes cluster needs to be far bigger, Cilium allows for a reduced latency while looking them up.

2.1.6 Verification

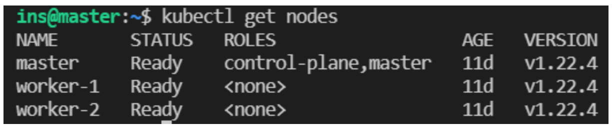
To check if the cluster is ready the lab required us to create a bash script which can be run. We could just enter a single command line which showed the `kubectl get nodes` output, but this wouldn't be interesting. We wrote our script to check each of the nodes' status and output "is ready" or "is not ready" for them.

```
#!/bin/sh
NUMBER=0
for i in 1 2 3
do
     JSONPATH="{range.items[$NUMBER]}{@.metadata.name}:
               {range@.status.conditions[?(@.type=='Ready')]}
               {@.type}={@.status};{end}{end}"
     STATUS=$(kubectl get nodes -o jsonpath="$JSONPATH")
     CHECK="Ready=True"
     if [[ "$STATUS" == *"$CHECK"* ]];
     then
          echo "${STATUS%%:*} is ready"
     else
          echo "${STATUS%%:*} is not ready"
     fi
     ((NUMBER++))
done
```

Figure 3 check-is-cluster-ready.sh bash script



```
ins@master:~$ bash check-is-cluster-ready.sh
master is ready
worker-1 is ready
worker-2 is ready
```



```
ins@master:~$ kubectl get nodes
NAME       STATUS   ROLES                  AGE   VERSION
master     Ready    control-plane,master   11d   v1.22.4
worker-1   Ready    <none>                 11d   v1.22.4
worker-2   Ready    <none>                 11d   v1.22.4
```

Figure 4 check-is-cluster-ready.sh bash script compared to normal command output

2.1.7 Disadvantages

There are possible disadvantages with having only one control plane node. A downtime of said node makes it hard or even impossible to change any Kubernetes objects. For example, if the master (control plane) node is down for any reason, we would have difficulties to schedule new deployments, update application configuration or even adding / removing additional worker nodes. Another possibility is that, if a worker node happens to go down while the master node is down too, the worker node might have some problems re-joining the cluster after recovery

2.1.8 Diagram for a High-Available Kubernetes Cluster

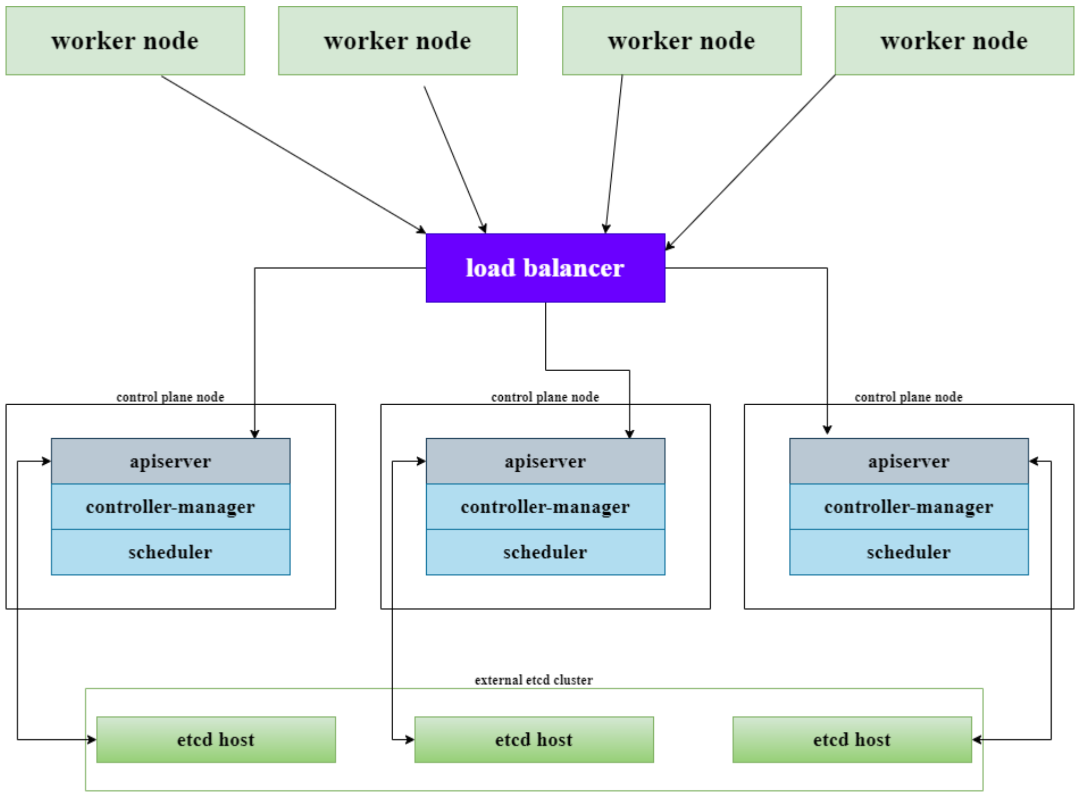


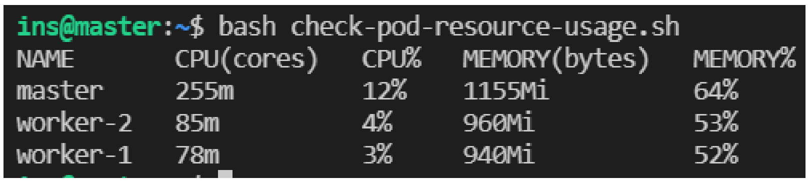
Figure 5 HA kubernetes cluster

We have decided to draw an “external etcd topology” HA cluster. Each control plane node runs an instance of the apiserver, controller-manager and scheduler. The worker nodes access via the load balancer the kube-apiserver. The etcd members run on hosts, which are separated from the control plane node and each etcd host communicates with the apiserver of each control plane node. This topology provides an HA setup, where losing a control plane instance or an etcd member has less impact and doesn’t affect the cluster redundancy as much as a HA topology with the etcd members being in the control plane nodes. It also is easier to setup. The downside of this topology is, that it does require twice the number of hosts as a stacked HA topology. A minimum of 3 hosts for etcd nodes and control plane nodes are required.

2.2 Inspecting Node/Pod Resource Usage

To get the cpu usage of the nodes we first had to install the metrics server by downloading the components.yaml file from the git and adding the "—kubelet-insecure-tls" to it before running the command: `kubectl apply -f components.yaml` to install the server with these settings.

We didn't find a fancy way to write this bash script, so we just added the command to it which sorts it by CPU usage instead of the name: `kubectl top nodes --sort-by=cpu`.



```
ins@master:~$ bash check-pod-resource-usage.sh
NAME        CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
master      255m         12%    1155Mi          64%
worker-2    85m          4%     960Mi           53%
worker-1    78m          3%     940Mi           52%
```

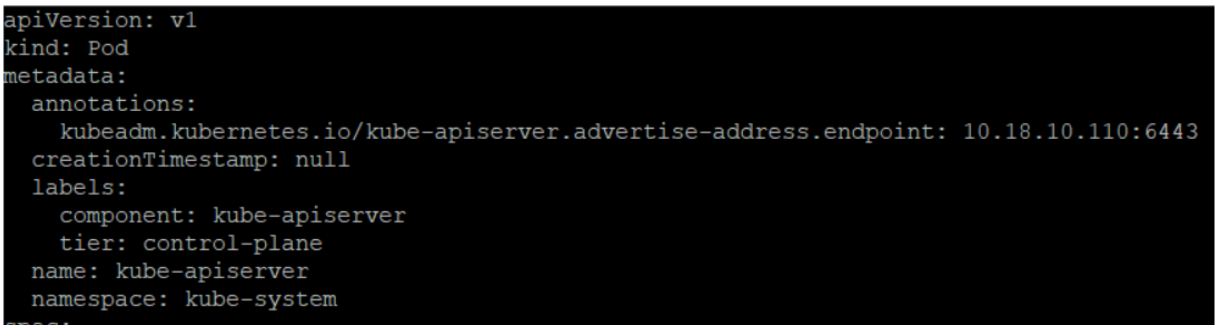
Figure 6 check-pod-resource-usage.sh script output

2.3 Advanced Questions

2.3.1 Get Master Information[5]

Kubelet is the primary "node agent" that runs on nodes. It can register the node with the apiserver using the hostname for example. Therefore, we would define it as a process (out of the given options since it is installed on the master node and isn't a pod)

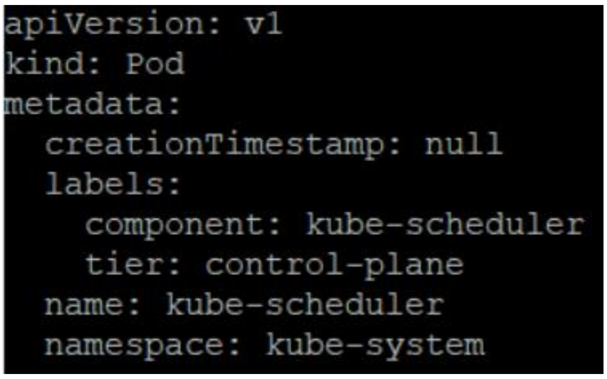
The next four: kube-apiserver, kube-scheduler, kube-controller-manager and etcd have "kind: pod" in their options yaml. But since they have a manifest installed in the /etc/Kubernetes/manifests folder, they are static. This is the path were kubelet should look for static Pod manifests.



```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint: 10.18.10.110:6443
  creationTimestamp: null
  labels:
    component: kube-apiserver
    tier: control-plane
  name: kube-apiserver
  namespace: kube-system
```

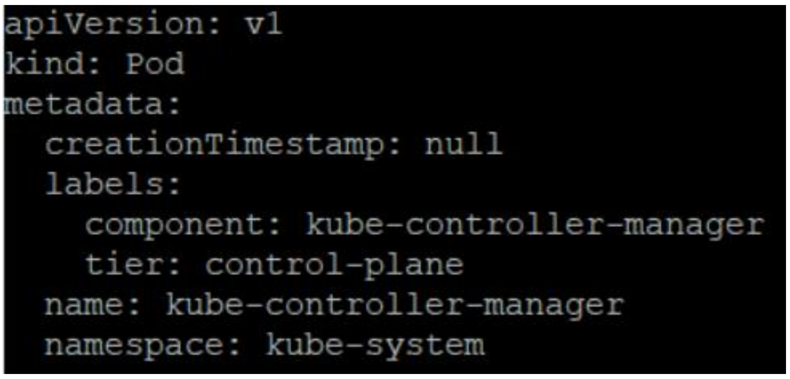
Figure 7 Master information for the kube-apiserver

[5] https://kubernetes.io/docs/reference/setup-tools/kubeadm/implementation-details/#constants-and-well-known-values-and-paths



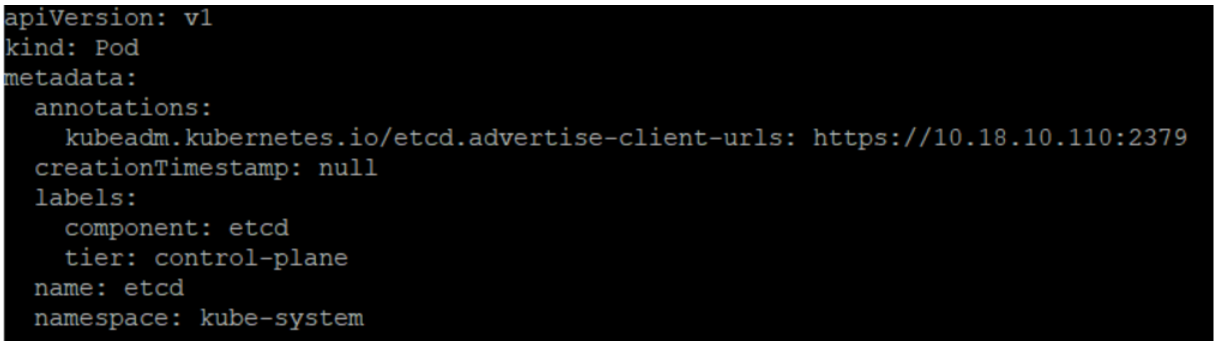
```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    component: kube-scheduler
    tier: control-plane
  name: kube-scheduler
  namespace: kube-system
```

Figure 8 Master information for the kube-scheduler



```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    component: kube-controller-manager
    tier: control-plane
  name: kube-controller-manager
  namespace: kube-system
```

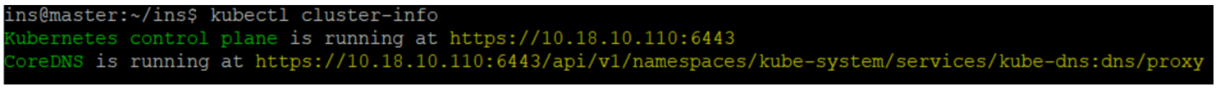
Figure 9 Master information for the kube-controller-manager



```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubeadm.kubernetes.io/etcd.advertise-client-urls: https://10.18.10.110:2379
  creationTimestamp: null
  labels:
    component: etcd
    tier: control-plane
  name: etcd
  namespace: kube-system
```

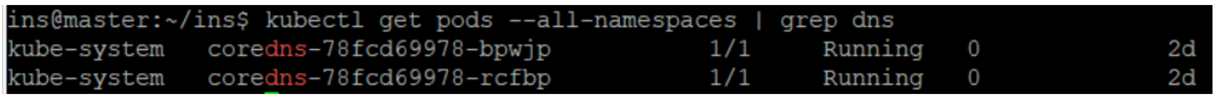
Figure 10 Master information for etcd

The DNS name can be found by entering: `kubectl cluster-info`



```
ins@master:~/ins$ kubectl cluster-info
Kubernetes control plane is running at https://10.18.10.110:6443
CoreDNS is running at https://10.18.10.110:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
```

Figure 11 kubectl cluster-info output to get name of the DNS service

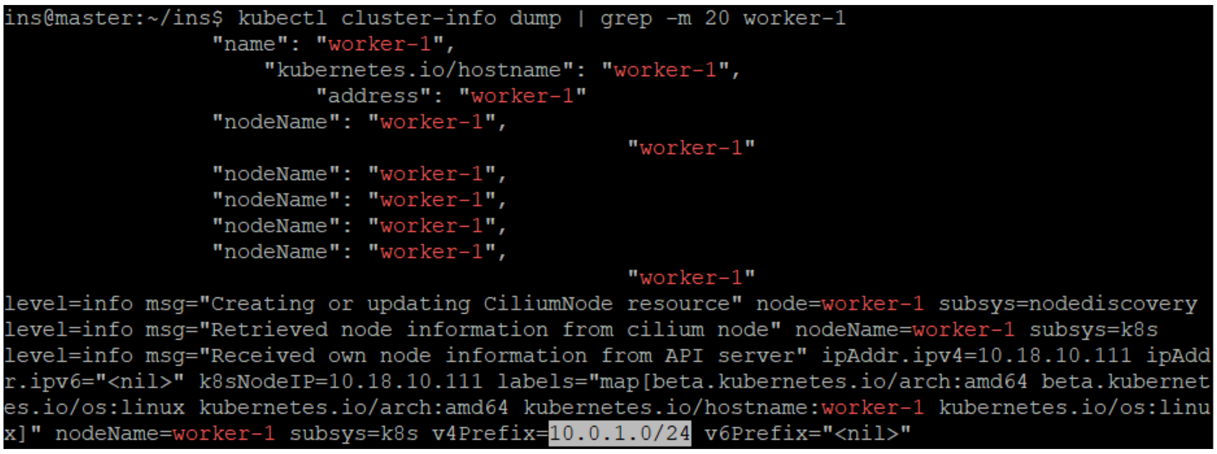


```
ins@master:~/ins$ kubectl get pods --all-namespaces | grep dns
kube-system   coredns-78fcd69978-bpwjp          1/1     Running   0          2d
kube-system   coredns-78fcd69978-rcfbp          1/1     Running   0          2d
```

Figure 12 Verifying found DNS service name

2.3.2 Find out Cluster Information

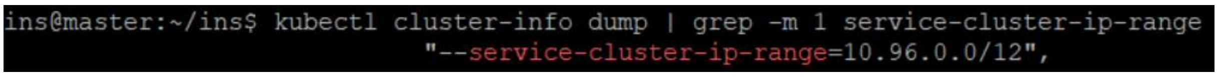
To get the Pod CIDR of worker-1 we used grep on the cluster-info dump and found 10.0.1.0/24



```
ins@master:~/ins$ kubectl cluster-info dump | grep -m 20 worker-1
                "name": "worker-1",
                    "kubernetes.io/hostname": "worker-1",
                        "address": "worker-1"
                "nodeName": "worker-1",
                                                "worker-1"
                "nodeName": "worker-1",
                "nodeName": "worker-1",
                "nodeName": "worker-1",
                "nodeName": "worker-1",
                                                "worker-1"
level=info msg="Creating or updating CiliumNode resource" node=worker-1 subsys=nodediscovery
level=info msg="Retrieved node information from cilium node" nodeName=worker-1 subsys=k8s
level=info msg="Received own node information from API server" ipAddr.ipv4=10.18.10.111 ipAdd
r.ipv6="<nil>" k8sNodeIP=10.18.10.111 labels="map[beta.kubernetes.io/arch:amd64 beta.kubernet
es.io/os:linux kubernetes.io/arch:amd64 kubernetes.io/hostname:worker-1 kubernetes.io/os:linu
x]" nodeName=worker-1 subsys=k8s v4Prefix=10.0.1.0/24 v6Prefix="<nil>"
```

Figure 13 Grep to get the worker-1 CIDR

Similar to getting the worker-1 CIDR above, for the Service CIDR we used grep again and got 10.96.0.0/12



```
ins@master:~/ins$ kubectl cluster-info dump | grep -m 1 service-cluster-ip-range
                                "--service-cluster-ip-range=10.96.0.0/12",
```

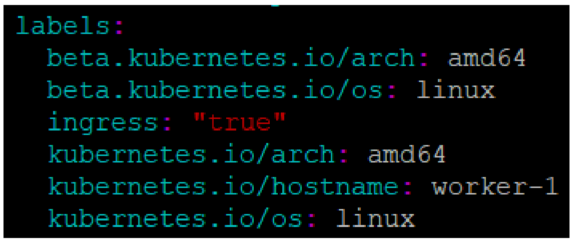
Figure 14 Grep to get the service CIDR

Static Pods running on the worker-1 node will be suffixed with the node hostname and a leading hyphen: “-worker-1”.

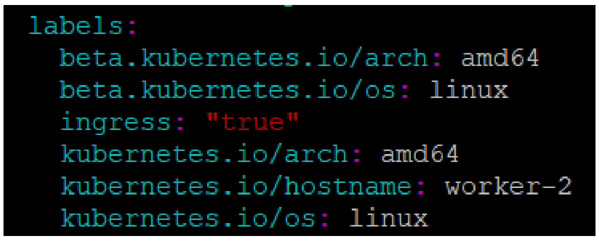
3 Ingress Controller

3.1 Prerequisites

To get a working ingress controller we needed an additional tool called MetalLB. It provides a network load-balancer implementation for our Kubernetes cluster. It can be deployed either with a simple Kubernetes manifest or with Helm (we used helm as it was recommended by the lab document). Before installing we gave both worker nodes the "ingress: "true""-label.



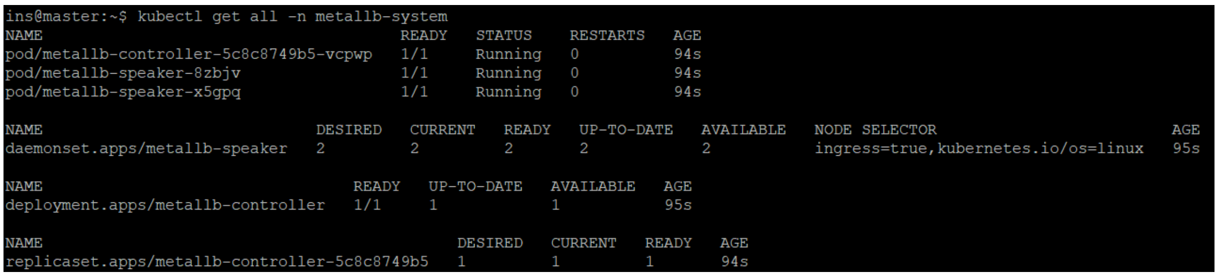
```
labels:
  beta.kubernetes.io/arch: amd64
  beta.kubernetes.io/os: linux
  ingress: "true"
  kubernetes.io/arch: amd64
  kubernetes.io/hostname: worker-1
  kubernetes.io/os: linux
```



```
labels:
  beta.kubernetes.io/arch: amd64
  beta.kubernetes.io/os: linux
  ingress: "true"
  kubernetes.io/arch: amd64
  kubernetes.io/hostname: worker-2
  kubernetes.io/os: linux
```

Figure 15 Labels for worker-1 and worker-2 before installing MetalLB

We created a values.yaml file with address-pools name "nginx" and our K8s VIP 10.18.9.7/32 and installed it with helm. After installing we checked if the MetalLB service works as intended



```
ins@master:~$ kubectl get all -n metallb-system
NAME                                      READY   STATUS    RESTARTS   AGE
pod/metallb-controller-5c8c8749b5-vcpwp   1/1     Running   0          94s
pod/metallb-speaker-8zbjv                 1/1     Running   0          94s
pod/metallb-speaker-x5gpq                 1/1     Running   0          94s

NAME                             DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR                             AGE
daemonset.apps/metallb-speaker   2         2         2       2            2           ingress=true,kubernetes.io/os=linux   95s

NAME                                 READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/metallb-controller   1/1     1            1           95s

NAME                                            DESIRED   CURRENT   READY   AGE
replicaset.apps/metallb-controller-5c8c8749b5   1         1         1       94s
```

Figure 16 Checking if MetalLB runs correctly

3.2 Installation

This is the values.yaml to install the NGINX Ingress Controller[6]. Again, installing it using helm

```
configInline:
  address-pools:
    - name: nginx
      protocol: layer2
      addresses:
        - 10.18.9.7/32

controller:
  kind: DaemonSet
  nodeSelector:
    ingress: "true"
  service:
    enabled: true
    type: LoadBalancer
    externalTrafficPolicy: Local
    annotations:
      metallb.universe.tf/address-pool: nginx

speaker:
  nodeSelector:
    ingress: "true"
```

Figure 17 NGINX Ingress Controller values.yaml

3.2.1 Questions:

If we create a Kubernetes cluster in a cloud environment (Amazon EKS, Google GKE) why is a tool such as MetalLB not necessary? Explain.

Because in cloud environments like Amazon EKS or Google GKE Kubernetes is supported and therefore you can use the loadbalancer of the cloud environments together with Kubernetes instead of installing your own.

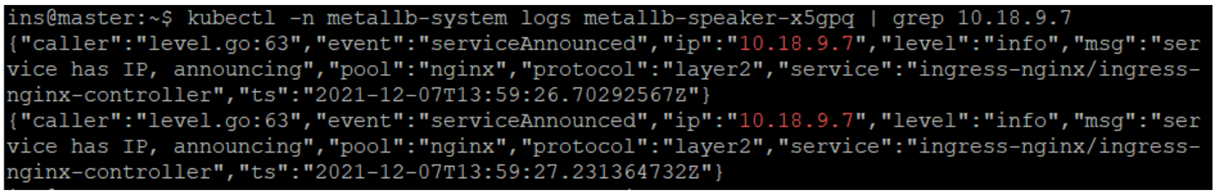
What does the externalTrafficPolicy: Local mean?

If you want to preserve the source IP address, you'll need to set the externalTrafficPolicy to "Local". When a node receives the traffic it sends it only to the service pods that are on the same node. There won't be any traffic flow between the nodes, thus the reason why your pods can see the real source IP address of incoming connections. The downside of this policy is that incoming traffic only goes to some pods in the service, the other pods that aren't on the current active node, are just acting as replicas in the case of a failover.

[6] https://kubernetes.github.io/ingress-nginx/deploy/

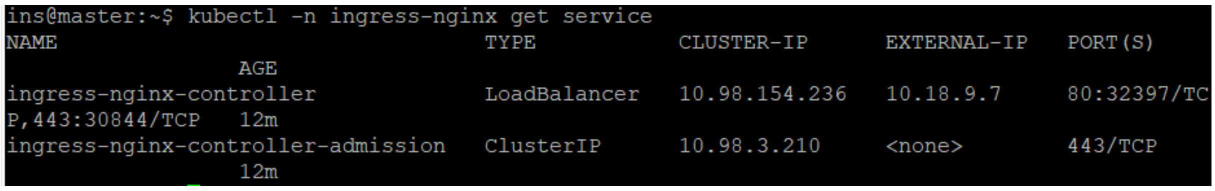
3.3 Verification

To check if the worker nodes have the correct VIP assigned, we used grep on the log file and the “get-service” command of kubectl.



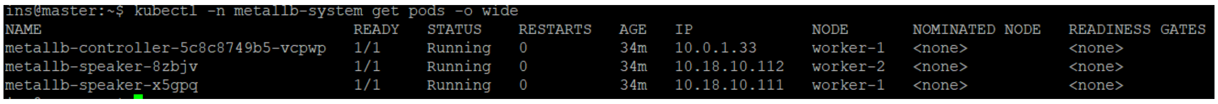
```
ins@master:~$ kubectl -n metallb-system logs metallb-speaker-x5gpq | grep 10.18.9.7
{"caller":"level.go:63","event":"serviceAnnounced","ip":"10.18.9.7","level":"info","msg":"ser
vice has IP, announcing","pool":"nginx","protocol":"layer2","service":"ingress-nginx/ingress-
nginx-controller","ts":"2021-12-07T13:59:26.70292567Z"}
{"caller":"level.go:63","event":"serviceAnnounced","ip":"10.18.9.7","level":"info","msg":"ser
vice has IP, announcing","pool":"nginx","protocol":"layer2","service":"ingress-nginx/ingress-
nginx-controller","ts":"2021-12-07T13:59:27.231364732Z"}
```

Figure 18 Grep on the log file to get the assigned VIP



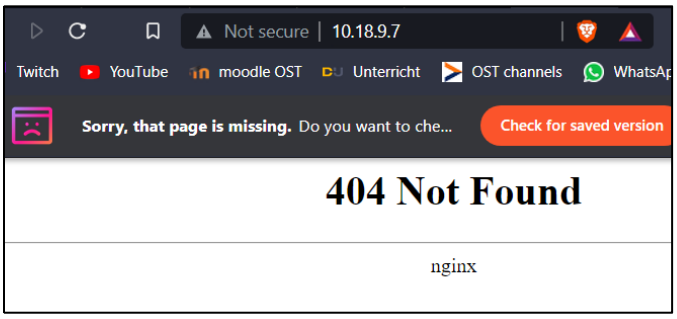
```
ins@master:~$ kubectl -n ingress-nginx get service
NAME                                 TYPE           CLUSTER-IP      EXTERNAL-IP    PORT(S)
                  AGE
ingress-nginx-controller             LoadBalancer   10.98.154.236   10.18.9.7      80:32397/TC
P,443:30844/TCP   12m
ingress-nginx-controller-admission   ClusterIP      10.98.3.210     <none>         443/TCP
                  12m
```

Figure 19 Get-service output to check if nginx controller is installed



```
ins@master:~$ kubectl -n metallb-system get pods -o wide
NAME                                 READY   STATUS    RESTARTS   AGE   IP             NODE       NOMINATED NODE   READINESS GATES
metallb-controller-5c8c8749b5-vcpwp  1/1     Running   0          34m   10.0.1.33      worker-1   <none>           <none>
metallb-speaker-8zbjv                1/1     Running   0          34m   10.18.10.112   worker-2   <none>           <none>
metallb-speaker-x5gpq                1/1     Running   0          34m   10.18.10.111   worker-1   <none>           <none>
```

Figure 20 get pods with wide output



Sorry, that page is missing. Do you want to che...

404 Not Found

nginx

Figure 21 Connecting to the IP to check for the Error message "nginx" connection to the sslip.io worked aswell

4 Storage

As told by the lab document we don’t have any external storage available, that’s why we had to create our own storage on the cluster.

4.1 StorageClass

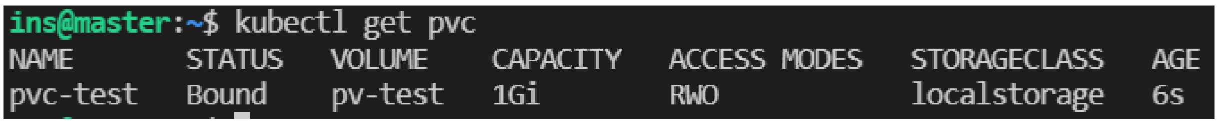
A storage class provides a way for us to describe the "classes" of storage we use. Each storage class contains the fields provisioner, parameters, and “reclaimPolicy”, which are used when a persistent volume belonging to the class needs to be dynamically provisioned. Different to other metadata names, the name in the storage class is important since its users can request a particular class this way using the “storageClassName”.

We were given the settings for their storage class by the lab document. The “reclaimPolicy” is set to “Retain” which will be used by the persistent volumes too. The “allowVolumeExpansion” allows the resizing of volumes by editing the corresponding claim object. The “waitForFirstConsumer” setting would delay the binding and provisioning of a persistent volume until a pod using the claim is created and wasn’t set.

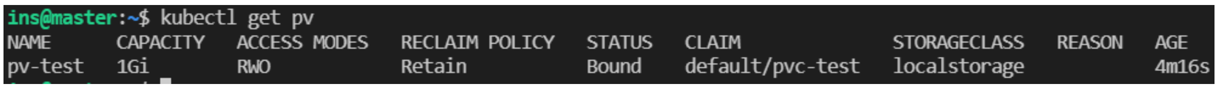
4.2 Verification

In this step we created the persistent volumes and their claim to access and interact with the storage we created. All the settings used were again given to us by the lab document and we used them to create the two YAML files, applying them with the `kubectl apply -f <yaml>` command.

Checking the claims’ status, we saw its in “Bound”. We edited the file with the command given to us and dumped the results in a “storage-verification” text file which can be found on our GitLab together with the YAML files for the storage, the volume and the claim.



```
ins@master:~$ kubectl get pvc
NAME       STATUS   VOLUME    CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pvc-test   Bound    pv-test   1Gi        RWO            localstorage   6s
```



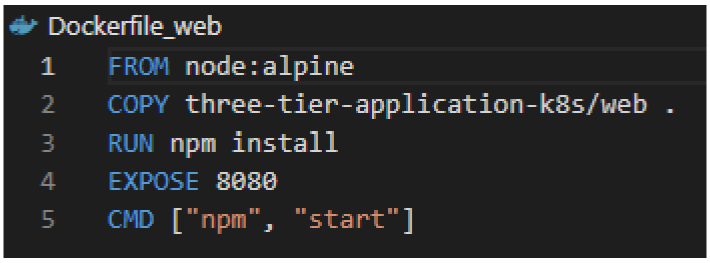
```
ins@master:~$ kubectl get pv
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM              STORAGECLASS   REASON   AGE
pv-test   1Gi        RWO            Retain           Bound    default/pvc-test   localstorage            4m16s
```

5 Deploy an App

5.1 Setting up Docker File with GIT

We decided to build our images directly using the given pipeline. To use this pipeline to create our GitLab registries, we first had to design our Dockerfiles.

The Web file looks as followed:



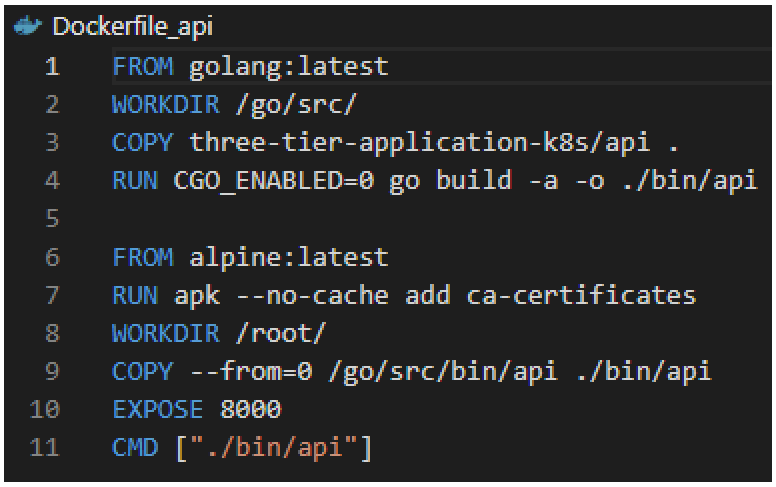
Dockerfile_web

```
FROM node:alpine
COPY three-tier-application-k8s/web .
RUN npm install
EXPOSE 8080
CMD ["npm", "start"]
```

Figure 22 Web Dockerfile

To briefly explain what each of the command line does: The `FROM` instruction initializes a new build stage and sets the Base Image for subsequent instructions. As such, a valid Dockerfile must start with a `FROM` instruction, The `COPY <src> <dest>` instruction copies new files or directories from `<src>` to `<dest>`. In our example the source was from the web directory and the destination was the current working directory (“.”). The `RUN <command> <parameter>` instruction executes any command (in shell form) in a new layer (/bin/sh -c as default) on top of the current image and commits the results. The resulting committed image can then be used in the next steps. There are known issues using the `RUN` instruction with file permissions problems when using the AUFS file system, but since this wasn’t a problem in our case, we used the `RUN` instruction. The `EXPOSE <port>` instruction informs Docker that the container listens on the specified network ports at runtime. Additionally, the protocol can be specified (default is TCP). The `CMD` instruction has different forms but the exec form (`CMD [“executable”,”parameter”])` is recommended. There can only be one `CMD` instruction in a Dockerfile. If more are listed only the last will take effect. We used it to start the server.

For the API Dockerfile we used the same commands but with different settings:



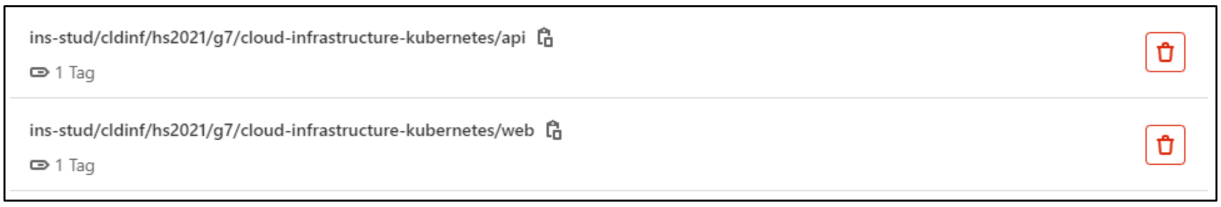
Dockerfile_api
```
FROM golang:latest
WORKDIR /go/src/
COPY three-tier-application-k8s/api .
RUN CGO_ENABLED=0 go build -a -o ./bin/api

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=0 /go/src/bin/api ./bin/api
EXPOSE 8000
CMD ["./bin/api"]
```

Figure 23 API Dockerfile

Since the API Dockerfile is a multistage build, we had to apply an additional instruction called `WORKDIR`. The `WORKDIR <PATH>` instruction sets the working directory for any `RUN`, `CMD`, `ENTRYPOINT`, `COPY` and `ADD` instructions that follow it in the Dockerfile. If the `WORKDIR` doesn't exist (Web Dockerfile), it will be created even if it's not used in any subsequent Dockerfile instruction.

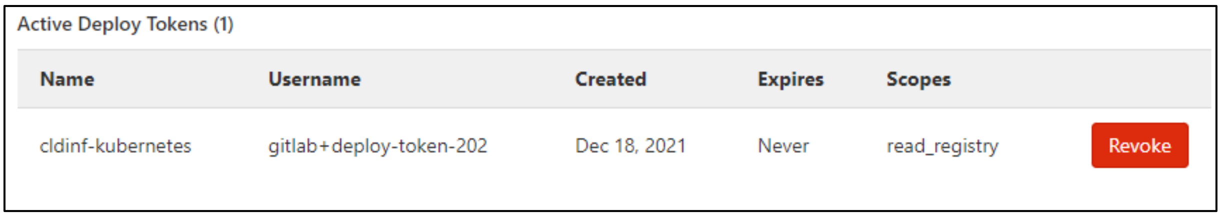
After pushing the files on our repository, the pipeline created the container registries which we used to pull the images from:



ins-stud/cldinf/hs2021/g7/cloud-infrastructure-kubernetes/api
1 Tag
ins-stud/cldinf/hs2021/g7/cloud-infrastructure-kubernetes/web
1 Tag

Figure 24 Container Registries generated by Pipeline

We had to additionally create a "deploy token" to allow the pods / deployment to access them using "ImagePullSecrets: <secret name>". To create such a token in the GitLab go to Settings – Repository – Deploy tokens and create one. For this lab only the "read-registry" scope is needed. After Creating it will show up below:



Active Deploy Tokens (1)

Name	Username	Created	Expires	Scopes	
cldinf-kubernetes	gitlab+deploy-token-202	Dec 18, 2021	Never	read_registry	Revoke

Figure 25 Deploy Token to allow the Pods to access our Registry

To create and use the secret on the cluster we had to enter the Kubernetes command with the password and username created by GIT:

```
kubectl create secret docker-registry regcred \
--docker-server=https://registry.gitlab.ost.ch:45023 \
--docker-username=gitlab+deploy-token-202 \
--docker-password=<password> \
--namespace=cldinf-app
```

Figure 26 Creation of Secret on the Cluster with Kubectl

5.2 Service and Deployment

A service on Kubernetes is a kind of direction to a logical set of pods and a policy by which to access them. Every pod gets its own IP but in a deployment the set of pods running can be different from time to time it creates This can create a problem if there is a frontend to backend structure in a cluster. The service controls this problem by keeping the frontend side updated on which IP it needs to connect to receive data from the backend. For Kubernetes they are in form of a YAML file and are applied on the control plane (master for this lab). The files are stored on this node and can be applied to the nodes using the `kubectl apply -f` command. Services can be changed by editing their YAML data consisting mainly of apiVersion-, kind-, metadata- and spec-settings. The api Version is to define the syntax and “kind: Service” indicates to the controller to make a Service with these settings. Metadata is used to add various information, for this lab it was mainly used to add a name, label, and the namespace which the pods are created in (cldinf-app). A template[7] of them can be found on the Kubernetes website and our version of it on the GitLab

A deployment is used to control pods by providing updates and upgrades. It describes a desired state to which the deployment controller changes the actual state. A deployment YAML file also consists of apiVersion-, kind-, metadata-, and spec-settings. These are similar to the service but the spec in deployments is used to define the containers and many other configurations. There is a general template[8] on the Kubernetes website on which we based our settings. The final versions of all the deployments can be found in the YAML directory on our GitLab. Deployment YAMLs are applied like the service ones.

[7] https://kubernetes.io/docs/concepts/services-networking/service/
[8] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

5.3 Web Tier

For the Web tier to work, we first had to make a deployment for it. In this report we only show the settings which have to be looked out for and are important to make the deployment work. Some aren't found on the template from Kubernetes. We excluded the obvious settings like "kind: Deployment" in our report. For the Web tier we need to have three replicas (3 active pods) and take the image from our generated container registry with the link you can get by clicking on the clipboard icon next to the web registry. Beside the environmental variables API_HOST and API_PORT we had to add GROUP_NAME and MODULE_NAME to the containers' settings. We gave the apiservice as value for the host variable, the port of the API, 8000, as string to the port variable and for the other two our group name and the module. All these Variables were set mandatory by the Readme given to us. At last, we added the "imagePullSecrets" setting with the value "name: regcred" to get the deploy token and allow the pods to pull the image from our registry

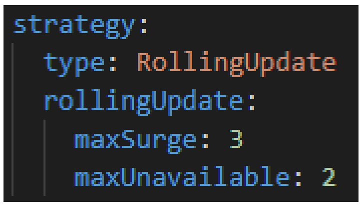
The next step of making the Web Tier work was to add a service which controls the deployment. Same as with the deployment we only talk about the settings we changed from or added to the default template of Kubernetes. For the web service we only had to change the port to 8080, added the namespace and changed the name and selector to fit our topology (the selector shows the service where to find the deployment by adding the label as value, "app: cldinf-web" in our case).

Now to make this tier accessible from the OST network we had to create an ingress object again from a template[9] and changing settings to fit our topology. Beside adding the namespace and changing the name we also added annotations to the metadata. This was to add a Rewrite to target the URI where the traffic must be redirected. In the spec settings we also added the ingress class name "nginx" to define it. Adding the host DNS "10-18-9-7.sslip.io". Changing the path, the service name (to direct the ingress to the web-service) and the port number allowed the ingress to work correctly.

[9] https://kubernetes.io/docs/concepts/services-networking/ingress/

5.3.1 Availability

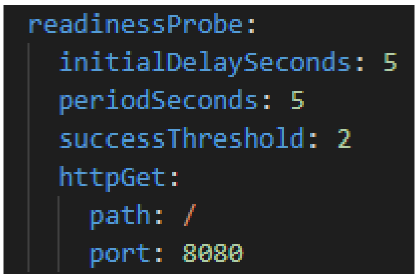
To ensure that at least 1 pod is active during an update Kubernetes allows to use “Rolling Updates”[10] which incrementally updates pods, instead of all at once. To make the deployment perform such updates we had to add it to its specifications. This is done by creating a new “strategy” tag and adding the settings to it. There are multiple options that can be added but the one that we found the most important for the given task was the “maxUnavailable” tag. This specifies the maximum number of unavailable pods during an update. It can be specified through a percentage or an absolute number and since we have 3 pods running 2 can be unavailable during an update. The “maxSurge” option specifies the maximum number of pods to be created beyond the desired state during the update.



```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 3
    maxUnavailable: 2
```

Figure 27 Rolling Update

Since Kubernetes can’t tell when a new pod is ready (it eliminates the old pod as soon as the new one is created), it can create downtime until the new pod becomes able to accept requests. To fix this problem we added a “readinessProbe” which checks the state of pods and allow for rolling update to proceed only when all the containers in a pod are ready. Again, there are many options to add to that probe, but we decided to only add the ones which seemed to be needed. The “initialDelaySeconds” option specifies how long the probe has to wait to start after the container starts, “periodSeconds” specifies the time between two probes and “successThreshold” the number of consecutive successful probes after a failed one to consider a process a success. We commented that part out of the final version of our yaml, since it’s an optional part.



```
readinessProbe:
  initialDelaySeconds: 5
  periodSeconds: 5
  successThreshold: 2
  httpGet:
    path: /
    port: 8080
```

Figure 28 readinessProbe for R. Update

[10] https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/

5.4 API Tier

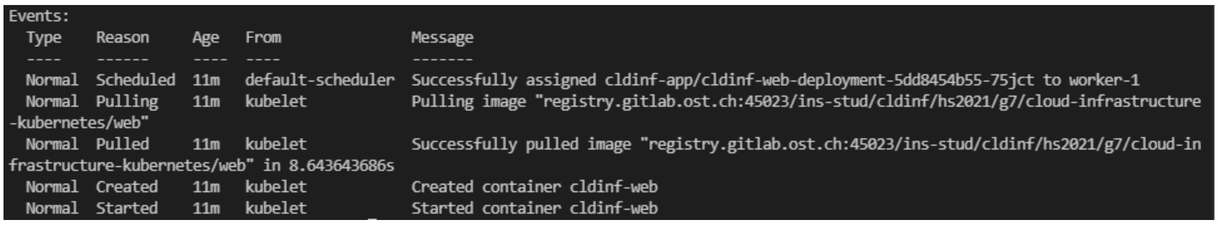
The API deployment is structurally pretty similar to the web deployment (different name, labels and image of course). Since the API only runs 1 replica at a time, configuring a rollout update would be useless.

For the API we had to add different environmental variables (given by the Readme). These are the variables to login to the database and get access to it: PG_USER, PG_PASSWORD, PG_DATABASE and PG_HOST. There are additional, optional variables but the default values of them corresponds with the value we want, which means its not needed to add them. Only the DEBUG variable was used to find and fix mistakes we did.

The API service was again similar to the web service (they differ in name, namespace and selector). As port we were told to use 8000.

5.4.1 Debugging and Logging

To see event-logs of the pod the command `kubectl describe pod <pod-name> -n=<namespace>` can be used. Scrolling to the bottom of the output will show you a log of the events that happened on the pod:



```
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  11m   default-scheduler  Successfully assigned cldinf-app/cldinf-web-deployment-5dd8454b55-75jct to worker-1
  Normal  Pulling    11m   kubelet            Pulling image "registry.gitlab.ost.ch:45023/ins-stud/cldinf/hs2021/g7/cloud-infrastructure
-kubernetes/web"
  Normal  Pulled     11m   kubelet            Successfully pulled image "registry.gitlab.ost.ch:45023/ins-stud/cldinf/hs2021/g7/cloud-in
frastructure-kubernetes/web" in 8.643643686s
  Normal  Created    11m   kubelet            Created container cldinf-web
  Normal  Started    11m   kubelet            Started container cldinf-web
```

Figure 29 Eventlog; Output of kubectl describe command

To allow debugging on the API Tier we had to add the DEBUG with the value “True” to the environmental variables. With the command `kubectl logs <pod-name> -n=<namespace>` you can then output the logs created by the application to the terminal. Since this wasn’t allowed for the task we tried different options like side-car deployments but couldn’t make it work in time.

5.5 Database Tier

Setting up the database differs to the other two deployments. First of all, it's a StatefulSet instead of a deployment since they're valuable for applications that require stable, persistent storage provisioned by a persistent volume. Deleting the StatefulSet will not delete the volumes associated with it. This is perfect for data safety.

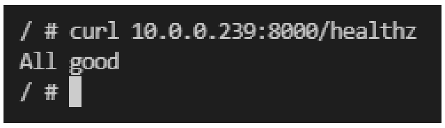
As additional value to the "matchLabels" selector we added "tier:database". For the image we pulled the latest version of postgres and limited the cpu resources to guarantee performance. Additionally, we added the environmental variables: POSTGRES_USER, POSTGRES_PASSWORD and POSTGRES_DB to allow connection to the database. The database needs a persistent volume claim which we gave by adding the information to the volumes selector. Finally, the nodeSelector allows us to control on which node the database will be, which is worker-1 as told by the lab document.

A StatefulSet needs a service too, like the deployment does, which is why we created one based on the others but this time with port 5432 (postgres default port) and an additional selector "tier:database".

5.5.1 pVolume and pvClaim

The database is built on a storage class we did in chapter 5. We created new persistent volumes and their claim with the correct namespace, labels, and names. We didn't need the "persistentVolumeReclaimPolicy: Retain" in the persistent volume and changed the storage request to 1Gi. The files can be found on our GitLab.

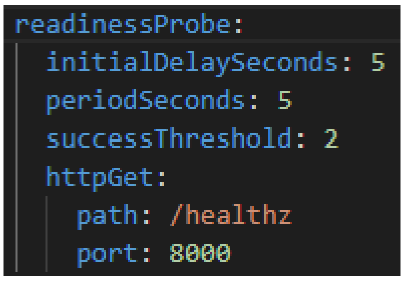
5.6 Verification



```
/ # curl 10.0.0.239:8000/healthz
All good
/ #
```

Figure 30 curl with /healthz path

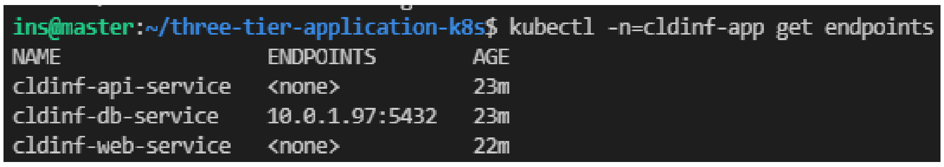
In this step we checked different options like container health, endpoints, uptime, updates and so on. First, we added the readinessProbe on the API (commented out in file) with the path /healthz and port 8000 to check the health of the container. To make sure the path works we checked it by opening a shell on the api pod with `kubectl exec -it <api pod> -n=cldinf-app – sh`. Entering the curl command with the apis ip or localhost port and path gave us an "All good" response.



```
readinessProbe:
  initialDelaySeconds: 5
  periodSeconds: 5
  successThreshold: 2
  httpGet:
    path: /healthz
    port: 8000
```

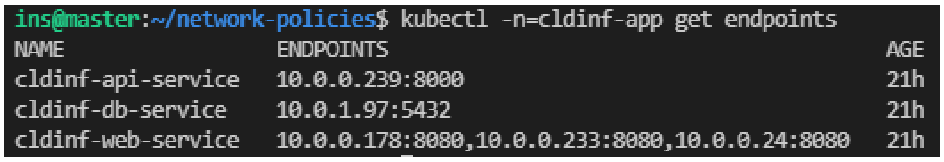
Figure 31 Readiness Probe for the API (Container health-check)

We also had problems with the endpoints which got fixed by entering the correct label in the corresponding services:



```
ins@master:~/three-tier-application-k8s$ kubectl -n=cldinf-app get endpoints
NAME                  ENDPOINTS        AGE
cldinf-api-service    <none>           23m
cldinf-db-service     10.0.1.97:5432   23m
cldinf-web-service    <none>           22m
```

Figure 32 Endpoints before (wrong labels in services' selector)



```
ins@master:~/network-policies$ kubectl -n=cldinf-app get endpoints
NAME                  ENDPOINTS                                              AGE
cldinf-api-service    10.0.0.239:8000                                        21h
cldinf-db-service     10.0.1.97:5432                                         21h
cldinf-web-service    10.0.0.178:8080,10.0.0.233:8080,10.0.0.24:8080        21h
```

Figure 33 Endpoints after (correct labels in services' selector)

5.7 Network Policies

Kubernetes allows to add network policies to pods. These control the ingress and egress traffic with rules that we can define. These Policies work on a default deny all principle which made them easier to build. We used the base template[11] from the Kubernetes documentation and added lines to it to make it fit our plan:

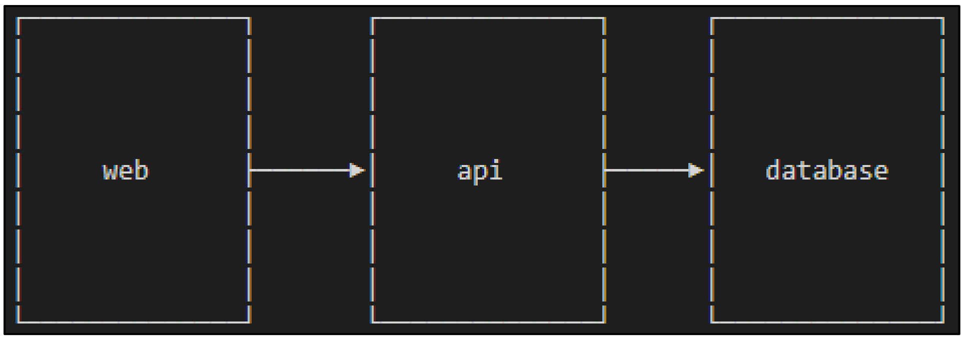
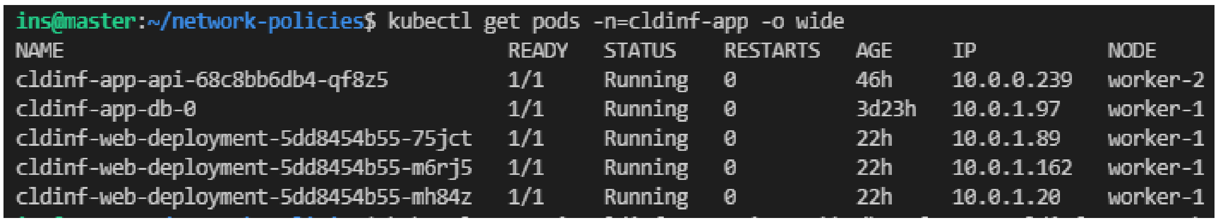


Figure 34 Simplification of Topology of the Pods

For port policy we allowed only the configured ports to be used and only TCP and UDP protocols since they were needed to establish the connections. In general, we needed the ports TCP 8000, 8080 and 5432; as well as the UDP port 53 to allow DNS to work. The other settings and verifications are described in the following chapters.

As with the files before, network-policies are YAML files applied to the cluster using the `kubectl apply -f <yaml file>` command. The files can be found on our GitLab as in this report we only describe what we used and not how the file looked at the end.

5.7.1 Overview



```
ins@master:~/network-policies$ kubectl get pods -n=cldinf-app -o wide
NAME                                  READY   STATUS    RESTARTS   AGE     IP            NODE
cldinf-app-api-68c8bb6db4-qf8z5       1/1     Running   0          46h     10.0.0.239    worker-2
cldinf-app-db-0                       1/1     Running   0          3d23h   10.0.1.97     worker-1
cldinf-web-deployment-5dd8454b55-75jct 1/1    Running   0          22h     10.0.1.89     worker-1
cldinf-web-deployment-5dd8454b55-m6rj5 1/1    Running   0          22h     10.0.1.162    worker-1
cldinf-web-deployment-5dd8454b55-mh84z 1/1    Running   0          22h     10.0.1.20     worker-1
```

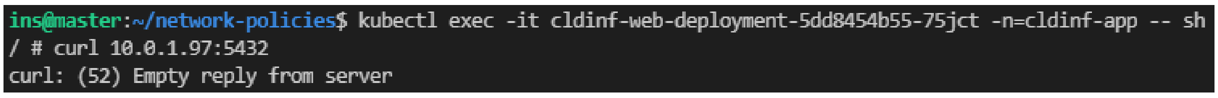
Figure 35 Overview of Cluster (kubectl get pods -n=<namespace> -o wide)

5.7.2 Database

As seen in first figure of the chapter before (6.7 Network Policies), only the API pod should be able to access the database which is why we applied an ingress rule that only allows pods that were deployed by the api-deployment to connect.

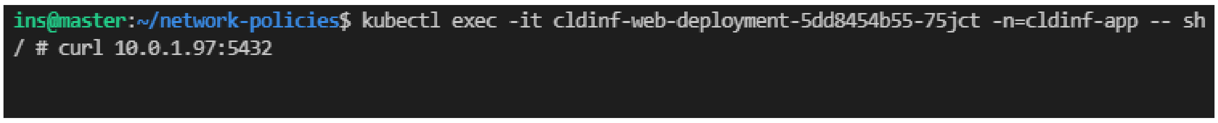
[11] https://kubernetes.io/docs/concepts/services-networking/network-policies/

To confirm that the policy works we connected to a web pod and tried to curl to the database. The result was, as expected, that no response came back:



```
ins@master:~/network-policies$ kubectl exec -it cldinf-web-deployment-5dd8454b55-75jct -n=cldinf-app -- sh
/ # curl 10.0.1.97:5432
curl: (52) Empty reply from server
```

Figure 36 Curl command on a Web Pod before Network Policy



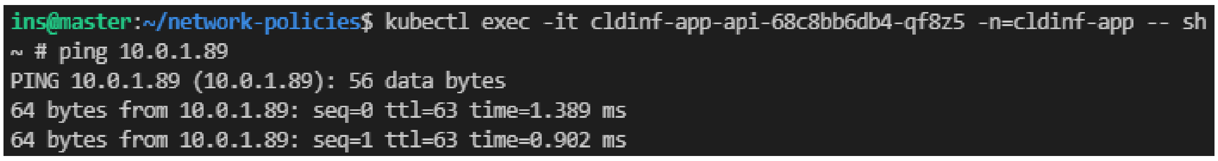
```
ins@master:~/network-policies$ kubectl exec -it cldinf-web-deployment-5dd8454b55-75jct -n=cldinf-app -- sh
/ # curl 10.0.1.97:5432
```

Figure 37 Curl command on a Web Pod after Network Policy

5.7.3 API

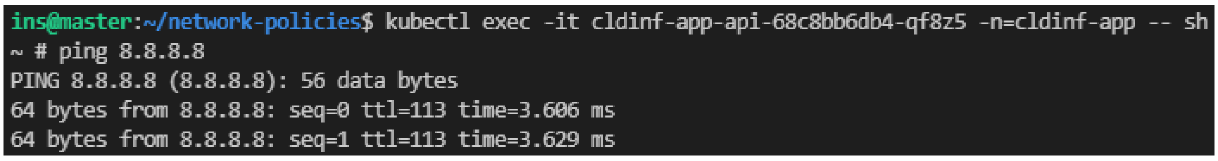
The API pods are only allowed to talk with the database and no other pod in the topology. At the moment we don’t have more than the Web pods but if there were more added in the future having an API that can connect to them without regulations could pose a problem.

To verify our network policy on the api-deployment we tried to ping to a web pod and to the google DNS. Both worked before applying the policy but not after:



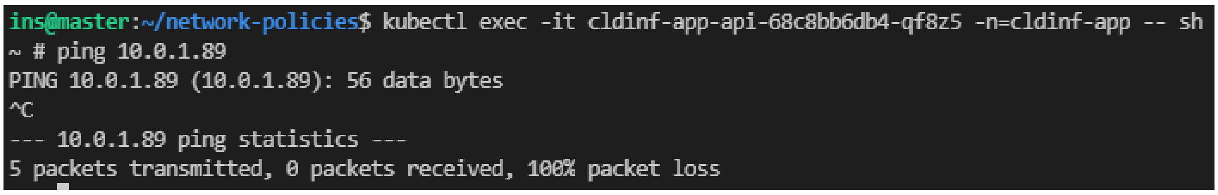
```
ins@master:~/network-policies$ kubectl exec -it cldinf-app-api-68c8bb6db4-qf8z5 -n=cldinf-app -- sh
~ # ping 10.0.1.89
PING 10.0.1.89 (10.0.1.89): 56 data bytes
64 bytes from 10.0.1.89: seq=0 ttl=63 time=1.389 ms
64 bytes from 10.0.1.89: seq=1 ttl=63 time=0.902 ms
```

Figure 38 Connection from API to a Web Pod before adding Policy



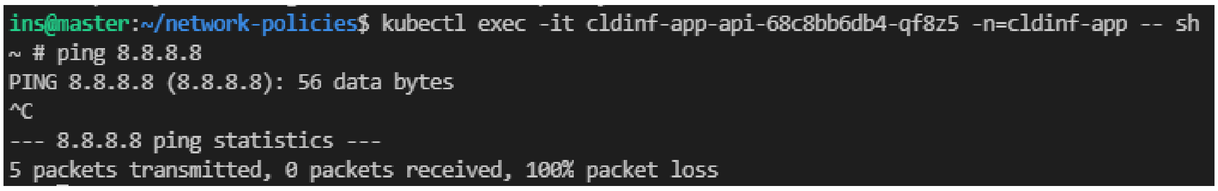
```
ins@master:~/network-policies$ kubectl exec -it cldinf-app-api-68c8bb6db4-qf8z5 -n=cldinf-app -- sh
~ # ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=113 time=3.606 ms
64 bytes from 8.8.8.8: seq=1 ttl=113 time=3.629 ms
```

Figure 39 Connection to Google DNS before adding Network Policy



```
ins@master:~/network-policies$ kubectl exec -it cldinf-app-api-68c8bb6db4-qf8z5 -n=cldinf-app -- sh
~ # ping 10.0.1.89
PING 10.0.1.89 (10.0.1.89): 56 data bytes
^C
--- 10.0.1.89 ping statistics ---
5 packets transmitted, 0 packets received, 100% packet loss
```

Figure 40 Connection from API to a Web Pod after adding Policy



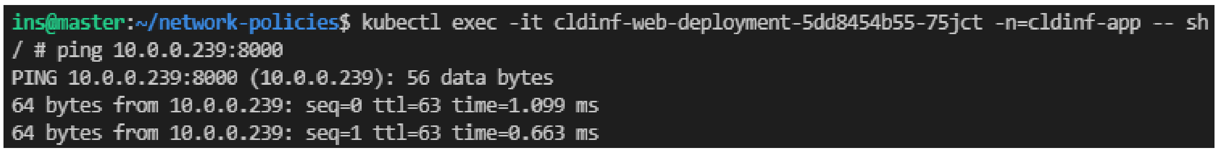
```
ins@master:~/network-policies$ kubectl exec -it cldinf-app-api-68c8bb6db4-qf8z5 -n=cldinf-app -- sh
~ # ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
^C
--- 8.8.8.8 ping statistics ---
5 packets transmitted, 0 packets received, 100% packet loss
```

Figure 41 Connection to Google DNS after adding Network Policy

5.7.4 Web

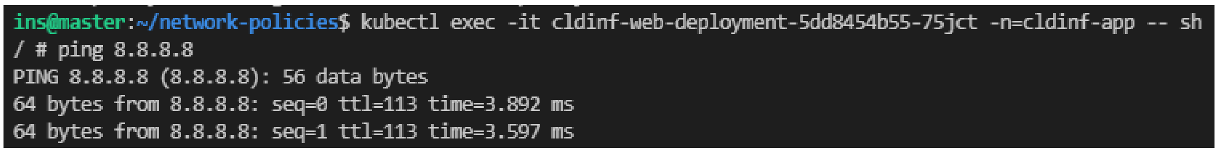
For the Web pods we planned to have them connect to the API pod and nowhere else. This was achieved by matching the label of the API-deployment in the “podSelector”.

To verify that the Web pods can’t connect to anything else but the API-pod we first did a reference ping to it with the port number as well as the google DNS. After applying the network policy, which denied all protocols but UDP and TCP to go out, we instead used curl to check the connection and got the correct response from the API but as expected, nothing from the google DNS ping.



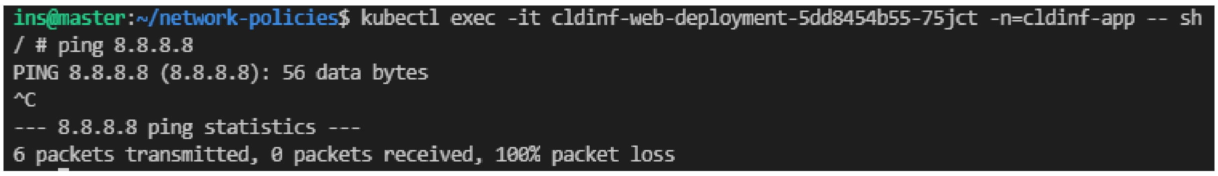
```
ins@master:~/network-policies$ kubectl exec -it cldinf-web-deployment-5dd8454b55-75jct -n=cldinf-app -- sh
/ # ping 10.0.0.239:8000
PING 10.0.0.239:8000 (10.0.0.239): 56 data bytes
64 bytes from 10.0.0.239: seq=0 ttl=63 time=1.099 ms
64 bytes from 10.0.0.239: seq=1 ttl=63 time=0.663 ms
```

Figure 42 Pinging API pod from a Web pod before Network Policy



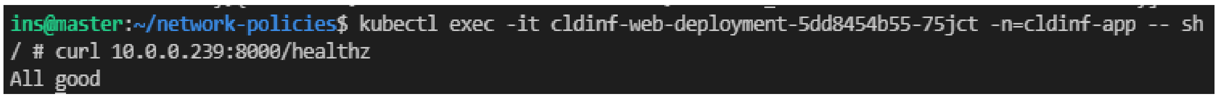
```
ins@master:~/network-policies$ kubectl exec -it cldinf-web-deployment-5dd8454b55-75jct -n=cldinf-app -- sh
/ # ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=113 time=3.892 ms
64 bytes from 8.8.8.8: seq=1 ttl=113 time=3.597 ms
```

Figure 43 Pinging Google DNS before adding Network Policy



```
ins@master:~/network-policies$ kubectl exec -it cldinf-web-deployment-5dd8454b55-75jct -n=cldinf-app -- sh
/ # ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
^C
--- 8.8.8.8 ping statistics ---
6 packets transmitted, 0 packets received, 100% packet loss
```

Figure 44 Pinging Google DNS after adding Network Policy



```
ins@master:~/network-policies$ kubectl exec -it cldinf-web-deployment-5dd8454b55-75jct -n=cldinf-app -- sh
/ # curl 10.0.0.239:8000/healthz
All good
```

Figure 45 Curl on the /healthz path (/time too big of an output) from a Web Pod

To further improve on our design of the network policy structure we also wanted to apply one for the ingress object that connects to the web pod but couldn’t get it to work in time for the submission.

To check if everything works with the network policies enabled we connected to the address: 10-18-9-7.sslip.io with our browser of choice and it showed us the site without any problems.

6 Troubleshooting Lab

For this part we were given a YAML file from a git repository[12] to find and fix build in mistakes.

After applying the YAML files from the given git repository we decided to first dump all the files into to a file. The files can be edited by using the command `kubectl edit <pod-name>` or dumping it to a new file, edit them and apply the new settings with `kubectl apply -f <pod-name>`.

6.1 Tshoot 1

The first part was to find three mistakes in the “tshoot-1” namespace. The problem was that the application in this namespace wasn’t available over Port 30080.

The first mistake we found after analysing the output of the `describe` command was that the application can’t pull the image because of a typo in the containers “image” tag (ngix instead of ngi**n**x”).

The second mistake was the missing label in the deployment. Since the service has as label selector “run: another-app” which wasn’t in the deployment, it couldn’t connect / select correctly. Adding the “label” tag in the deployments metadata and changing the “matchLabels” in the “spec” fixed this issue.

The last mistake we found was the incorrect selection of “targetPort” in the services’ YAML. Since the container has a nginx image behind it, which listens on Port 80 per default, we had to fix the connection by editing “targetPort” to be 80 instead of 8080. Changing the port on the deployment to 8080 wouldn’t have fixed the issue since the image behind the container still listens on Port 80.

6.2 Tshoot 2

After again looking at the describe commands information we saw that the problem was with either the CPU limitations or the “taint” which is affected by the “toleration” tag. We decided to first change the CPU limit to the same we used int the previous exercises (1 / 1000m) and this fixed it.

[12] https://raw.githubusercontent.com/INSRapperswil/k8s-tshoot-lab/main/tshoot-lab.yaml