NSO in Kubernetes and using Kubernetes Lab

June 14, 2019

Table of Contents

Introduction

This set of lab experiments will provide the student a good overview of how to run Network Service Orchestrator (NSO) as an application container in a Kubernetes (K8s) cloud native environment along with the benefits that may bring. We also review an innovative approach on how you can use NSO to YANG model your HELM charts such that NSO can drive your cloud native applications into Kubernetes.

The student will be provided with the step-by-step instruction to run the lab scenarios and we do not assume any prior hands-on experience with containers, Kubernetes or cloud native technologies. While we provide basic explanations, these labs are not meant to be a replacement for formal Kubernetes training. Students with some previous container and Kubernetes knowledge is the target audience.

Requirements

The table below outlines the requirements for this lab.

Table 1. Requirements

Required	Optional
• Laptop (MAC preferred) with Cisco AnyConnect® • Laptop Chrome browser • Laptop terminal emulation application for SSH (SecureCRT, iTerm, PuTTY, and so forth).	• Generic text editor (Sublime, Text Wrangler, Notepad)

Topology

This content includes preconfigured components to illustrate the scripted scenarios and features of the solution. You can use the IP address below and user account credentials to access a component.

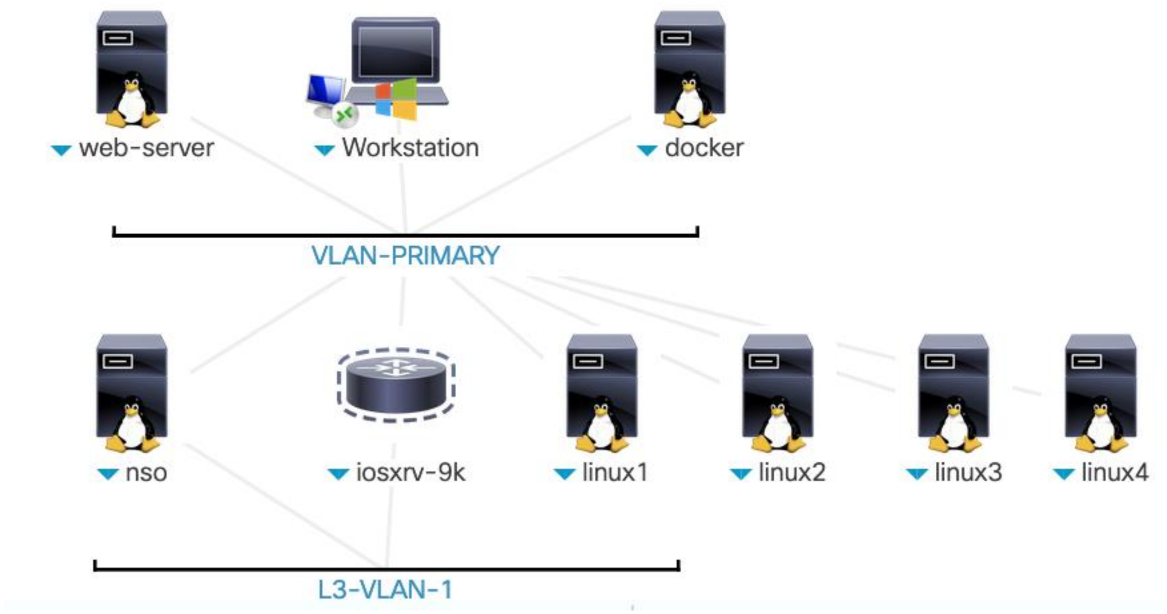


Figure 1 - dCloud Topology

Name	IP Address	Role	Protocol	Credentials
NSO	198.18.134.4		ssh	root C1sco12345
linux1	198.18.134.29	Master	ssh	root C1sco12345
linux2	198.18.134.30	Node	ssh	root C1sco12345
linux3	198.18.134.31	Node	ssh	root C1sco12345
linux4	198.18.134.32	Node	ssh	root C1sco12345
XRv Router	198.18.134.46		telnet	admin admin

NOTE: Cisco recommends that you use the AnyConnect VPN access method from your laptop to the dCloud environment and not the RDP-based Windows workstation (but this is an alternative if you do not have Anyconnect).

Get Started

For best performance, it is highly recommended to connect to the workstation with **Cisco AnyConnect VPN** (See next section "Explore the Environment" or [Show Me How]). As an alternate method you can try the Remote Desktop Client (**RDP**) [Show Me How] by expanding the "Workstation" device icon in the topology map and selecting RDP or use the following credentials in your own RDP laptop client. In this case you will RDP to the Windows Workstation and run all commands (SSH) from that environment.

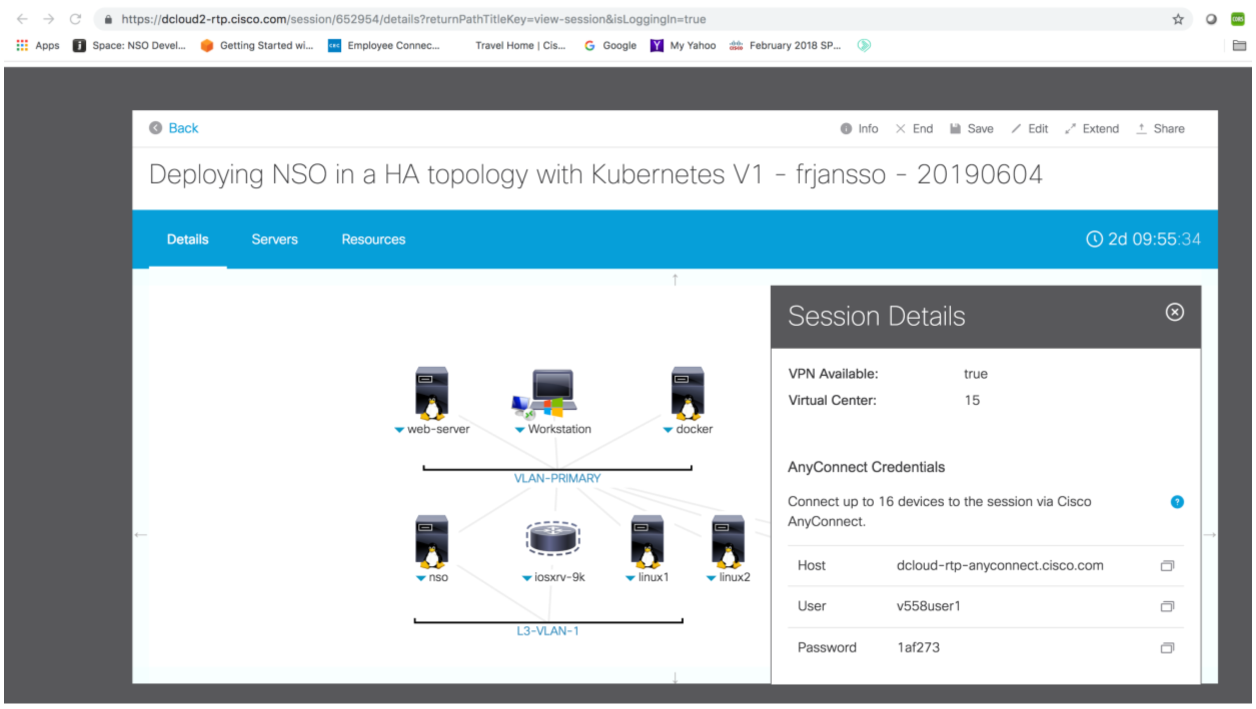
Workstation 1: **198.18.133.252**, Username: **administrator**, Password: **C1sco12345**.

Explore the Environment

1. Go to the Cisco dCloud site in your browser. Make sure you are in the right dCloud DC Region if your lab is not showing up under MyHub (check DC region top right).

NOTE: The Chrome browser application is recommended.

2. **View your Lab Session**: Select the view button to access the lab.
3. **Access dCloud with AnyConnect**: From the dCloud lab topology screen, select "**Details**" in the top left. This opens a pop-up window; scroll to "AnyConnect Credentials". On your laptop, open the AnyConnect VPN application and enter the "Host" information in the network connection box and after connecting, enter the dCloud provided "User" and "Password" credentials. Once connected, your laptop will be given an IP address from the dCloud lab subnet and will have full routing to all the IP addresses needed in the lab. You can now use your favorite terminal emulation client (iTerm, SecureCRT, PuTTY or just your MAC laptop Term window) to SSH or Telnet to the various devices and also your laptop browser to access any GUI screens.



https://dcloud2-rtp.cisco.com/session/652954/details?returnPathTitleKey=view-session&isLoggingIn=true

Deploying NSO in a HA topology with Kubernetes V1 - frjansso - 20190604

2d 09:55:34

Session Details

VPN Available: true
Virtual Center: 15

AnyConnect Credentials

Connect up to 16 devices to the session via Cisco AnyConnect.

Host	dcloud-rtp-anyconnect.cisco.com
User	v558user1
Password	1af273

Using and moving around the Kubernetes environment:

When you login into the master node (linux1), k8s is already installed along with Weave networking as the CNI (Container Networking Interface). The CNI is how Pods (containers) in the same cluster communicate with each other, over a private subnet space. For the lab, the container networking technology is not important.

To get an overview of the k8s system, look at the k8s nodes

```
# ssh root@198.18.134.29
root@198.18.134.29's password:C1sco12345
# kubectl get nodes

NAME     STATUS   ROLES    AGE   VERSION
linux1   Ready    master   44d   v1.13.4
linux2   Ready    <none>   44d   v1.13.4
linux3   Ready    <none>   44d   v1.13.4
linux4   Ready    <none>   44d   v1.13.4
```

Here we can see the master node and three worker nodes. For many commands you can add the "-o wide" option for more details:

```
# kubectl get nodes -o wide
NAME     STATUS   ROLES    AGE   VERSION   INTERNAL-IP     EXTERNAL-IP   OS-IMAGE                KERNEL-VERSION               CONTAINER-RUNTIME
linux1   Ready    master   47d   v1.13.4   198.18.134.29   <none>        CentOS Linux 7 (Core)   3.10.0-957.5.1.el7.x86_64   docker://18.9.3
linux2   Ready    <none>   47d   v1.13.4   198.18.134.30   <none>        CentOS Linux 7 (Core)   3.10.0-957.5.1.el7.x86_64   docker://18.9.3
linux3   Ready    <none>   47d   v1.13.4   198.18.134.31   <none>        CentOS Linux 7 (Core)   3.10.0-957.5.1.el7.x86_64   docker://18.9.3
linux4   Ready    <none>   47d   v1.13.4   198.18.134.32   <none>        CentOS Linux 7 (Core)   3.10.0-957.5.1.el7.x86_64   docker://18.9.3
```

Currently we only have a few k8s namespaces defined:

```
# kubectl get ns
NAME          STATUS   AGE
default       Active   44d
gluster       Active   43d
kube-public   Active   44d
kube-system   Active   44d
```

If we look in the kube-system namespace, we can see the system pods. *Pods* are the smallest deployable units of computing that can be created and managed in Kubernetes. Pods group one or more containers. All containers in a Pod are always co-located on the same host and co-scheduled and run in a shared context. A Pod has a single IP address on the container network.

```
# kubectl -n kube-system get pod
NAME                              READY   STATUS    RESTARTS   AGE
coredns-86c58d9df4-9hhkw          1/1     Running   1          44d
coredns-86c58d9df4-jml97          1/1     Running   1          44d
etcd-linux1                       1/1     Running   0          44d
kube-apiserver-linux1             1/1     Running   0          44d
kube-controller-manager-linux1    1/1     Running   0          44d
```

```
kube-proxy-7djkd                 1/1     Running   1          44d
kube-proxy-rvdxs                 1/1     Running   0          44d
kube-proxy-w8sw6                 1/1     Running   0          44d
kube-proxy-w975c                 1/1     Running   0          44d
kube-scheduler-linux1            1/1     Running   1          44d
tiller-deploy-5b7c66d59c-tszbx   1/1     Running   0          44d
weave-net-996fj                  2/2     Running   0          44d
weave-net-dx5n4                  2/2     Running   0          44d
weave-net-ssr2c                  2/2     Running   3          44d
weave-net-xv2gm                  2/2     Running   0          44d
```

TIP: You can also view all pods in all namespaces with the "—all-namespaces" parameter.

On most kubectl get commands, you can also add the -owide flag to get more information

```
# kubectl -n kube-system get pod -o wide
NAME                             READY   STATUS    RESTARTS   AGE   IP              NODE     NOMINATED NODE   READINESS GATES
coredns-86c58d9df4-9hhkw         1/1     Running   1          44d   10.45.0.3       linux2   <none>           <none>
coredns-86c58d9df4-jml97         1/1     Running   1          44d   10.45.0.4       linux2   <none>           <none>
etcd-linux1                      1/1     Running   0          44d   198.18.134.29   linux1   <none>           <none>
kube-apiserver-linux1            1/1     Running   0          44d   198.18.134.29   linux1   <none>           <none>
kube-controller-manager-linux1   1/1     Running   0          44d   198.18.134.29   linux1   <none>           <none>
kube-proxy-7djkd                 1/1     Running   1          44d   198.18.134.30   linux2   <none>           <none>
kube-proxy-rvdxs                 1/1     Running   0          44d   198.18.134.29   linux1   <none>           <none>
kube-proxy-w8sw6                 1/1     Running   0          44d   198.18.134.32   linux4   <none>           <none>
kube-proxy-w975c                 1/1     Running   0          44d   198.18.134.31   linux3   <none>           <none>
kube-scheduler-linux1            1/1     Running   1          44d   198.18.134.29   linux1   <none>           <none>
tiller-deploy-5b7c66d59c-tszbx   1/1     Running   0          44d   10.40.0.1       linux4   <none>           <none>
weave-net-996fj                  2/2     Running   0          44d   198.18.134.29   linux1   <none>           <none>
weave-net-dx5n4                  2/2     Running   0          44d   198.18.134.31   linux3   <none>           <none>
weave-net-ssr2c                  2/2     Running   3          44d   198.18.134.30   linux2   <none>           <none>
weave-net-xv2gm                  2/2     Running   0          44d   198.18.134.32   linux4   <none>           <none>
```

This also tells us, among other things, on what node the pod is running.
We can quickly test the k8s cluster by running a simple pod. Go to the /root/lab/test-k8s directory, there you'll find a *deployment.yaml* file

Note: Since we're going to push new Docker images, we need to have a Docker registry from where the k8s nodes can pull the image. This has already been setup for the lab, instructions on how to create a docker registry can be found at the end of the document.

```
# cd /root/lab/test-k8s
# cat deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-test
  labels:
    app: web
spec:
  replicas: 1
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
      - name: web
        image: fredrikjanssonse/devdays-web
        ports:
        - containerPort: 80

---
kind: Service
apiVersion: v1
metadata:
    name: web-test
spec:
  selector:
    app: web
  ports:
    - protocol: TCP
      port: 80
      nodePort: 30080
  type: NodePort
```

To deploy this in k8s

```
# kubectl apply -f deployment.yaml

deployment.apps/web-test created
service/web-test created
```

This will create a deployment and a NodePort service in the default namespace.

```
# kubectl get pod
NAME                       READY   STATUS    RESTARTS   AGE
web-test-fd4678575-pgpxw   1/1     Running   0          10m
```

```
# kubectl get svc web-test
NAME       TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
web-test   NodePort   10.98.129.59   <none>        80:30080/TCP   4m14s
```

Pay attention to the ports in the service, in this case the deployment file specifically asked for port 30080 (see the yaml file above).

Our sample application is a simple web server that returns a banner with the hostname and IP address of the pod. Since this is a NodePort service, you can access the webserver from any external Node IP in the cluster if you use the unique external port number. Try it with different Node IPs!

```
# curl http://linux1:30080
Banner: No Banner!
Host name: web-test-fd4678575-pgpxw
IP Addresses: 10.44.0.2
```

After verifying everything works, we can delete the pod and service.

```
# kubectl delete -f deployment.yaml
deployment.apps "web-test" deleted
service "web-test" deleted
```

GlusterFS

Gluster is a scalable, network-attached distributed file system that aggregates disk storage resources from multiple servers into a single global namespace.

When we deploy NSO later in the lab, it will utilize the cluster's replicated storage provided by Gluster for persistency of the NSO CDB. The installation of Gluster has already been completed and is not part of the lab.

To see what pods Gluster has installed, we can look at the pods in the gluster namespace. Each node has a virtual disk dedicated to Gluster. The three Gluster pods correspond to each of the nodes and serve the Gluster volume.

Heketi provides a RESTful management interface which can be used to manage the life cycle of GlusterFS volumes. With Heketi, cloud services like OpenStack Manila, Kubernetes, and OpenShift can dynamically provision GlusterFS volumes with any of the supported durability types.

The heketi pod is a pod we can use to get information on Gluster.

```
# kubectl -n gluster get pods
NAME                      READY   STATUS    RESTARTS   AGE
glusterfs-5snqf           1/1     Running   0          44d
```

```
glusterfs-fqzt5          1/1     Running   0          44d
glusterfs-jq92t          1/1     Running   0          44d
heketi-7495cdc5fd-tdg9v  1/1     Running   0          44d
```

We have also created a Gluster storage class for k8s volumes and volume claims.

```
# kubectl get sc
NAME                 PROVISIONER               AGE
glusterfs-storage    kubernetes.io/glusterfs   44d
```

K8s allows us to create persistent volumes claims to get a volume of a certain size. Later in the lab we will use this for NSO persistent storage.

We can also dig into Gluster and get some information on the volumes.

First, list the volumes, please note that the name of the heketi pod may differ on your system, please see the pod list (**kubectl -n gluster get pods**).

```
# kubectl -n gluster exec heketi-7495cdc5fd-tdg9v heketi-cli volume list
Id:0c79..    Cluster:b0005..    Name:heketidbstorage
```

Next, get information on that volume, replace the ID number in the below command with your ID.

```
# kubectl -n gluster exec heketi-7495cdc5fd-tdg9v heketi-cli volume info 0c79358512162e586f15acbd4fd492b8
Name: heketidbstorage
Size: 2
Volume Id: 0c79358512162e586f15acbd4fd492b8
Cluster Id: b0005db1c98552d324469559a448c89e
Mount: 198.18.134.31:heketidbstorage
Mount Options: backup-volfile-servers=198.18.134.32,198.18.134.30
Block: false
Free Size: 0
Reserved Size: 0
Block Hosting Restriction: (none)
Block Volumes: []
Durability Type: replicate
Distributed+Replica: 3
```

We can see that we have three replicas.

To test Gluster, we can manually test to create a persistent volume claim and make sure a dynamic persistent volume is created.

Please go to /root/lab/test-gluster, there you'll find persistent-volume-claim.yaml.

```
# cd /root/lab/test-gluster
# cat persistent-volume-claim.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: test-gluster
 annotations:
   volume.beta.kubernetes.io/storage-class: glusterfs-storage
spec:
 accessModes:
  - ReadWriteOnce
 resources:
   requests:
     storage: 5Gi
persistent-volume.yaml
```

This volume claim will ask for a 5 Gb volume from the “glusterfs-storage” storage class.

Let’s create the claim

```
# kubectl apply -f persistent-volume-claim.yaml
persistentvolumeclaim/test-gluster created
```

To see the claim, please note that the initial state is pending, but it should become bound after a short time period.

```
# kubectl get pvc
NAME           STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS        AGE
test-gluster   Bound    pvc-f7   5Gi        RWO            glusterfs-storage   3m23s
```

And the actual persistent volume created from the claim

```
# kubectl get pv
NAME     CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                  STORAGECLASS        REASON   AGE
pvc-f7   5Gi        RWO            Delete           Bound    default/test-gluster   glusterfs-storage            4m38s
```

And finally, let’s remove this volume claim.

```
# kubectl delete -f persistent-volume-claim.yaml
persistentvolumeclaim "test-gluster" deleted
```

After a couple of seconds, the dynamically generated volume should be deleted

```
# kubectl get pv
No resources found.
```

Helm

Helm is a tool for managing Kubernetes charts. Charts are packages of pre-configured Kubernetes resources. Helm streamlines installing and managing Kubernetes applications. Think of it like apt/yum/homebrew for Kubernetes.

The master node comes with Helm installed. To verify it's operational we can check the client and server version.

```
# helm version
Client: &version.Version{SemVer:"v2.13.0",
GitCommit:"79d07943b03aea2b76c12644b4b54733bc5958d6", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.13.0",
GitCommit:"79d07943b03aea2b76c12644b4b54733bc5958d6", GitTreeState:"clean"}
```

We can also test Helm by installing a test chart.

Go to the /root/lab/web-chart directory on the master node.

This directory contains a Helm chart for the same web service we used earlier.

We can deploy this helm chart

```
# cd /root/lab/web-chart

# helm upgrade --install helm-test .
Release "helm-test" does not exist. Installing it now.
NAME:   helm-test
LAST DEPLOYED: Thu May  2 16:25:48 2019
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/ConfigMap
NAME                DATA  AGE
helm-test-configmap  1     <invalid>

==> v1/Deployment
NAME           READY  UP-TO-DATE  AVAILABLE  AGE
helm-test-web  0/3    0           0          <invalid>

==> v1/Pod(related)
NAME                            READY  STATUS             RESTARTS  AGE
helm-test-web-6dfc7c7746-5lngk  0/1    ContainerCreating  0         <invalid>
```

```
helm-test-web-6dfc7c7746-bm7w4  0/1    ContainerCreating  0         <invalid>
helm-test-web-6dfc7c7746-jl2hh  0/1    ContainerCreating  0         <invalid>

==> v1/Service
NAME           TYPE      CLUSTER-IP      EXTERNAL-IP  PORT(S)       AGE
helm-test-web  NodePort  10.110.237.232  <none>       80:30080/TCP  <invalid>
```

This will create a number of k8s resources, including three webserver pods.

We can also see the helm-test when we list the helm releases.

```
# helm list
NAME            REVISION        UPDATED                         STATUS
CHART           APP VERSION     NAMESPACE
helm-test       1               Thu May  2 16:25:48 2019        DEPLOYED
web-0.1.0       1.0             default
```

View the webserver pods with the “-o wide” option to see the Node names they are deployed on, along with their CNI IP:

```
# kubectl get pods -o wide
NAME                             READY   STATUS    RESTARTS   AGE     IP          NODE     NOMINATED NODE   READINESS GATES
helm-test-web-6dfc7c7746-29kz7   1/1     Running   0          3m56s   10.45.0.5   linux2   <none>           <none>
helm-test-web-6dfc7c7746-kp5kd   1/1     Running   0          3m56s   10.44.0.2   linux3   <none>           <none>
helm-test-web-6dfc7c7746-ps4fp   1/1     Running   0          3m56s   10.40.0.2   linux4   <none>           <none>
```

Again, we can look at the services

```
# kubectl get svc
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
helm-test-web   NodePort    10.110.237.232   <none>        80:30080/TCP   103s
kubernetes      ClusterIP   10.96.0.1        <none>        443/TCP        44d
```

Again, we can use curl to verify the service is up

```
# curl http://linux1:30080
Banner: Welcome to the Lab!
Host name: helm-test-web-6dfc7c7746-bm7w4
IP Addresses: 10.44.0.2
```

If you run the curl command a couple of times, you’ll see that the CNI IP address and Pod hostname changes. K8s round robins between the three different pods.

We can also see the power of Helm. Let’s make a change to the values file, and then redeploy.

Let’s change the banner and replica count, please feel free to choose your own values.

```
# vim values.yaml
```

```
web:
  replicaCount: 10
  banner: "NSO Rocks!"

image:
  repository: fredrikjanssonse/devdays-web
  tag: latest
  pullPolicy: Always

nameOverride: ""
fullnameOverride: ""

service:
  type: NodePort
  port: 80
  nodePort: 30080
```

Let's upgrade the running release, by re-running helm upgrade. This will make Helm do an upgrade on the parts of our application that have changed.

Before running the upgrade, open a new window to the master node (linux1) and start the curl command via watch. You should see how helm and k8s perform a rolling upgrade of the pods and you'll get a mix of old and new banners until all ten new containers have been deployed.

```
# watch curl http://linux1:30080
Banner: Welcome to the Lab!
Host name: helm-test-web-6dfc7c7746-bm7w4
IP Addresses: 10.44.0.2

...

Banner: NSO Rocks!
Host name: helm-test-web-76c4dc9bc6-9plfd
IP Addresses: 10.44.0.7
```

Now perform the upgrade in the original window:

```
# helm upgrade --install helm-test .
Release "helm-test" has been upgraded. Happy Helming!
LAST DEPLOYED: Thu May  2 16:32:57 2019
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/ConfigMap
```

```
NAME                 DATA  AGE
helm-test-configmap  1     50s

==> v1/Deployment
NAME           READY  UP-TO-DATE  AVAILABLE  AGE
helm-test-web  3/10   0           3          50s

==> v1/Pod(related)
NAME                            READY  STATUS             RESTARTS  AGE
helm-test-web-6dfc7c7746-5d4qc  0/1    ContainerCreating  0         <invalid>
helm-test-web-6dfc7c7746-5lngk  1/1    Running            0         50s
helm-test-web-6dfc7c7746-bdkbm  0/1    ContainerCreating  0         <invalid>
helm-test-web-6dfc7c7746-bm7w4  1/1    Running            0         50s
helm-test-web-6dfc7c7746-jl2hh  1/1    Running            0         50s
helm-test-web-6dfc7c7746-lsl8t  0/1    Pending            0         <invalid>
helm-test-web-6dfc7c7746-mfmkf  0/1    Pending            0         <invalid>
helm-test-web-6dfc7c7746-tj984  0/1    Pending            0         <invalid>
helm-test-web-6dfc7c7746-xdsv2  0/1    Pending            0         <invalid>
helm-test-web-6dfc7c7746-z7zkk  0/1    Pending            0         <invalid>
helm-test-web-76c4dc9bc6-2vcsh  0/1    Pending            0         <invalid>
helm-test-web-76c4dc9bc6-48qpv  0/1    Pending            0         <invalid>
helm-test-web-76c4dc9bc6-9hx2x  0/1    Pending            0         <invalid>

==> v1/Service
NAME           TYPE      CLUSTER-IP      EXTERNAL-IP  PORT(S)       AGE
helm-test-web  NodePort  10.110.237.232  <none>       80:30080/TCP  50s
```

If your fast enough you can see the containers with the original banner message terminating.

```
# watch kubectl get pods -o wide
NAME                            READY   STATUS        RESTARTS   AGE     IP            NODE     NOMINATED NODE   READINESS GATES
helm-test-web-6dfc7c7746-29kz7  1/1     Terminating   0          9m58s   10.45.0.5     linux2   <none>           <none>
helm-test-web-6dfc7c7746-7lfg4  1/1     Terminating   0          33s     10.44.0.4     linux3   <none>           <none>
helm-test-web-6dfc7c7746-86gmw  1/1     Terminating   0          33s     10.45.0.7     linux2   <none>           <none>
helm-test-web-6dfc7c7746-k8mgw  1/1     Terminating   0          33s     10.44.0.5     linux3   <none>           <none>
helm-test-web-6dfc7c7746-kp5kd  1/1     Terminating   0          9m58s   10.44.0.2     linux3   <none>           <none>
helm-test-web-6dfc7c7746-ps4fp  1/1     Terminating   0          9m58s   10.40.0.2     linux4   <none>           <none>
helm-test-web-6dfc7c7746-sqxvk  1/1     Terminating   0          33s     10.40.0.3     linux4   <none>           <none>
helm-test-web-6dfc7c7746-tb56x  1/1     Terminating   0          33s     10.45.0.6     linux2   <none>           <none>
helm-test-web-f94459b66-5gqdn   1/1     Running       0          30s     10.40.0.6     linux4   <none>           <none>
helm-test-web-f94459b66-98stp   1/1     Running       0          33s     10.40.0.4     linux4   <none>           <none>
helm-test-web-f94459b66-bgdwr   1/1     Running       0          29s     10.45.0.9     linux2   <none>           <none>
helm-test-web-f94459b66-fwwvx   1/1     Running       0          23s     10.45.0.10    linux2   <none>           <none>
helm-test-web-f94459b66-mbjqf   1/1     Running       0          33s     10.40.0.5     linux4   <none>           <none>
helm-test-web-f94459b66-mpc4t   1/1     Running       0          33s     10.45.0.8     linux2   <none>           <none>
helm-test-web-f94459b66-q92n6   1/1     Running       0          25s     10.40.0.7     linux4   <none>           <none>
helm-test-web-f94459b66-qlqss   1/1     Running       0          33s     10.44.0.6     linux3   <none>           <none>
helm-test-web-f94459b66-rzp4w   1/1     Running       0          33s     10.44.0.3     linux3   <none>           <none>
helm-test-web-f94459b66-td7zg   1/1     Running       0          28s     10.44.0.7     linux3   <none>           <none>
```

Finally, we will remove this chart and all the resources it created. Verify that the Pods and services have been deleted.

```
# helm delete --purge helm-test
release "helm-test" deleted
```

NSO in a Container Lab

Create NSO base container with a simple package

For the NSO in container lab, we're going to build an NSO container based on an NSO system install. We will also add the XR NED to be able to connect to the XRv router that is part of the dCloud lab.

Go to /root/lab/nso-base directory

In this directory, you'll find a number of files. The most interesting file is the Dockerfile

```
# cd /root/lab/nso-base

# cat Dockerfile
FROM bitnami/minideb

ARG NSOVER

COPY nso-$NSOVER.linux.x86_64.installer.bin /tmp/nso
COPY packages/ncs-4.7.4-cisco-iosxr-7.11.1.tar.gz /tmp/xr.tar.gz

RUN apt-get update; \
    apt-get install -y openssh-client; \
    sh /tmp/nso --system-install; \
    rm -rf /opt/ncs/current/var/ncs/webui; \
    rm -rf /opt/ncs/current/doc; \
    rm -rf /opt/ncs/current/man; \
    rm -rf /opt/ncs/current/examples.ncs; \
    rm -rf /opt/ncs/current/include; \
    rm -rf /opt/ncs/current/packages; \
    rm -rf /opt/ncs/current/support; \
    rm -rf /opt/ncs/current/src/aaa; \
    rm -rf /opt/ncs/current/src/build; \
    rm -rf /opt/ncs/current/src/cli; \
    rm -rf /opt/ncs/current/src/configuration_policy; \
    rm -rf /opt/ncs/current/src/errors; \
    rm -rf /opt/ncs/current/src/ncs_config; \
    rm -rf /opt/ncs/current/src/netconf; \
    rm -rf /opt/ncs/current/src/package-skeletons; \
    rm -rf /opt/ncs/current/src/project-skeletons; \
    rm -rf /opt/ncs/current/src/snmp; \
    rm -rf /opt/ncs/current/src/tools; \
    rm -rf /opt/ncs/current/src/yang; \
    rm -rf /opt/ncs/current/lib/ncs/lib/confdc; \
    rm -rf /opt/ncs/current/lib/ncs-project; \
    rm -rf /opt/ncs/current/lib/pyang; \
    rm -rf /opt/ncs/current/erlang \
    rm -rf /opt/ncs/current/netsim/confd/src/confd/pyapi/doc \
    rm -rf /opt/ncs/current/netsim/confd/erlang/econfd/doc \
    rm -rf /opt/ncs/current/src/ncs/pyapi/doc

RUN tar -C /var/opt/ncs/packages -xf /tmp/xr.tar.gz; \
    rm -rf /var/opt/ncs/packages/cisco-iosxr/doc; \
    rm -rf /var/opt/ncs/packages/cisco-iosxr/netsim; \
    rm -rf /var/opt/ncs/packages/cisco-iosxr/src

FROM bitnami/minideb

RUN install_packages openssh-client default-jre-headless python; \
    echo '. /opt/ncs/current/ncsrc' >> /root/.bashrc; \
    rm -rf /tmp/* /var/tmp/* /var/lib/{apt,dpkg,cache,log}/; \
    mkdir /var/log/ncs; \
    groupadd ncsadmin; \
    useradd admin -m -g ncsadmin; \
    echo "admin:admin" | chpasswd; \
    rm -rf /usr/share/doc; \
    mkdir /cdb-init

COPY --from=0 /etc/ncs /etc/ncs/
COPY --from=0 /var/opt/ncs /var/opt/ncs/
COPY --from=0 /opt/ncs /opt/ncs/
COPY --from=0 /etc/init.d/ncs /etc/init.d/.
# COPY packages/* /opt/

COPY run-nso.sh /
# COPY java.xml /var/opt/ncs/cdb/
# COPY admin.xml /var/opt/ncs/cdb/
COPY java.xml /cdb-init/
COPY admin.xml /cdb-init/
COPY ncs.conf /etc/ncs/ncs.conf
COPY xr.cli /home/admin/.

EXPOSE 80 830 2022 2023 4569

CMD ["/run-nso.sh"]
```

This is a two-stage file, in the first stage NSO is installed and cleaned of files that are not necessary for running NSO in a container (docs, compilers etc.).

The second stage copies the minified NSO installation and adds the project specific files, e.g. the XR NED.

To build the Docker image, simply run

```
# make image
...
Successfully tagged 198.18.134.29.nip.io:5000/nso:latest
docker tag "198.18.134.29.nip.io:5000/nso":latest
"198.18.134.29.nip.io:5000/nso":4.7.4
```

This will generate a docker image.

```
# docker images
REPOSITORY                             TAG                 IMAGE ID
CREATED              SIZE
198.18.134.29.nip.io:5000/nso          4.7.4               5ec8a9ef209b        About
a minute ago   376MB
198.18.134.29.nip.io:5000/nso          latest              5ec8a9ef209b        About
a minute ago   376MB
...
```

To make this image available to all nodes, we will push it to the repository.

```
# make push

docker push "198.18.134.29.nip.io:5000/nso":4.7.4
...
latest: digest:
sha256:834b5381ea9bd7c7858aa9e9ccbdf78759b966e9d8fb3e39f92aa78d24aa7ef0 size: 2408
```

Run NSO base container with docker

With the image ready, we can take it for a spin in plain docker.

```
# docker run -d --rm --name nso 198.18.134.29.nip.io:5000/nso

5a5b7b9017d7b6e2834010f78a051c9360c8a351b6524b2ed053b9b20f98c1da
```

Please note the --rm flag, this means that Docker will automatically remove the container after we stop it.

You can verify the container is running with the “docker ps” command:

```
# docker ps
CONTAINER ID        IMAGE                              COMMAND
CREATED             STATUS              PORTS
NAMES
e034386e8f8c        198.18.134.29.nip.io:5000/nso   "/run-nso.sh"           3
seconds ago       Up 2 seconds        80/tcp, 830/tcp, 2022-2023/tcp, 4569/tcp
nso
```

With the container running, we can attach to it

NOTE! It can take a could take a short while for NSO to come up.

```
# docker exec -it nso bash

root@5a5b7b9017d7:/# ncs_cli -Cu admin

admin connected from 127.0.0.1 using console on 5a5b7b9017d7

admin@ncs# show packages package oper-status

PACKAGE
                PROGRAM                                                           META     FILE
                CODE     JAVA           BAD NCS  PACKAGE  PACKAGE  CIRCULAR    DATA     LOAD   ERROR
NAME        UP  ERROR    UNINITIALIZED  VERSION  NAME     VERSION  DEPENDENCY  ERROR    ERROR  INFO
-----------------------------------------------------------------------------------------------------
cisco-iosxr  X   -        -              -        -        -        -           -        -      -

admin@ncs#
```

We can add the XR router to NSO by loading some configuration manually

```
admin@ncs# config

Entering configuration mode terminal
admin@ncs(config)# load merge /home/admin/xr.cli
Loading.
256 bytes parsed in 0.04 sec (5.60 KiB/sec)

admin@ncs(config)# commit
Commit complete.

admin@ncs(config)# end
```

If we sync from the device, we should see the config

```
admin@ncs# devices device xr sync-from
result true
```

```
admin@ncs# show running-config devices device xr config
devices device xr
 config
...
```

That means that NSO is now running in a docker container.

Before we move on, let's clean this up.

First exit the container

```
admin@ncs# exit
root@5a5b7b9017d7:/# exit
exit
```

Then we'll stop the container

```
# docker stop nso
nso
```

Since we previously started the container with the --rm flag, docker automatically cleans up when we stop the container.

With that we can move on to running NSO in k8s.

Running the NSO base container as simple k8s StatefulSet

The next step is to run NSO as a stateful set in k8s. StatefulSets maintain a sticky identity for each of their Pods. These pods are created from the same spec but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling. StatefulSets are valuable for applications that require stable persistent storage.

In the /root/lab/nso-ss directory, you'll find a YAML file containing a stateful set and a service (ssh to NSO).

```
# cd /root/lab/nso-ss

# cat nso-ss.yaml
apiVersion: v1
kind: Service
metadata:
  name: nso-ssh
  labels:
    app: nso
spec:
  type: NodePort
  ports:
  - port: 2024
    name: ssh
    nodePort: 30024
  selector:
    app: nso
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nso
spec:
  selector:
    matchLabels:
      app: nso
  serviceName: "nso-ssh"
  replicas: 1
  template:
    metadata:
      labels:
        app: nso
    spec:
      containers:
      - name: nso
        image: 198.18.134.29.nip.io:5000/nso
        imagePullPolicy: Always
        ports:
        - containerPort: 2024
          name: ssh
        readinessProbe:
          tcpSocket:
            port: 2024
          initialDelaySeconds: 5
          periodSeconds: 5
        livenessProbe:
          tcpSocket:
            port: 2024
          initialDelaySeconds: 15
          periodSeconds: 20
```

The readiness probe will try to connect to the ssh port, when it becomes available the pod is marked as ready (see e.g. kubectl get pods, READY 0/1, means that this pod is not yet ready).

The liveness probe will continuously connect to the pod to make sure the pod is alive. If the pod stops responding, k8s can take different actions depending on the type of application.

Let's deploy this

```
# kubectl apply -f nso-ss.yaml
service/nso-ssh created
statefulset.apps/nso created
```

```
# kubectl get statefulset
NAME   READY   AGE
nso    1/1     99s
```

```
# kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
nso-0   1/1     Running   0          60s
```

```
# kubectl get svc
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP          45d
nso-ssh      NodePort    10.109.46.217   <none>        2024:30024/TCP   2m20s
```

We can now ssh into NSO using the nso-ssh nodeport service.

```
# ssh -p 30024 admin@linux1
admin@linux1's password: admin

admin connected from 10.32.0.1 using ssh on nso-0
admin@ncs#
```

Tip: You can also ssh from your laptop client if you VPNed into the dcloud lab, use any of the Node IP's instead of "linux1".

Let's add the device back and sync from.

```
admin@ncs# config
Entering configuration mode terminal

admin@ncs(config)# load merge xr.cli
Loading.
256 bytes parsed in 0.03 sec (6.38 KiB/sec)

admin@ncs(config)# commit
Commit complete.
```

```
admin@ncs(config)# devices device xr sync-from
result true

admin@ncs(config)# end
admin@ncs# show devices list
NAME  ADDRESS        DESCRIPTION  NED ID        ADMIN STATE
-----------------------------------------------------------
xr    198.18.134.46  -            cisco-ios-xr  unlocked

admin@ncs# exit
Connection to linux1 closed.
#
```

NSO Re-startability with K8s

This is all good... but let's say NSO crashes now. We can simulate that by running docker kill on the pod.

Let's find out what node NSO is running, please note that it may be a different pod on your system!

```
# kubectl get pod -o wide
NAME    READY   STATUS    RESTARTS   AGE     IP          NODE     NOMINATED NODE
READINESS GATES
nso-0   1/1     Running   0          5m22s   10.40.0.2   linux4   <none>
<none>
```

ssh to that pod, and docker kill the NSO process ID. This ID will be unique for each of you.

```
# ssh linux4

# docker ps | grep run-nso
db2e828c4524        198.18.134.29.nip.io:5000/nso    "/run-nso.sh"          5 minutes ago       Up 5 minutes

# docker kill db2e828c4524
db2e828c4524

# exit
logout
Connection to linux4 closed.
```

Running docker kill, effectively kills the NSO process without proper shutdown.

Now, if we look the pods

```
# kubectl get pod
NAME     READY    STATUS     RESTARTS    AGE
nso-0    1/1      Running    1           6m49s
```

We can see that the NSO pod has been restarted automatically (RESTARTS=1) by k8s, this is really great!

Let's jump back into NSO

```
# ssh -p 30024 admin@linux1
admin@linux1's password: admin

admin connected from 10.32.0.1 using ssh on nso-0
admin@ncs# show devices list
NAME  ADDRESS  DESCRIPTION  NED ID  ADMIN STATE
-----------------------------------------------
admin@ncs#
```

Not, good... k8s brought back NSO, but CDB is empty. This is not surprising; we ran the container without any persistent storage.

Before we move on, please delete this deployment from the master node, make sure you're in the /root/lab/nso-ss directory.

```
admin@ncs# exit
# kubectl delete -f nso-ss.yaml
service "nso-ssh" deleted
statefulset.apps "nso" deleted
```

Run NSO container with replicated storage

To address the problem with crashing NSO, we're going to mount the most important directories using persistent volumes. The two most important directories are these two

- CDB directory (/var/opt/ncs/cdb)
- State directory (/var/opt/ncs/state)

In certain cases, you may want to keep e.g. logs or rollbacks, but for this lab we'll focus on the two above.

Navigate to /root/lab/nso-pv

```
# cd /root/lab/nso-pv/

# cat nso.yaml
---
apiVersion: v1
kind: Service
metadata:
  name: nso-ssh
  labels:
    app: nso
spec:
  type: NodePort
  ports:
  - port: 2024
    name: ssh
    nodePort: 30024
  selector:
    app: nso
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nso
spec:
  selector:
    matchLabels:
      app: nso
  serviceName: "nso-ssh"
  replicas: 1
  template:
    metadata:
      labels:
        app: nso
    spec:
      containers:
      - name: nso
        image: 198.18.134.29.nip.io:5000/nso
        imagePullPolicy: Always
        ports:
        - containerPort: 2024
          name: ssh
        readinessProbe:
          tcpSocket:
            port: 2024
          initialDelaySeconds: 5
          periodSeconds: 5
        livenessProbe:
          tcpSocket:
            port: 2024
          initialDelaySeconds: 15
          periodSeconds: 20
        volumeMounts:
          - name: cdb
            mountPath: /var/opt/ncs/cdb
          - name: state
            mountPath: /var/opt/ncs/state
  volumeClaimTemplates:
    - metadata:
        name: cdb
      spec:
        accessModes: [ "ReadWriteOnce" ]
        storageClassName: glusterfs-storage
        resources:
          requests:
            storage: 5Gi
    - metadata:
        name: state
      spec:
        accessModes: [ "ReadWriteOnce" ]
        storageClassName: glusterfs-storage
        resources:
          requests:
            storage: 1Gi
```

The new interesting parts are the volume mounts for CDB and the state. K8s will get these volumes from the glusterfs-storage.

Again, let’s apply this

```
# kubectl apply -f nso.yaml
service/nso-ssh created
statefulset.apps/nso created
```

Now if we look at volume claims and volumes, we see that this deployment has requested (and after a while received) two volumes.

```
# kubectl get pvc,pv
NAME                               STATUS   VOLUME     CAPACITY   ACCESS MODES   STORAGECLASS        AGE
persistentvolumeclaim/cdb-nso-0    Bound    pvc-fb..   5Gi        RWO            glusterfs-storage   100s
persistentvolumeclaim/state-nso-0  Bound    pvc-fb..   1Gi        RWO            glusterfs-storage   100s

NAME                        CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                 STORAGECLASS        REASON   AGE
persistentvolume/pvc-..     5Gi        RWO            Delete           Bound    default/cdb-nso-0     glusterfs-storage            94s
persistentvolume/pvc-fb..   1Gi        RWO            Delete           Bound    default/state-nso-0   glusterfs-storage            88s
```

Now, let’s get back into NSO and add the device.

If you get an error (ssh: connect to host linux1 port 30024: Connection refused), check the pods and make sure NSO is ready.

```
# ssh -p 30024 admin@linux1
admin@linux1's password: admin

admin connected from 10.32.0.1 using ssh on nso-0

admin@ncs# config
Entering configuration mode terminal

admin@ncs(config)# load merge xr.cli
Loading.
256 bytes parsed in 0.04 sec (5.39 KiB/sec)

admin@ncs(config)# commit
Commit complete.

admin@ncs(config)# end
admin@ncs# devices sync-from
sync-result {
    device xr
    result true
}

admin@ncs# show devices list
NAME  ADDRESS        DESCRIPTION  NED ID        ADMIN STATE
-----------------------------------------------------------
xr    198.18.134.46  -            cisco-ios-xr  unlocked

admin@ncs# exit
Connection to linux1 closed.
```

First, we are going to repeat what we did above, i.e. find which node is running NSO and do a docker kill NSO. The values below in red will be different for your environment.

```
# kubectl get pod -owide
NAME    READY   STATUS    RESTARTS   AGE     IP          NODE     NOMINATED NODE   READINESS GATES
nso-0   1/1     Running   0          2m55s   10.44.0.2   linuxX   <none>           <none>

# ssh linuxX
Last login: Sun May  5 20:25:22 2019 from linux1

# docker ps | grep run-nso
fa955fa29593        198.18.134.29.nip.io:5000/nso   "/run-nso.sh" ....
```

```
# docker kill fa955fa29593
fa955fa29593

# exit
logout
Connection to linux3 closed.
```

Make sure NSO is ready

```
# watch kubectl get statefulset

NAME    READY    AGE
nso     1/1      6m13s
```

Now if we ssh back into NSO and check the devices

```
# ssh -p 30024 admin@linux1
admin@linux1's password: admin
sh
admin connected from 10.32.0.1 using ssh on nso-0

admin@ncs# show devices list
NAME   ADDRESS        DESCRIPTION  NED ID        ADMIN STATE
-----------------------------------------------------------
xr     198.18.134.46  -            cisco-ios-xr  unlocked

admin@ncs# exit
Connection to linux1 closed.
#
```

That's all good, CDB survives a simple kill of the NSO process.

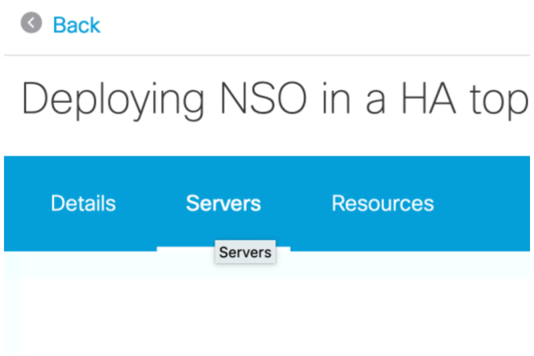
The next thing to test is loss of a k8s node. We're going to find out what node NSO is running on. We'll stop that node using the dCloud UI.

First let's see what node NSO runs on

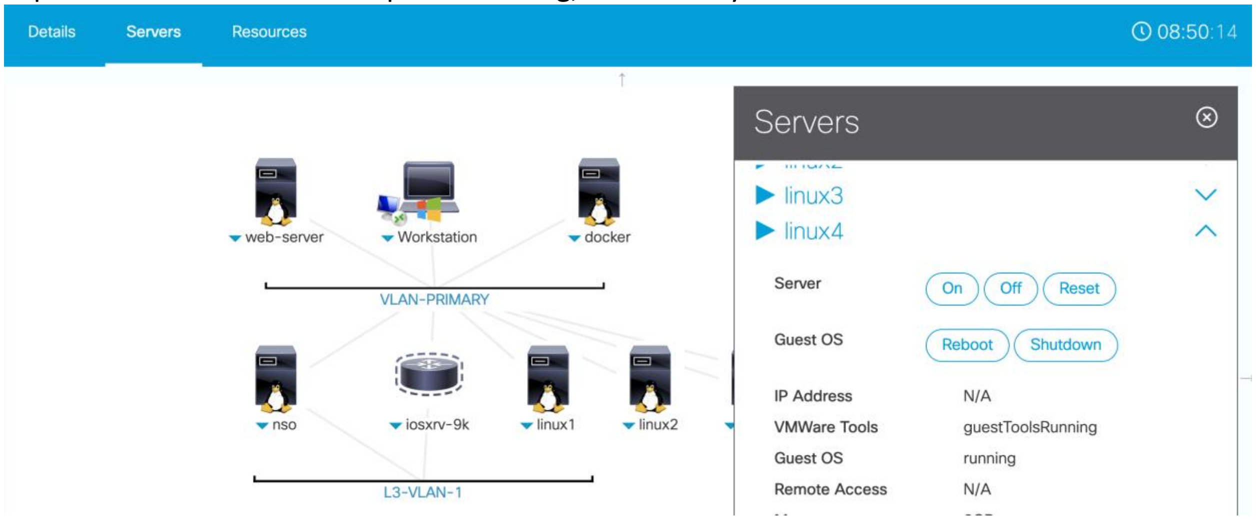
```
# kubectl get pod -o wide
NAME     READY    STATUS     RESTARTS    AGE     IP           NODE     NOMINATED NODE
READINESS GATES
nso-0    1/1      Running    0           5m4s    10.40.0.2    linuxX   <none>
<none>
```

NOTE! On my system it's linux3, in the lab it might be a different host.

Go to the dCloud UI, and click Servers

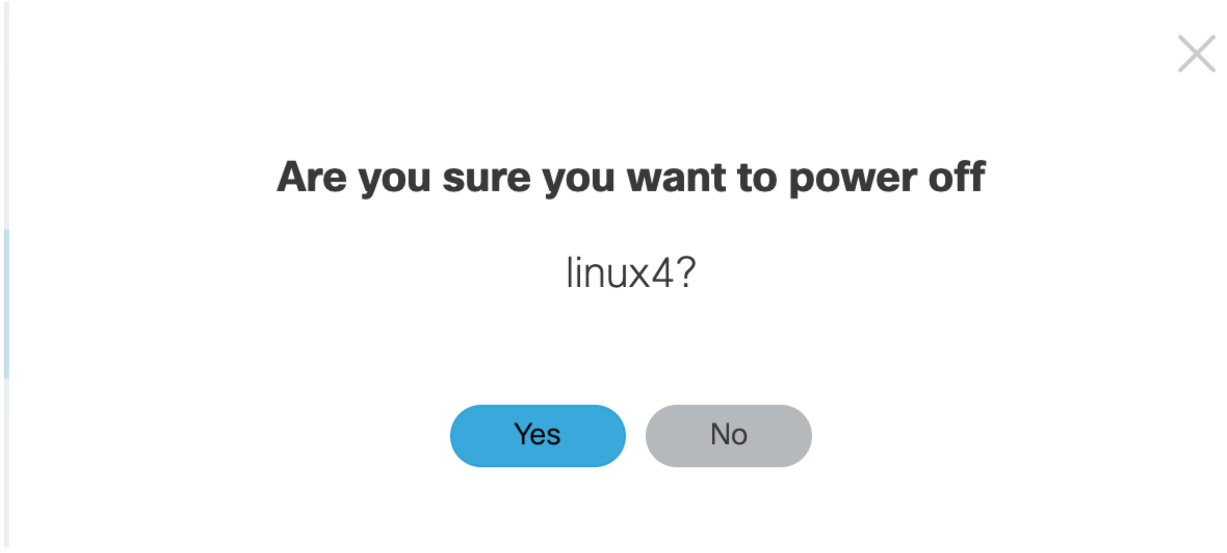


Expand the server where the pod is running, linux4 in my case.



IP Address N/A
VMWare Tools guestToolsRunning
Guest OS running
Remote Access N/A

Click the “Off” button and then Yes.



Are you sure you want to power off

linux4?

Now go back and check the nodes, after a while the stopped node will become NotReady.

```
# watch kubectl get nodes
```

```
NAME     STATUS     ROLES    AGE   VERSION
linux1   Ready      master   49d   v1.13.4
linux2   Ready      <none>   49d   v1.13.4
linux3   Ready      <none>   49d   v1.13.4
linux4   NotReady   <none>   49d   v1.13.4
```

And if we check the stateful set, you'll see that it will be stuck in not ready state.

```
# kubectl get statefulset
NAME   READY   AGE
nso    0/1     8m28s
```

NOTE! Since we deployed NSO as a **stateful set**, k8s will not automatically migrate the pod from the stopped node. The recommended way of forcing a migration, is to simply delete the pod.

```
# kubectl delete pods nso-0 --grace-period=0 --force
warning: Immediate deletion does not wait for confirmation that the running
resource has been terminated. The resource may continue to run on the cluster
indefinitely.
pod "nso-0" force deleted
```

Make sure NSO is running on a new node

```
# kubectl get pod -owide
NAME    READY   STATUS    RESTARTS   AGE   IP          NODE     NOMINATED NODE   READINESS GATES
nso-0   0/1     Running   0          14s   10.44.0.2   linux3   <none>           <none>
```

NOTE Make sure you wait until NSO is ready

Now if we ssh back into NSO and check the devices

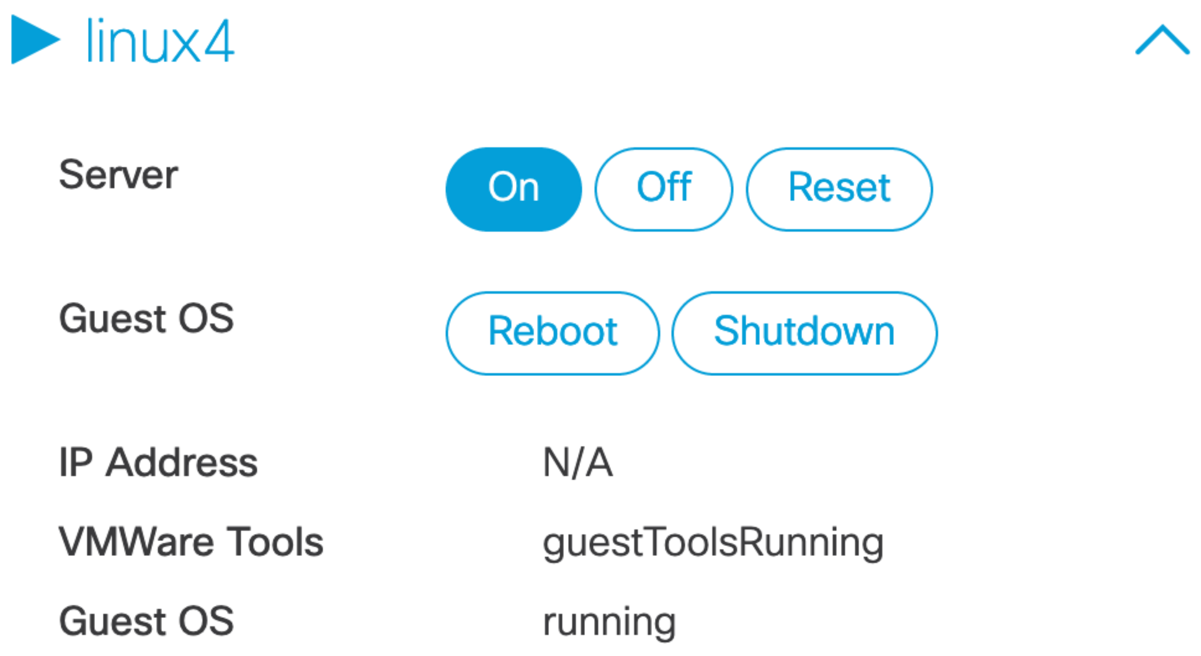
```
# ssh -p 30024 admin@linux1
admin@linux1's password: admin
sh
admin connected from 10.32.0.1 using ssh on nso-0

admin@ncs# show devices list
NAME  ADDRESS        DESCRIPTION  NED ID        ADMIN STATE
-----------------------------------------------------------
xr    198.18.134.46  -            cisco-ios-xr  unlocked

admin@ncs# devices device xr check-sync
result in-sync
```

Success! CDB survived the migration from one node to another.

Before moving on, make sure you start the node you previously stopped by clicking the On button and then Yes in the popup.



IP Address N/A
VMWare Tools guestToolsRunning
Guest OS running

And delete the deployment

```
admin@ncs# exit
# kubectl delete -f nso.yaml
service "nso-ssh" deleted
statefulset.apps "nso" deleted
```

We can leave the persistent volumes in place as we'll use them in the next step.

You can also verify that your nodes come online after a short while.

```
# kubectl get nodes
NAME     STATUS   ROLES    AGE   VERSION
linux1   Ready    master   49d   v1.13.4
linux2   Ready    <none>   49d   v1.13.4
linux3   Ready    <none>   49d   v1.13.4
linux4   Ready    <none>   49d   v1.13.4
```

Run the NSO container with Helm

The last step is to reproduce the above using helm.

A helm chart has been created in /root/lab/nso-chart directory.

Please take a look at values.yaml and templates/statefulset.yaml.

To deploy this helm chart

```
# cd /root/lab/nso-chart

# helm upgrade --install nso .
Release "nso" does not exist. Installing it now.
NAME:   nso
LAST DEPLOYED: Thu May  2 21:20:58 2019
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Pod(related)
NAME   READY  STATUS   RESTARTS  AGE
nso-0  0/1    Pending  0         <invalid>

==> v1/Service
NAME  TYPE      CLUSTER-IP    EXTERNAL-IP  PORT(S)         AGE
nso   NodePort  10.97.68.119  <none>       2024:30024/TCP  <invalid>

==> v1/StatefulSet
NAME  READY  AGE
nso   0/1    <invalid>
```

After a short while, you can SSH into NSO and you should see the device onboarded. This is because our Helm charts reuses the previously created volumes.

```
# ssh -p 30024 admin@linux1
admin@linux1's password: admin

admin connected from 10.32.0.1 using ssh on nso-0

admin@ncs# show devices list
NAME  ADDRESS        DESCRIPTION  NED ID        ADMIN STATE
-----------------------------------------------------------
xr    198.18.134.46  -            cisco-ios-xr  unlocked

admin@ncs# exit
Connection to linux1 closed.
```

And finally we can delete the chart

```
# helm delete --purge nso
release "nso" deleted
```

Using NSO to deploy your k8s Application Lab

Let's now change the topic from working with the NSO app in a container, to how NSO can deploy K8s applications directly (usually these apps would be Network Functions and part of a larger NSO service, but that is not a requirement). For the second part of the lab, we will look at using NSO to drive a cloud native application in K8s. This can be done using NSO in a container or not. In our lab environment we will use an NSO system external to K8s and not in a container since that is a typical deployment model. It will also show that how you deploy NSO is not important to how NSO can integrate with K8s Helm to drive cloud native applications.

Run the web server using Helm (again)

NOTE! Please login into the NSO provided by dCloud in another terminal. This is the dCloud lab topology NSO, please find IP address and credentials in the table at the beginning of this document.

Please navigate to /root/lab-nso/web-chart directory on the NSO machine, where you will see our sample helm chart.

Let's take another look at the values file.

```
$ ssh root@198.18.134.4
root@198.18.134.4's password: C1sco12345
Last login: Sun May  5 20:03:05 2019 from 10.16.6.50
#
# cd /root/lab-nso/web-chart
# cat values.yaml

web:
  replicaCount: 3
  banner: "Welcome to the Lab!"

image:
  repository: fredrikjanssonse/devdays-web
  tag: latest
  pullPolicy: Always

nameOverride: ""
fullnameOverride: ""

service:
  type: NodePort
  port: 80
  nodePort: 30080
```

The NSO VM also has the Helm client installed. Let's try to launch the Helm chart from the NSO host.

```
# cd /root/lab-nso

# helm install --name web web-chart
NAME:   web
LAST DEPLOYED: Mon Jun 10 18:27:13 2019
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/ConfigMap
NAME           DATA  AGE
web-configmap  1     2s

==> v1/Deployment
NAME  READY  UP-TO-DATE  AVAILABLE  AGE
web   0/3    3           0          2s

==> v1/Pod(related)
NAME                  READY  STATUS             RESTARTS  AGE
web-597d88547b-bpbrz  0/1    ContainerCreating  0         2s
web-597d88547b-s6h5g  0/1    ContainerCreating  0         2s
web-597d88547b-xwjsg  0/1    ContainerCreating  0         2s

==> v1/Service
NAME  TYPE      CLUSTER-IP     EXTERNAL-IP  PORT(S)       AGE
web   NodePort  10.107.148.90  <none>       80:30080/TCP  2s
```

Please note, this will launch the chart on the k8s cluster, not on the NSO VM

```
# curl http://198.18.134.29:30080
Banner: Welcome to the Lab!
Host name: web-597d88547b-s6h5g
IP Addresses: 10.44.0.3
```

Finally, let's delete the helm chart

```
# helm delete --purge web
release "web" deleted
```

In this lab we will look at how we can use NSO to YANG model the HELM deployment values in this chart and then launch instances of this chart from NSO and then dynamically make changes to the values such as banner and the replica count.

Local NSO

On the NSO host, you'll find a local install of the NSO application.

Navigate to the /root/lab-nso/nso-helm folder and start NSO if it is not running.

```
# cd /root/lab-nso/nso-helm

# ncs
```

If you look in the packages directory, you'll see two packages

```
# cd packages
# ls -l
total 8
drwxr-xr-x. 7 501 games 4096 May  3 11:55 cisco-helm
drwxr-xr-x. 4 501 games 4096 May  3 13:33 web-server
```

cisco-helm
This is the generic package that connects NSO to Helm created by Cisco. Under the hood it uses the helm client.

web-server
This is the package that maps YANG to our example web-server Helm chart. When using this approach in your non-dcloud customer environment, you will create this package to match the values in your K8s application's HELM chart. Please take a look at the YANG model for the example:

```
# less web-server/src/yang/web-server.yang
module web-server {
  namespace "http://com/example/webserver";
  prefix web-server;

  import ietf-inet-types {
    prefix inet;
  }
  import tailf-ncs {
    prefix ncs;
  }

  list web {
    key name;

    leaf name {
      type string;
    }
    leaf banner {
      type string;
    }

    leaf replica-count {
      default 1;
      type int16 {
        range "1..10";
      }
    }
  }
}
```

The YANG model is simply a list named web, with three leafs: name, banner and replica-count. If you have a more complex values file, you can tailor the YANG model accordingly.

If you scroll back up, you'll see that these YANG variables map to several of the items in the helm chart values.yaml file:

```
web:                                                list web
  replicaCount: 3                         →         leaf replica-count
  banner: "Welcome to the Lab!"                     leaf banner
```

The cisco-helm package automatically converts dashes to camelCase, i.e. replica-count becomes replicaCount.

Jump into the CLI and check the packages are up:

```
# ncs_cli -Cu admin

# show packages package oper-status

PACKAGE
                PROGRAM                                                                 META     FILE
                CODE     JAVA           BAD NCS  PACKAGE  PACKAGE  CIRCULAR    DATA     LOAD   ERROR
NAME        UP  ERROR    UNINITIALIZED  VERSION  NAME     VERSION  DEPENDENCY  ERROR    ERROR  INFO
-----------------------------------------------------------------------------------------------------
cisco-helm  X   -        -              -        -        -        -           -        -      -
web-server  X   -        -              -        -        -        -           -        -      -
```

First we need to create an instance of our web-server service. This will create some data for the banner and the replica count

```
admin@ncs# config
Entering configuration mode terminal
```

```
admin@ncs(config)# web lab-2019
admin@ncs(config-web-lab-2019)# banner "This is awesome!"
admin@ncs(config-web-lab-2019)# replica-count 10
admin@ncs(config-web-lab-2019)# commit
Commit complete.

admin@ncs(config-web-lab-2019)# top
```

This will just create some configuration in NSO, nothing will happen yet.

With Helm, you can override any value in the values file at deployment time. This is typically done by creating a new YAML file with the variables you want to override. When you deploy using Helm, you pass -f <path to override file>. This allows you to make changes to the default values that are shipped with the Helm chart.

The NSO helm-chart package uses the exact same technique. The data in the YANG model (the “web” list above) is rendered into a YAML override file and then passed to Helm.

To deploy this service in the kubernetes cluster, invoke the NSO helm service. This will then create an instance of the web-server chart and deploy to the cluster. In the below example we call our helm chart service “lab”, but this can be anything.

We need to point this instance to the path where the helm client can find the chart (/root/lab-nso/web-chart). In our case it’s a local directory on NSO, but this could point to a Helm repository which is remote from NSO.

We also need to specify from where NSO will render values from, in our case we point to the previously created web-server instance at /web[name=’lab-2019’], this is the data you previously created.

```
admin@ncs(config)# helm chart lab
admin@ncs(config-chart-lab)# chart-path /root/lab-nso/web-chart
admin@ncs(config-chart-lab)# values /web[name='lab-2019']
admin@ncs(config-chart-lab)# commit
```

As soon as we committed this, NSO went to the helm client with the appropriate parameters and initiated an API call to the kubernetes Helm server.

The rendered values file looks like this:

```
web:
  banner: This is awesome!
  name: lab-2019
  replicaCount: 10
```

And the NSO helm service actually launches the real helm client with this command (you do not type this in):

```
helm upgrade --install -f /tmp/tmprW5N32 lab /root/lab-nso/web-chart
```

Where the /tmp/tmprW5N32 is the override values file, please see above. **Note that this file is immediately is cleaned up after the command is run, i.e. you will not be able to look at this file.**

Now go back to the k8s master node (linux1)

```
# helm list
NAME    REVISION        UPDATED                         STATUS          CHART
APP VERSION     NAMESPACE
lab     1               Fri May  3 14:02:22 2019        DEPLOYED        web-0.1.0
1.0             default
```

```
[root@linux1 ~]# kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
labtest-web-84bc9cb9c9-5d552   1/1     Running   0          67s
labtest-web-84bc9cb9c9-c9pfb   1/1     Running   0          72s
labtest-web-84bc9cb9c9-gzpbm   1/1     Running   0          76s
labtest-web-84bc9cb9c9-l7n7v   1/1     Running   0          76s
labtest-web-84bc9cb9c9-m5h84   1/1     Running   0          76s
labtest-web-84bc9cb9c9-ncnxb   1/1     Running   0          68s
labtest-web-84bc9cb9c9-r6sk6   1/1     Running   0          76s
labtest-web-84bc9cb9c9-tnfhh   1/1     Running   0          72s
labtest-web-84bc9cb9c9-vpccb   1/1     Running   0          76s
labtest-web-84bc9cb9c9-wnjf8   1/1     Running   0          70s
```

```

# watch curl -s http://linux1:30080
Banner: This is awesome!
Host name: lab-web-7d54bb8995-xbhgp
IP Addresses: 10.45.0.9
```

So, the Helm chart was successfully deployed in your cluster.

Let’s go back to NSO and make a change, quickly get ready to view the watch command to see the update.

```
# ncs_cli -Cu admin
```

```
admin@ncs# config
Entering configuration mode terminal

admin@ncs(config)# web lab-2019 banner "Even more so!"
admin@ncs(config-web-lab-2019)# commit
Commit complete.
```

This will again trigger Helm, and if you switch back to the k8s master

```
# watch curl http://linux1:30080
Banner: Even more so!
Host name: lab-web-7f75c87847-w8jx5
IP Addresses: 10.44.0.2
```

If you're quick, you may see the old banner message. This is since Helm/k8s does rolling upgrades of the chart.

Finally, let's delete the chart from NSO

```
admin@ncs(config)# no helm chart lab
admin@ncs(config)# commit
Commit complete.
```

And on the k8s master

```
# helm list
```

The helm release is successfully deleted.

The Lab has been successfully completed!

Appendix

Create Docker Repository

First, we need to create a certificate for our Docker repository, make sure the Common Name matches your host where you'll host the Docker repo.

```
# mkdir certs
# openssl req \
  -newkey rsa:4096 -nodes -sha256 -keyout certs/domain.key \
  -x509 -days 365 -out certs/domain.crt
Generating a 4096 bit RSA private key
.....................++
........++
writing new private key to 'certs/domain.key'
-----
```

```
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Cisco
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:172.28.112.105.nip.io
Email Address []:
```

This will generate two files in the certs directory

```
# ls certs
domain.crt domain.key
```

The domain.crt will have to be copied to all Docker hosts (k8s nodes), if not docker will refuse to connect to the registry (unknown certificate).

On each host that will access the docker repository, create a directory matching the common name, in the example above: /etc/docker/certs.d/172.28.112.105.nip.io:5000

And copy the domain.crt file as /etc/docker/certs.d/172.28.112.105.nip.io:5000/ca.crt
Note! It has to be called ca.crt.

To run the docker registry

```
docker run -d \
    --restart=always \
    --name registry \
    -v "$(pwd)"/certs:/certs \
    -e REGISTRY_HTTP_ADDR=0.0.0.0:5000 \
    -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \
    -e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \
    -p 5000:5000 \
    registry:2
```

Please note that this call mounts the certs directory to /certs in the docker container.

To test the registry

```
# docker pull alpine
```

```
Using default tag: latest
latest: Pulling from library/alpine
6c40cc604d8e: Pull complete
Digest: sha256:b3dbf31b77fd99d9c08f780ce6f5282aba076d70a513a8be859d8d3a4d0c92b8
Status: Downloaded newer image for alpine:latest

# docker tag alpine:latest 172.28.112.105.nip.io:5000/alpine:latest

# docker push 172.28.112.105.nip.io:5000/alpine:latest
The push refers to repository [172.28.112.105.nip.io:5000/alpine]
503e53e365f3: Pushed
latest: digest:
sha256:25b4d910f4b76a63a3b45d0f69a57c34157500faf6087236581eca221c62d214 size: 528
```