**Dynamic NFS volume and Helm installation with sample Jenkins deployment**

**Karthikeyan.S**

172-172, 5th floor Old Mahabalipuram Road

(Above Axis Bank-PTC Bus Stop)  
Thuraipakkam  
Chennai 600097

**🖂 karthikeyan1038@gmail.com**

**✆+91 7502047405**

# Contents

* Overviews
* Requirements

I. Kubernetes mater

II. kubernetes node

III. NFS server

* Result

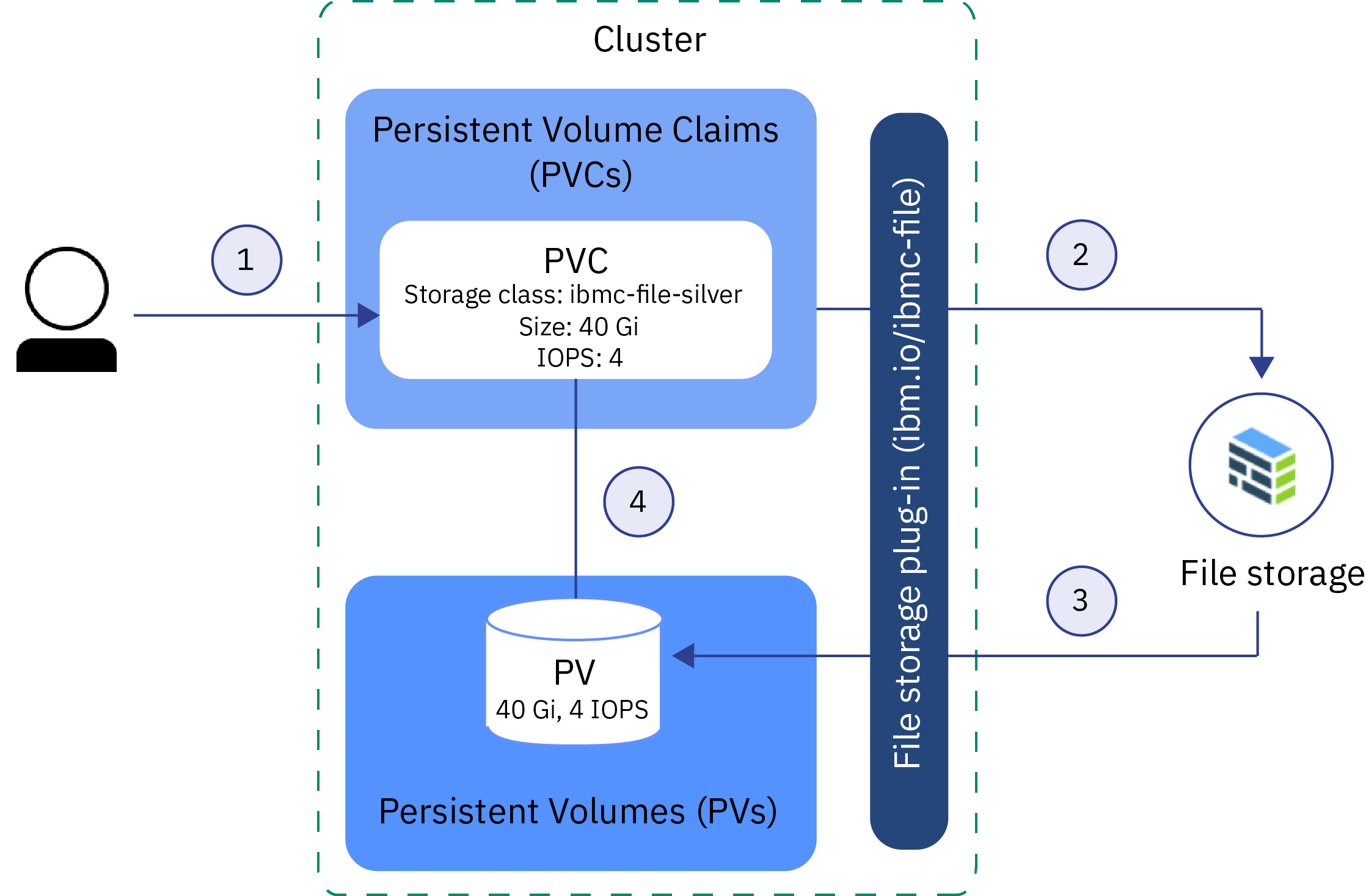
# Overview

## dynamic persistent volume

Dynamic provisioning is a feature that is native to Kubernetes and that allows a cluster developer to order storage with a pre-defined type and configuration without knowing all the details about how to provision the physical storage device. To abstract the details for the specific storage type, the cluster admin must create [storage classes](https://console.bluemix.net/docs/containers/cs_storage_basics.html#storageclasses) that the developer can use, or use the storage classes that are provided with the IBM® Cloud storage plug-ins.

To order the storage, you must create a PVC. The PVC determines the specification for the storage that you want to provision. After the PVC is created, the storage device and the PV are automatically created for you.

The following image shows how file storage is dynamically provisioned in a cluster. This sample flow works similarly with other storage types, such as block storage.



## Nfs server

Network File System (NFS) is a [distributed file system](https://en.wikipedia.org/wiki/Distributed_file_system) protocol originally developed by [Sun Microsystems](https://en.wikipedia.org/wiki/Sun_Microsystems) in 1984,[[1]](https://en.wikipedia.org/wiki/Network_File_System#cite_note-sun85-1) allowing a user on a client [computer](https://en.wikipedia.org/wiki/Computer) to access files over a [computer network](https://en.wikipedia.org/wiki/Computer_network) much like local storage is accessed. NFS, like many other protocols, builds on the [Open Network Computing Remote Procedure Call](https://en.wikipedia.org/wiki/Open_Network_Computing_Remote_Procedure_Call) (ONC RPC) system. The NFS is an open standard defined in [Request for Comments](https://en.wikipedia.org/wiki/Request_for_Comments) (RFC), allowing anyone to implement the protocol.

## HElm

Helm is a tool that streamlines installing and managing Kubernetes applications. Think of it like apt/yum/homebrew for Kubernetes.

* Helm has two parts: a client (helm) and a server (tiller)
* Tiller runs inside of your Kubernetes cluster, and manages releases (installations) of your charts.
* Helm runs on your laptop, CI/CD, or wherever you want it to run.
* Charts are Helm packages that contain at least two things:
* A description of the package (Chart.yaml)
* One or more templates, which contain Kubernetes manifest files
* Charts can be stored on disk, or fetched from remote chart repositories (like Debian or RedHat packages)

# Requirments

Three centos machine. One with 2Cpu, 2GB RAM with 50 GB disk files for kuberenetes master. And other two machine with 1CPU 1GB RAM with 30GB disk space One machine for kubernetes node and other one for Nfs server.

# apply weave network in kubemaster

This command apply the weave network in kubernetes cluster

kubectl apply -f [https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')](https://cloud.weave.works/k8s/net?k8s-version=$(kubectl%20version%20|%20base64%20|%20tr%20-d%20'\n'))

# setup nfs server

One of the most common needs when deploying Kubernetes is the ability to use shared storage. While there are several options available, one of the most commons and easier to setup is to use an NFS server

Install nfs package:

yum install -y nfs-utils

Enable and start nfs service and rpcbind

systemctl enable rpcbind  
systemctl enable nfs-server  
systemctl start rpcbind  
systemctl start nfs-server

Create the directory that will be shared by NFS, and change the permissions

mkdir /var/nfsshare  
chmod -R 755 /var/nfsshare  
chown nfsnobody:nfsnobody /var/nfsshare

Share the NFS directory over the network, creating the /etc/exports file:

vi /etc/exports

------------  
/var/nfsshare 192.168.1.90(rw,sync,no\_root\_squash,no\_all\_squash,no\_subtree\_check,insecure)

/var/nfsshare 192.168.1.91(rw,sync,no\_root\_squash,no\_all\_squash,no\_subtree\_check,insecure)

Where 192.168.1.90 and 192.168.1.91 are kubemaster and kube nodes

Restart the nfs service to apply the content:

systemctl restart nfs-server

Add NFS and rpcbind services to firewall:

firewall-cmd --permanent --zone=public --add-service=nfs  
firewall-cmd --permanent --zone=public --add-service=rpcbind  
firewall-cmd –reload

export the nfs files by giving the following command

$ exportfs –rav

exporting 192.168.1.90:/var/nfsshare

exporting 192.168.1.91:/var/nfsshare

$ exportfs –rav

/var/nfsshare 192.168.1.90(sync,wdelay,hide,no\_subtree\_check,sec=sys,rw,insecure,no\_root\_squash,no\_all\_squash)

/var/nfsshare 192.168.1.91(sync,wdelay,hide,no\_subtree\_check,sec=sys,rw,insecure,no\_root\_squash,no\_all\_squash)

The NFS server is now ready to be used

# Install NFS client provisioner in kubemaster

To achieve that, we will rely on Kubernetes external storage provisioner (<https://github.com/kubernetes-incubator/external-storage>) .

An external provisioner is a dynamic volume provisioner, whose code lives outside kubernetes code.  
It relies on an StorageClass object, that defines the external provisionerinstance. Then, that instance will wait for PersistentVolumeClaims asking for that specific StorageClass, and will automatically create PersistentVolumes.  
In that case we are relying on NFS-client provisioner

(<https://github.com/kubernetes-incubator/external-storage/tree/master/nfs-client>) , that will provide those volumes, relying on an existing NFS server.

In order to use that, several steps are needed:

Clone the external-storage repository and switch to the nfs-client folder:

yum install git –y

git clone https://github.com/kubernetes-incubator/external-storage  
cd external-storage/nfs-client

Customize the deploy/class.yaml file, to give a custom provisioner name to your instance:

$ cat deploy/class.yaml

apiVersion: storage.k8s.io/v1   
kind: StorageClass   
metadata:   
  name: managed-nfs-storage   
provisioner: fuseim.pri/ifs # or choose another name, must match deployment's env PROVISIONER\_NAME'   
parameters:  
  archiveOnDelete: "false"

Customize the deploy/deployment.yaml file, to specify the location and folder for your NFS server, and to give the right provisioner name:

$ cat deploy/deployment.yaml

apiVersion: v1

kind: ServiceAccount   
metadata:  
  name: nfs-client-provisioner   
---   
kind: Deployment   
apiVersion: extensions/v1beta1   
metadata:   
  name: nfs-client-provisioner   
spec:   
  replicas: 1   
  strategy:   
    type: Recreate   
  template:   
   metadata:   
    labels:   
      app: nfs-client-provisioner   
   spec:   
     serviceAccountName: nfs-client-provisioner   
     containers:   
       - name: nfs-client-provisioner   
         image: quay.io/external\_storage/nfs-client-provisioner:latest   
         volumeMounts:   
           - name: nfs-client-root   
             mountPath: /persistentvolumes   
         env:   
           - name: PROVISIONER\_NAME   
             value: fuseim.pri/ifs # or choose another name, must match   
           - name: NFS\_SERVER   
             value: 192.168.1.90  #<<IP\_OF\_YOUR\_NFS\_SERVER>>   
           - name: NFS\_PATH   
             value: /var/nfsshare #<<PATH\_TO\_NFS\_SHARED\_FOLDER>>  
     volumes:   
       - name: nfs-client-root   
         nfs:   
           server: 192.168.1.90 #<<IP\_OF\_YOUR\_NFS\_SERVER>>  
           path: /var/nfsshare #<<PATH\_TO\_NFS\_SHARED\_FOLDER>>

check the deploy/rbac.yaml

$ cat deploy/rbac.yaml

kind: ServiceAccount

apiVersion: v1

metadata:

name: nfs-client-provisioner

---

kind: ClusterRole

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: nfs-client-provisioner-runner

rules:

- apiGroups: [""]

resources: ["persistentvolumes"]

verbs: ["get", "list", "watch", "create", "delete"]

- apiGroups: [""]

resources: ["persistentvolumeclaims"]

verbs: ["get", "list", "watch", "update"]

- apiGroups: ["storage.k8s.io"]

resources: ["storageclasses"]

verbs: ["get", "list", "watch"]

- apiGroups: [""]

resources: ["events"]

verbs: ["create", "update", "patch"]

---

kind: ClusterRoleBinding

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: run-nfs-client-provisioner

subjects:

- kind: ServiceAccount

name: nfs-client-provisioner

namespace: default

roleRef:

kind: ClusterRole

name: nfs-client-provisioner-runner

apiGroup: rbac.authorization.k8s.io

---

kind: Role

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: leader-locking-nfs-client-provisioner

rules:

- apiGroups: [""]

resources: ["endpoints"]

verbs: ["get", "list", "watch", "create", "update", "patch"]

---

kind: RoleBinding

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: leader-locking-nfs-client-provisioner

subjects:

- kind: ServiceAccount

name: nfs-client-provisioner

# replace with namespace where provisioner is deployed

namespace: default

roleRef:

kind: Role

name: leader-locking-nfs-client-provisioner

apiGroup: rbac.authorization.k8s.io

Create the objects into your kubernetes cluster:

kubectl create -f deploy/rbac.yaml  
kubectl create -f deploy/class.yaml  
kubectl create -f deploy/deployment.yaml

To actually start testing the system, you will need to create a PersistentVolumeClaim using that StorageClass. Then you will need to create a pod that uses this PersistentVolumeClaim:

Cat deploy/test-claim.yaml

---------  
kind: PersistentVolumeClaim   
apiVersion: v1   
metadata:  
  name: test-claim  
  annotations:   
    volume.beta.kubernetes.io/storage-class: "managed-nfs-storage"   
spec:  
  accessModes:   
    - ReadWriteMany   
  resources:   
    requests:   
      storage: 1Mi  
  
cat deploy/test-pod.yaml  
--------  
kind: Pod  
apiVersion: v1  
metadata:   
  name: test-pod   
spec:   
  containers:   
   - name: test-pod   
     image: gcr.io/google\_containers/busybox:1.24   
     command:   
       - "/bin/sh"   
     args:   
       - "-c"   
       - "touch /mnt/SUCCESS && exit 0 || exit 1"   
     volumeMounts:   
       - name: nfs-pvc   
         mountPath: "/mnt"   
restartPolicy: "Never"   
volumes:   
   - name: nfs-pvc  
     persistentVolumeClaim:  
       claimName: test-claim

You check the pv and pvc is created, pv is created automatically

[root@kubemaster deploy]# kubectl get pv,pvc

NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE

persistentvolume/pvc-5a5dc0f1-5448-11e9-beb8-08002777b8d4 1Mi RWX Delete Bound default/test-claim managed-nfs-storage 23s

NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE

persistentvolumeclaim/test-claim Bound pvc-5a5dc0f1-5448-11e9-beb8-08002777b8d4 1Mi RWX managed-nfs-storage 35s

the nfs-client provision pods is running

[root@kubemaster deploy]# kubectl get pods

NAME READY STATUS RESTARTS AGE

nfs-client-provisioner-547fdf7d48-wnqhb 1/1 Running 2 44h

# helm installation

You can test your connectivity with the following command:

kubectl cluster-info

If you see no errors, you're connected to the cluster. If you access multiple clusters with kubectl, be sure to verify that you've selected the correct cluster context:

$ kubectl config get-contexts

Output

CURRENT NAME CLUSTER AUTHINFO NAMESPACE

\* do-nyc1-k8s-example do-nyc1-k8s-example do-nyc1-k8s-example-admin

docker-for-desktop docker-for-desktop-cluster docker-for-desktop

In this example the asterisk (\*) indicates that we are connected to the do-nyc1-k8s-examplecluster. To switch clusters run:

kubectl config use-context context-name

When you are connected to the correct cluster, continue to begin installing Helm.

## Step 1 — Installing Helm

First we'll install the helm command-line utility on our local machine. Helm provides a script that handles the installation process on MacOS, Windows, or Linux.

Change to a writable directory and download the script from Helm's GitHub repository:

$ cd /tmp

$ curl https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get > install-helm.sh

Make the script executable with chmod:

$ chmod u+x install-helm.sh

At this point you can use your favorite text editor to open the script and inspect it to make sure it’s safe. When you are satisfied, run it:

$ ./install-helm.sh

Output

helm installed into /usr/local/bin/helm

Run 'helm init' to configure helm

Next we will finish the installation by installing some Helm components on our cluster.

## Step 2 — Installing Tiller

Tiller is a companion to the helm command that runs on your cluster, receiving commands from helm and communicating directly with the Kubernetes API to do the actual work of creating and deleting resources. To give Tiller the permissions it needs to run on the cluster, we are going to make a Kubernetes serviceaccount resource

Note: We will bind this serviceaccount to the cluster-admin cluster role. This will give the tiller service superuser access to the cluster and allow it to install all resource types in all namespaces. This is fine for exploring Helm, but you may want a more locked-down configuration for a production Kubernetes cluster.

Create the tiller serviceaccount:

$ kubectl -n kube-system create serviceaccount tiller

Next, bind the tiller serviceaccount to the cluster-admin role:

$ kubectl create clusterrolebinding tiller --clusterrole cluster-admin --serviceaccount=kube-system:tiller

Now we can run helm init, which installs Tiller on our cluster, along with some local housekeeping tasks such as downloading the stable repo details:

$ helm init --service-account tiller

Output

. . .

Tiller (the Helm server-side component) has been installed into your Kubernetes Cluster.

Please note: by default, Tiller is deployed with an insecure 'allow unauthenticated users' policy.

For more information on securing your installation see: https://docs.helm.sh/using\_helm/#securing-your-helm-installation

Happy Helming!

To verify that Tiller is running, list the pods in thekube-system namespace:

$ kubectl get pods --namespace kube-system

---------

NAME READY STATUS RESTARTS AGE

coredns-fb8b8dccf-hzd57 1/1 Running 2 46h

coredns-fb8b8dccf-lpm2n 1/1 Running 0 46h

etcd-kubemaster.zippyops.com 1/1 Running 2 46h

kube-apiserver-kubemaster.zippyops.com 1/1 Running 2 46h

kube-controller-manager-kubemaster.zippyops.com 1/1 Running 2 46h

kube-proxy-gvzll 1/1 Running 2 46h

kube-proxy-zqv92 1/1 Running 2 46h

kube-scheduler-kubemaster.zippyops.com 1/1 Running 2 46h

**tiller-deploy-8458f6c667-9dk96 1/1 Running 2 45h**

weave-net-khccz 2/2 Running 5 46h

weave-net-zvfbk 2/2 Running 6 46h

The Tiller pod name begins with the prefix tiller-deploy-.

Now that we've installed both Helm components, we're ready to use helm to install our first application.

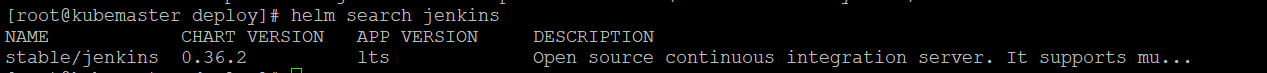
## basic helm commands

* [helm completion](https://helm.sh/docs/helm/#helm-completion) - Generate autocompletions script for the specified shell (bash or zsh)
* [helm create](https://helm.sh/docs/helm/#helm-create) - create a new chart with the given name
* [helm delete](https://helm.sh/docs/helm/#helm-delete) - given a release name, delete the release from Kubernetes
* [helm dependency](https://helm.sh/docs/helm/#helm-dependency) - manage a chart’s dependencies
* [helm fetch](https://helm.sh/docs/helm/#helm-fetch) - download a chart from a repository and (optionally) unpack it in local directory
* [helm get](https://helm.sh/docs/helm/#helm-get) - download a named release
* [helm history](https://helm.sh/docs/helm/#helm-history) - fetch release history
* [helm home](https://helm.sh/docs/helm/#helm-home) - displays the location of HELM-HOME
* [helm init](https://helm.sh/docs/helm/#helm-init) - initialize Helm on both client and server
* [helm inspect](https://helm.sh/docs/helm/#helm-inspect) - inspect a chart
* [helm install](https://helm.sh/docs/helm/#helm-install) - install a chart archive
* [helm lint](https://helm.sh/docs/helm/#helm-lint) - examines a chart for possible issues
* [helm list](https://helm.sh/docs/helm/#helm-list) - list releases
* [helm package](https://helm.sh/docs/helm/#helm-package) - package a chart directory into a chart archive
* [helm plugin](https://helm.sh/docs/helm/#helm-plugin) - add, list, or remove Helm plugins
* [helm repo](https://helm.sh/docs/helm/#helm-repo) - add, list, remove, update, and index chart repositories
* [helm reset](https://helm.sh/docs/helm/#helm-reset) - uninstalls Tiller from a cluster
* [helm rollback](https://helm.sh/docs/helm/#helm-rollback) - roll back a release to a previous revision
* [helm search](https://helm.sh/docs/helm/#helm-search) - search for a keyword in charts
* [helm serve](https://helm.sh/docs/helm/#helm-serve) - start a local http web server
* [helm status](https://helm.sh/docs/helm/#helm-status) - displays the status of the named release
* [helm template](https://helm.sh/docs/helm/#helm-template) - locally render templates
* [helm test](https://helm.sh/docs/helm/#helm-test) - test a release
* [helm upgrade](https://helm.sh/docs/helm/#helm-upgrade) - upgrade a release
* [helm verify](https://helm.sh/docs/helm/#helm-verify) - verify that a chart at the given path has been signed and is valid
* [helm version](https://helm.sh/docs/helm/#helm-version) - print the client/server version information

# deploy jenkins using helm and dynamic persistent volume NFS

Search for the Jenkins application using helm search

helm search jenkins



Create a values files for Jenkins

[root@kubemaster deploy]# cat /tmp/values

----------

Master:

AdminPassword: admin

resources:

limits:

cpu: "500m"

memory: "1024Mi"

ServiceType: NodePort

NodePort: 30100

Persistence:

StorageClass: "managed-nfs-storage"

Size: 5Gi

rbac:

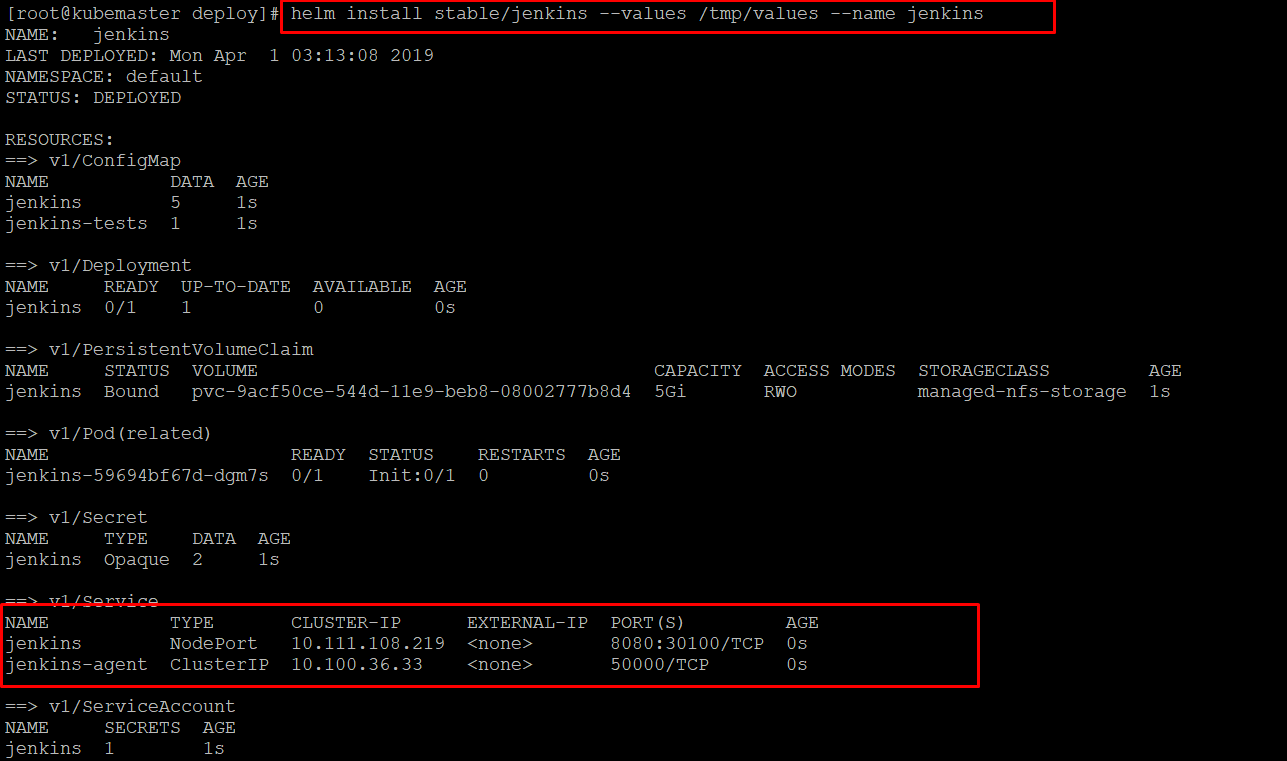
install: true

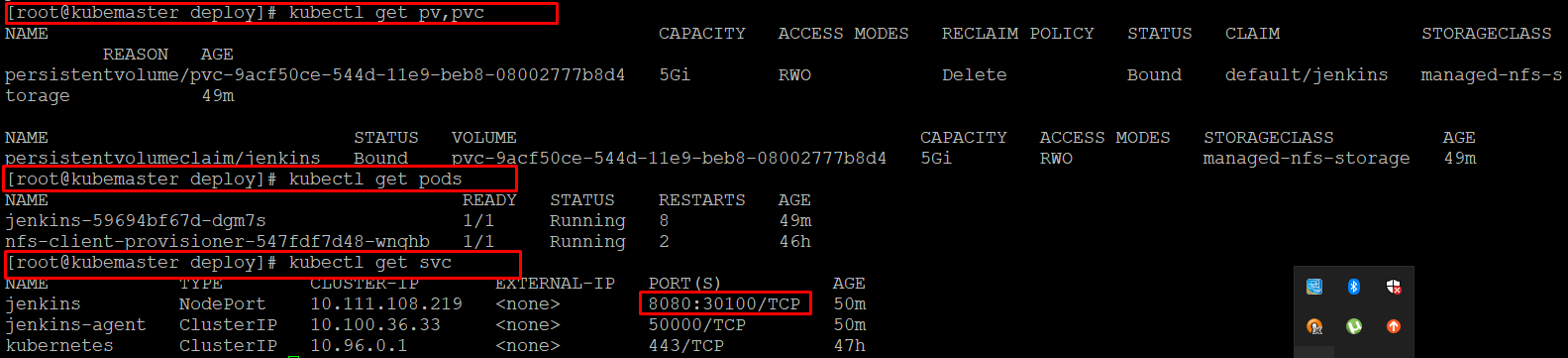
this value file is created to override the default value files

Now give the helm install command

$ helm install stable/jenkins --values /tmp/values --name Jenkins

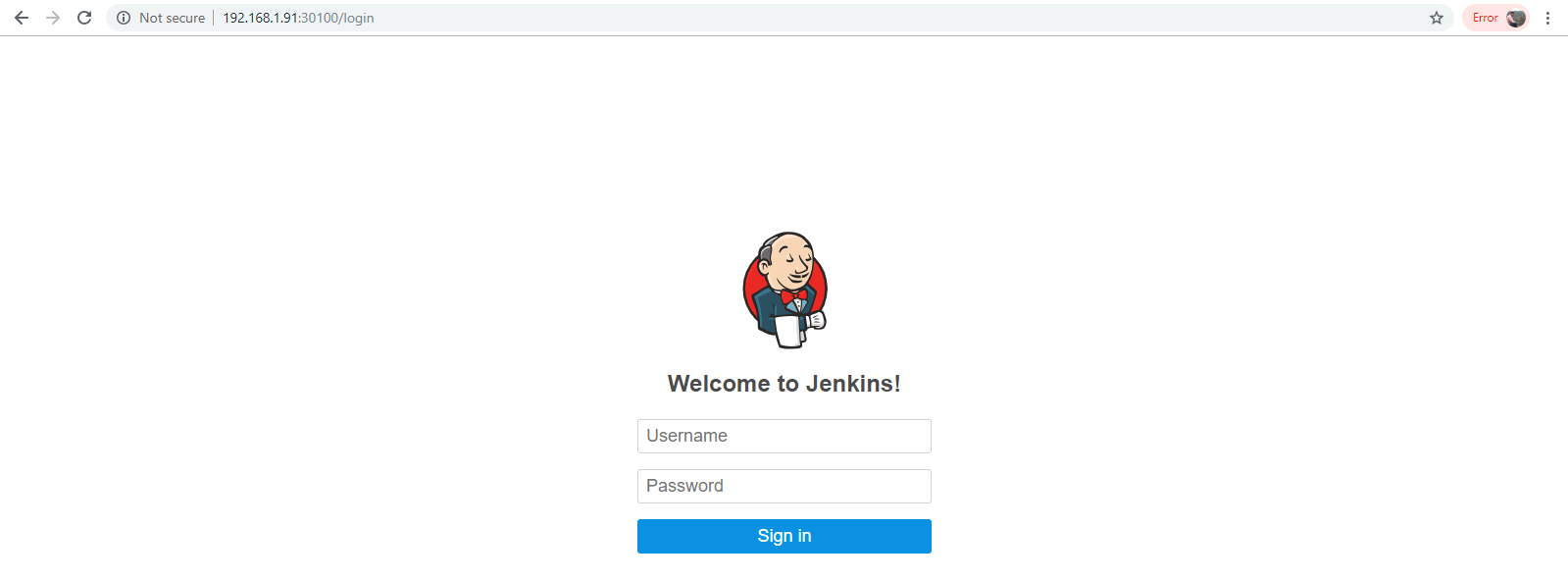
The install command should be stable/Jenkins where stable is the repo name of helm



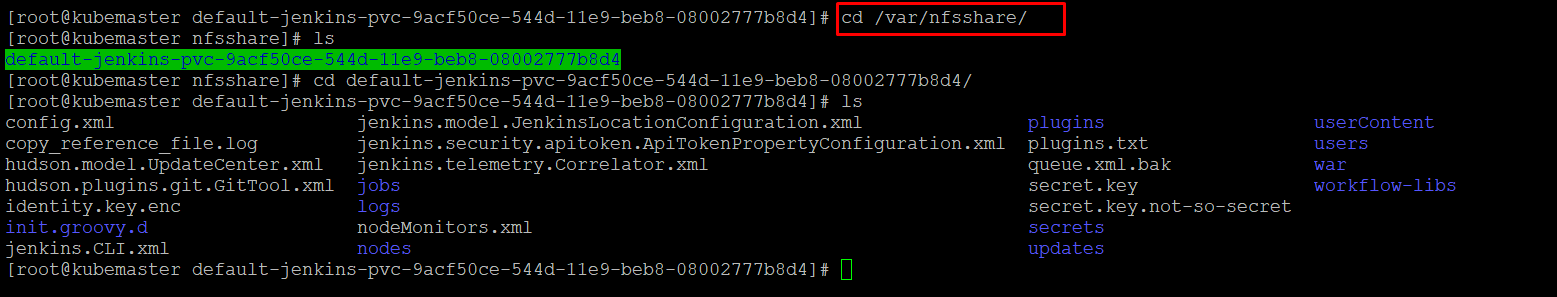


Now the Jenkins is ready

# result



In nfs server go to /var/nfsshare



# basic structure of helm application

Create a helm new chart by giving helm create <chart name>

$ Helm create app1

A new app1 chart is created



If you take a look at the app1/templates/ directory, you’ll notice a few files already there.

* NOTES.txt: The “help text” for your chart. This will be displayed to your users when they run helm install.
* deployment.yaml: A basic manifest for creating a Kubernetes [deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)
* service.yaml: A basic manifest for creating a [service endpoint](https://kubernetes.io/docs/concepts/services-networking/service/) for your deployment
* \_helpers.tpl: A place to put template helpers that you can re-use throughout the chart

The above procedure is the basic steps to create the chart of your own