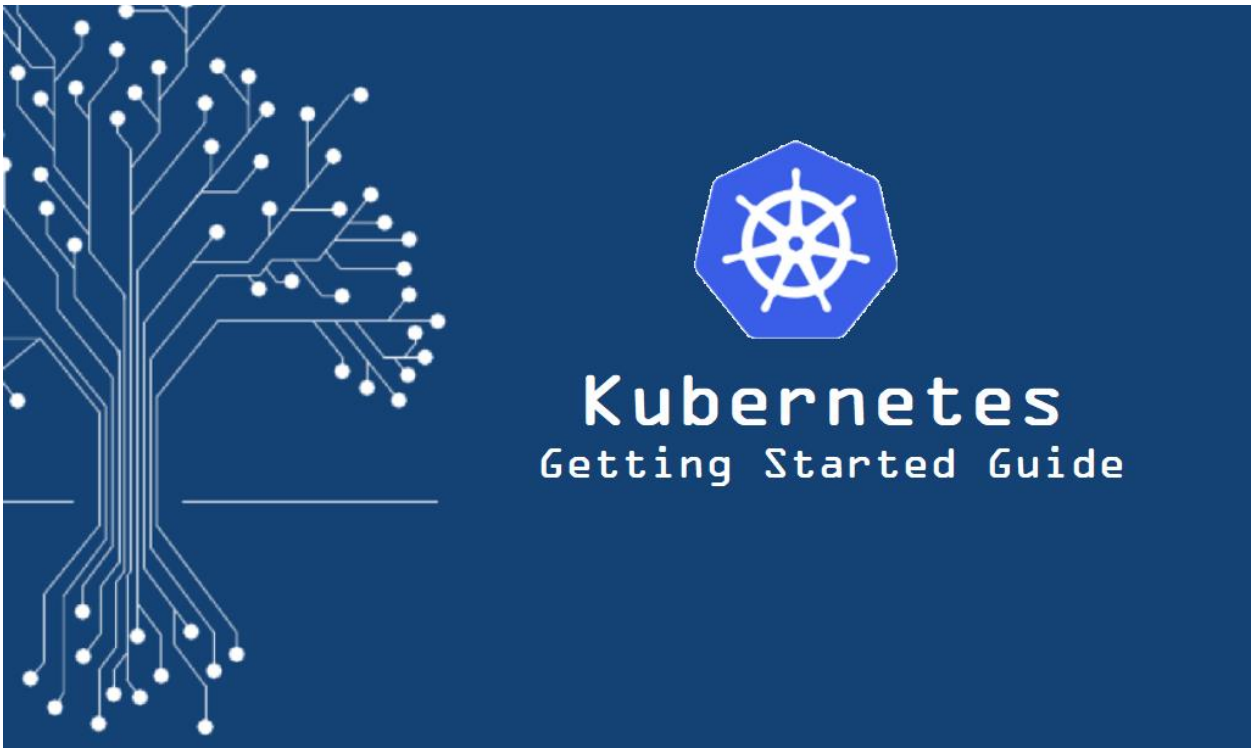



# Getting Started with Kubernetes

*By* ERIC SHANKS

## Contents

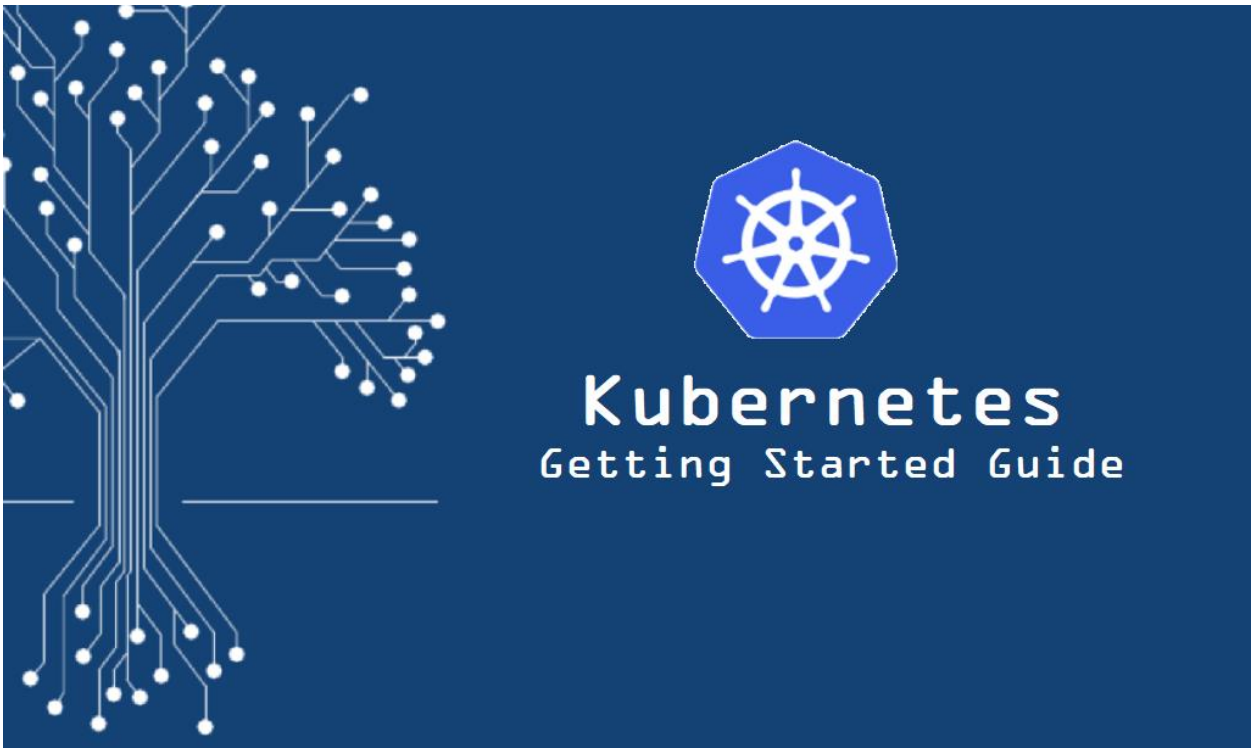
Getting Started with Kubernetes ..... 1
Setup a Kubernetes Cluster – Pick 1 Option ..... 6
Using Kubernetes ..... 7
1. Pods ..... 7
2. Replica Sets ..... 7
3. Deployments ..... 7
4. Services and Labels ..... 8
5. Endpoints ..... 8
6. Service Publishing ..... 9
7. Namespaces ..... 10
8. Context ..... 11
10. DNS ..... 13
11. ConfigMaps ..... 14

12. Secrets ..... 15
13. Persistent Volumes ..... 16
14. Cloud Providers and Storage Classes ..... 17
15. Stateful Sets ..... 18
16. Role Based Access ..... 18
17. Pod Backups ..... 19
18. Helm Charts ..... 19
19. Taints and Tolerations ..... 20
20. DaemonSets ..... 20
21. Network Policies ..... 21
22. Pod Security Policies ..... 21
23. Resource Requests and Limits ..... 22
24. Pod Autoscaling ..... 22
25. Liveness and Readiness Probes ..... 23
26. Validating Admission Controllers ..... 23
27. Jobs and CronJobs ..... 23
Kubernetes – Pods ..... 24
Pods – The Theory ..... 24
Single Container Pods ..... 24
Multi-container Pods ..... 24
Pods – In Action ..... 25
Kubernetes – Replica Sets ..... 27
Replica Sets – The Theory ..... 27
Replica Sets – In Action ..... 28
Kubernetes – Deployments ..... 30
Deployments – The Theory ..... 30
Deployments – In Action ..... 31
Kubernetes – Services and Labels ..... 35
Services – The Theory ..... 35
Services and Labels – In Action ..... 36
Kubernetes – Services and Labels ..... 38
Services – The Theory ..... 38
Services and Labels – In Action ..... 40

Kubernetes – Endpoints ..... 42
Endpoints – The Theory ..... 42
Endpoints – In Action ..... 42
Kubernetes – Service Publishing ..... 46
ClusterIP – The Theory ..... 46
NodePort – The Theory ..... 47
LoadBalancer – The Theory ..... 47
ClusterIP – In Action ..... 47
NodePort – In Action ..... 50
Kubernetes – Namespaces ..... 53
Namespaces – The Theory ..... 53
Namespaces – In Action ..... 54
Kubernetes – KUBECONFIG and Context ..... 56
KUBECONFIG and Context – The Theory ..... 57
KUBECONFIG and Context – In Action ..... 58
Kubernetes – Ingress ..... 61
Ingress Controllers – The Theory ..... 61
Ingress Controllers – In Action ..... 62
Kubernetes – DNS ..... 72
Kubernetes DNS – The theory ..... 73
Kubernetes DNS – In Action ..... 74
Kubernetes – ConfigMaps ..... 76
ConfigMaps – The Theory ..... 76
ConfigMaps – In Action ..... 77
Kubernetes – Secrets ..... 82
Secrets – The Theory ..... 82
Secrets – In Action ..... 82
Kubernetes – Persistent Volumes ..... 87
Volumes – The Theory ..... 87
Volumes – In Action ..... 88
Kubernetes – Cloud Providers and Storage Classes ..... 98
Cloud Providers – The Theory ..... 99
Storage Classes – The Theory ..... 99

Cloud Provider – In Action ..... 100
Storage Classes – In Action ..... 107
Kubernetes – StatefulSets ..... 108
StatefulSets – The Theory ..... 109
StatefulSets – In Action ..... 109
Kubernetes – Role Based Access ..... 117
Role Based Access – The Theory ..... 118
Role Based Access – In Action ..... 119
Kubernetes – Pod Backups ..... 124
Pod Backups – The Theory ..... 125
Pod Backups – In Action ..... 125
Kubernetes – Helm ..... 129
Helm – The Theory ..... 129
Helm – In Action ..... 130
Kubernetes – Taints and Tolerations ..... 135
Taints – The Theory ..... 135
Tolerations – The Theory ..... 136
Taints – In Action ..... 136
Tolerations – In Action ..... 138
Kubernetes – DaemonSets ..... 139
DaemonSets – The Theory ..... 139
DaemonSets – In Practice ..... 140
Kubernetes – Network Policies ..... 141
Network Policies – The Theory ..... 142
Network Policies – In Action ..... 142
Kubernetes – Pod Security Policies ..... 145
Pod Security Policies – The Theory ..... 145
Pod Security Policies – In Action ..... 145
Kubernetes Resource Requests and Limits ..... 151
Resource Requests and Limits – The Theory ..... 151
Resource Requests and Limits – In Action ..... 152
Kubernetes Pod Auto-scaling ..... 155
Horizontal Pod Auto-scaling – The Theory ..... 155

Horizontal Pod Auto-scaling – In Action ..... 156
Kubernetes Liveness and Readiness Probes ..... 158
Liveness and Readiness Probes – The Theory ..... 158
Liveness and Readiness Probes – In Action ..... 160
Kubernetes Validating Admission Controllers ..... 163
Validating Admission Controllers – The Theory ..... 163
Validating Admission Controllers – In Action ..... 167
Kubernetes – Jobs and CronJobs ..... 172
Jobs and CronJobs – The Theory ..... 172
Jobs and CronJobs – In Action ..... 173
Deploy Kubernetes Using Kubeadm – CentOS7 ..... 175
Prerequisites ..... 176
Setup Kubernetes Cluster on Master Node ..... 177
Configure Member Nodes ..... 178
Configure a Mac Client ..... 180
Deploy Kubernetes on AWS ..... 180
AWS Prerequisites ..... 180
Create a Kubeadm Config File ..... 187
Kubernetes EC2 Instance Setup ..... 188
Bootstrap the First K8s Control Plane Node ..... 189
Add Additional Control Plane Nodes ..... 189
Join Worker Nodes to the Cluster ..... 190
Setup KUBECONFIG ..... 190
Deploy a CNI ..... 191
Deploy an AWS Storage Class ..... 191
Common Errors ..... 191
Missing EC2 Cluster Tags ..... 192
Missing Subnet Cluster Tags ..... 192
Node Names Don’t Match Private DNS Names ..... 192




The following posts are meant to get a beginner started with the process of understanding Kubernetes. They include basic level information to start understanding the concepts of the Kubernetes service and include both theory and examples.

To follow along with the series, a Kubernetes cluster should be deployed, and admin permissions are needed to perform many of the steps. If you wish to follow along with each of the posts, a cluster with cloud provider integration may be needed. In some cases, we need a Load Balancer and elastic storage options.

If you would like to follow along, there is a github project corresponding to this guide to save from copying and pasting code snippets.

---

# Setup a Kubernetes Cluster – Pick 1 Option

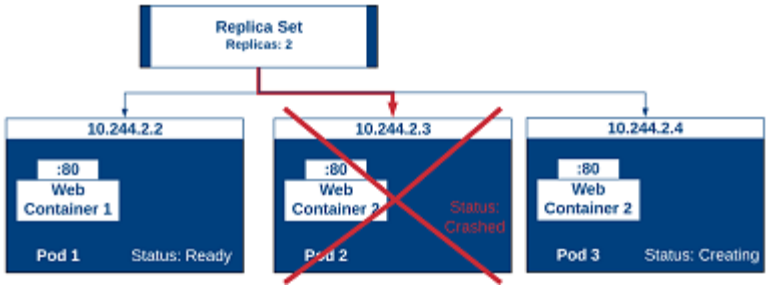
- Deploy Kubernetes on vSphere
- Deploy Kubernetes on AWS

---

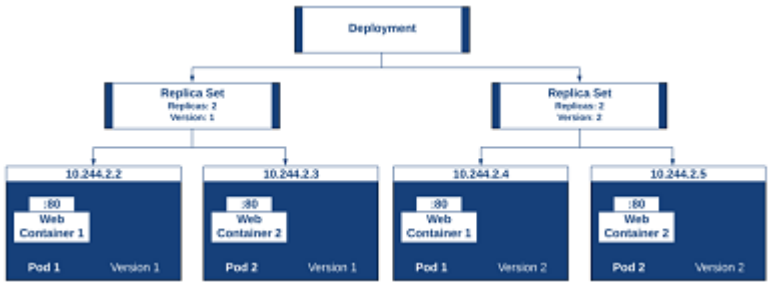
# Using Kubernetes

## 1. Pods

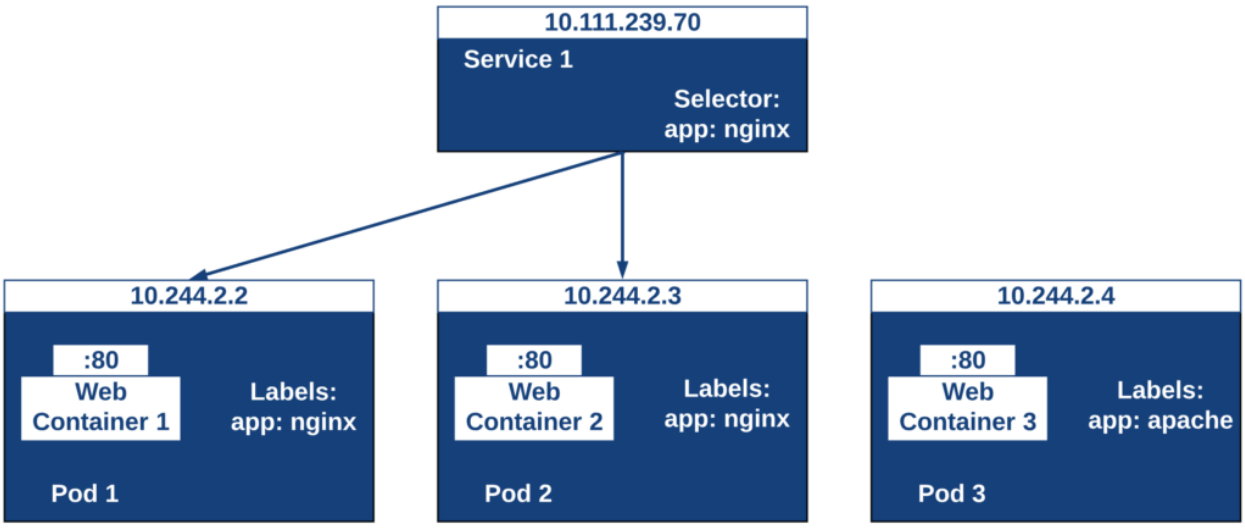

## 2. Replica Sets




## 3. Deployments



# 4. Services and Labels

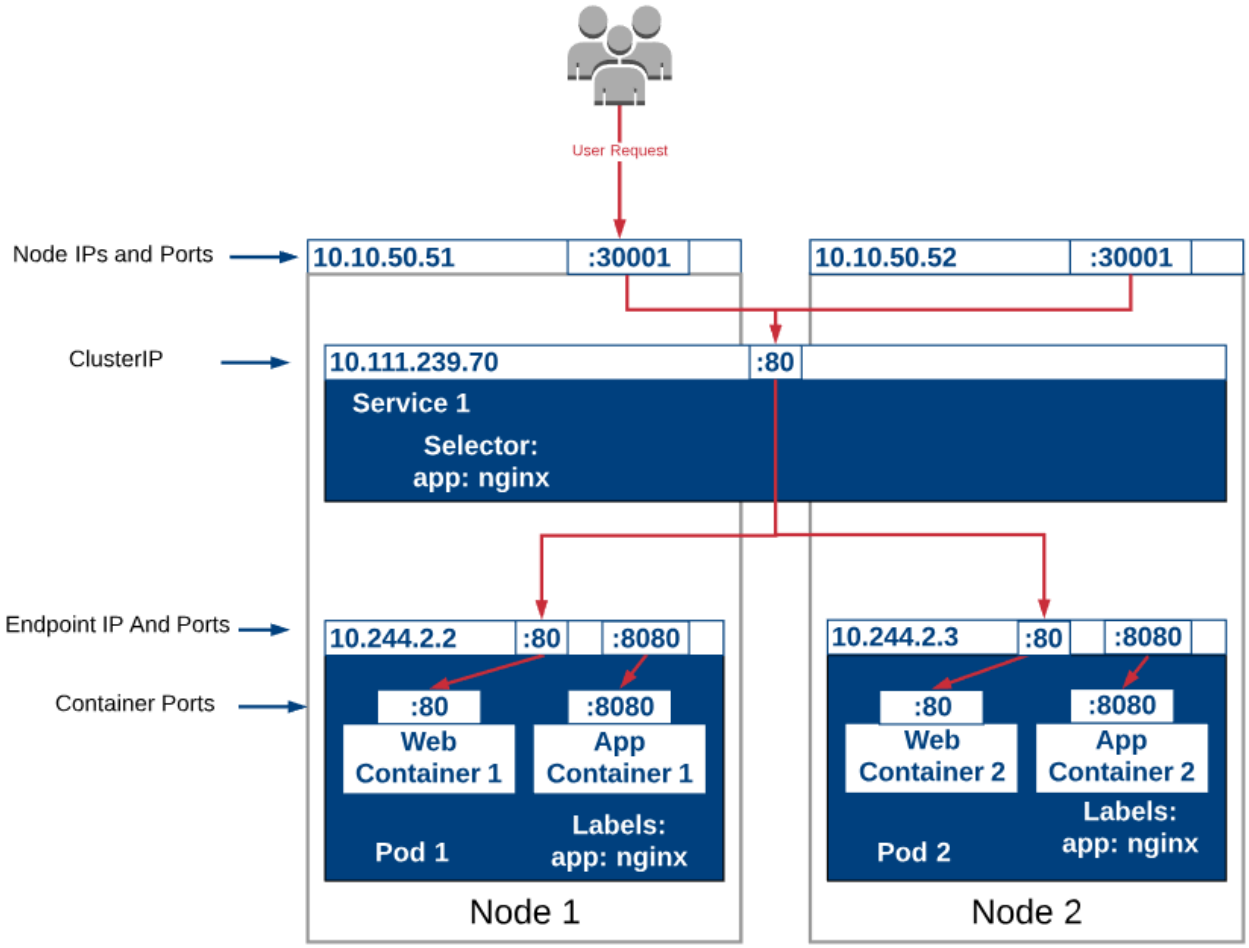



# 5. Endpoints

---

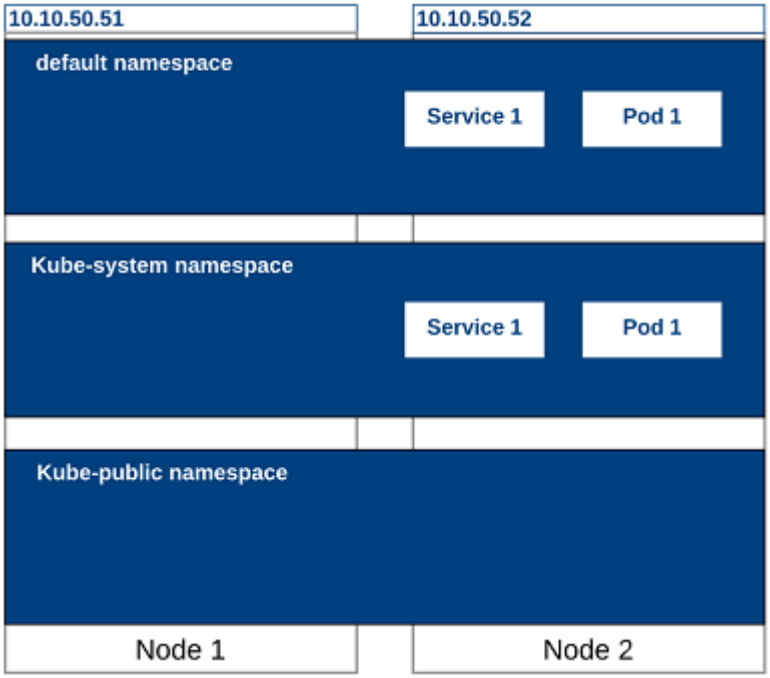
```
Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl get endpoints
NAME            ENDPOINTS                           AGE
ingress-nginx   10.244.1.143:80,10.244.2.204:80     23h
kubernetes      10.10.50.50:6443                    11d
```

# 6. Service Publishing

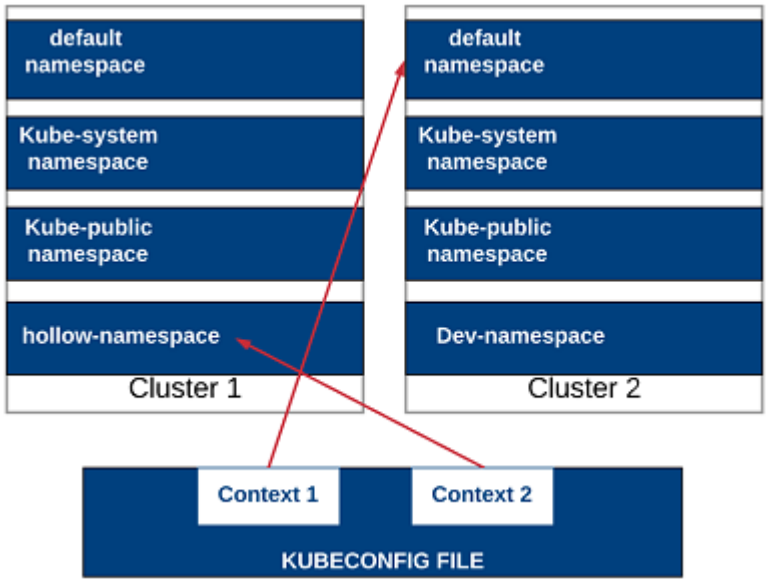


# 7. Namespaces

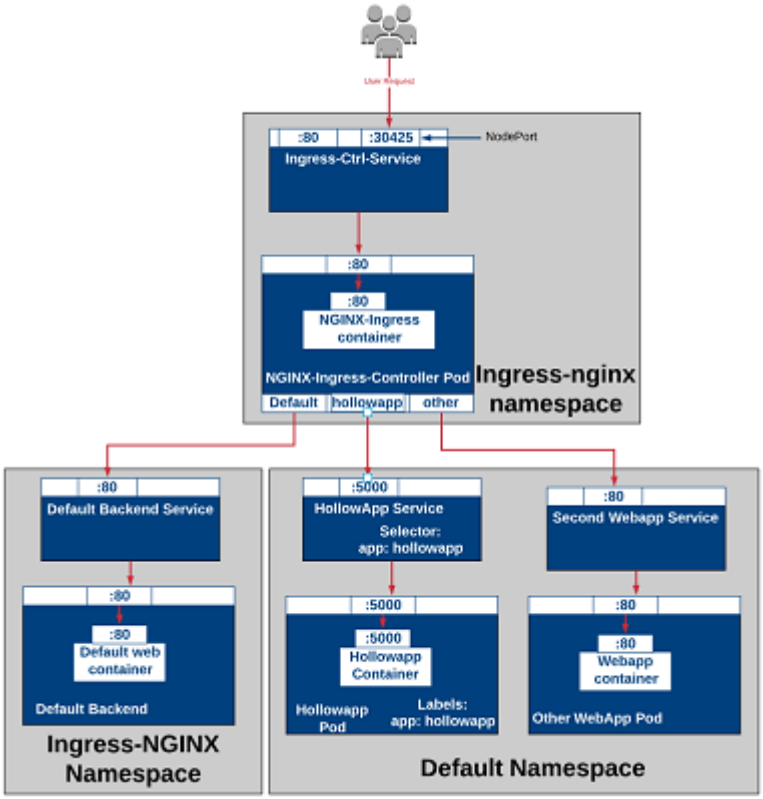
---



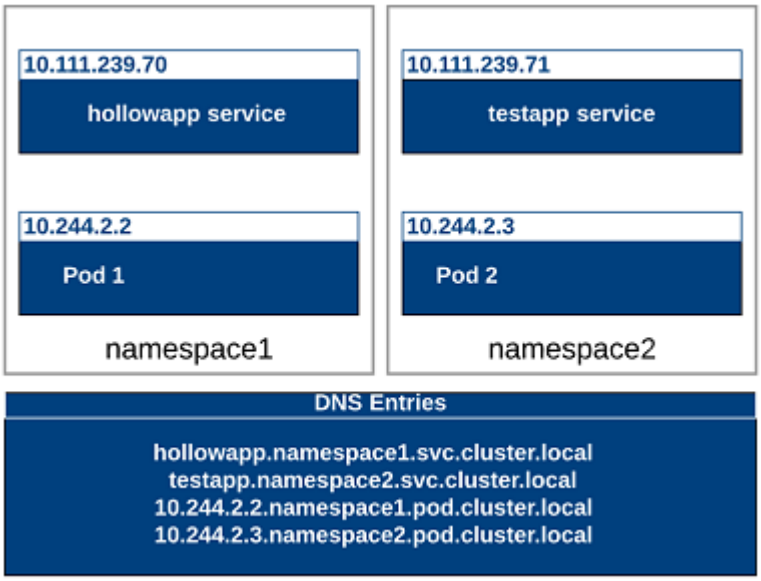
# 8. Context


default namespace
Kube-system namespace
Kube-public namespace
hollow-namespace
Cluster 1
default namespace
Kube-system namespace
Kube-public namespace
Dev-namespace
Cluster 2
Context 1
Context 2
KUBECONFIG FILE

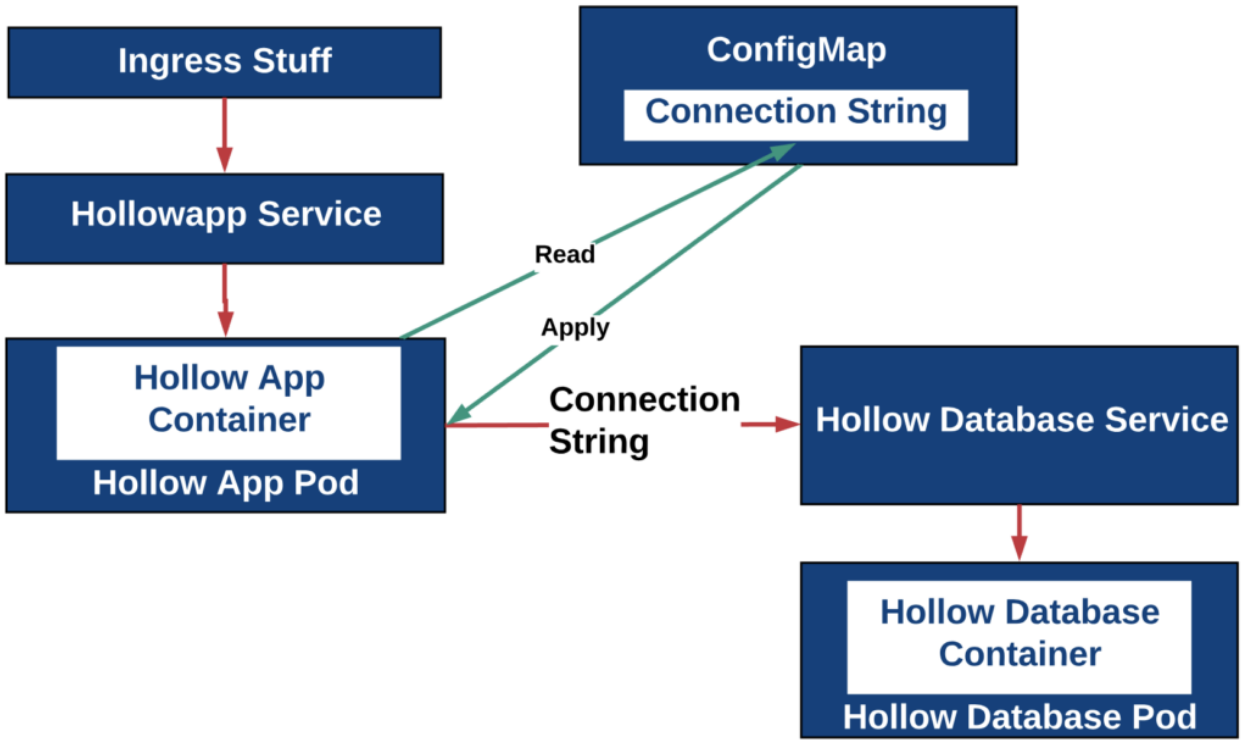
# 9. Ingress


:80
:30425
NodePort
Ingress-Ctrl-Service
:80
:80
NGINX-Ingress container
NGINX-Ingress-Controller Pod
Default
hollowapp
other
Ingress-nginx namespace
:80
Default Backend Service
:80
:80
Default web container
Default Backend
Ingress-NGINX Namespace
:5000
HollowApp Service
Selector: app: hollowapp
:5000
:5000
Hollowapp Container
Hollowapp Pod
Labels: app: hollowapp
:80
Second Webapp Service
:80
:80
Webapp container
Other WebApp Pod
Default Namespace

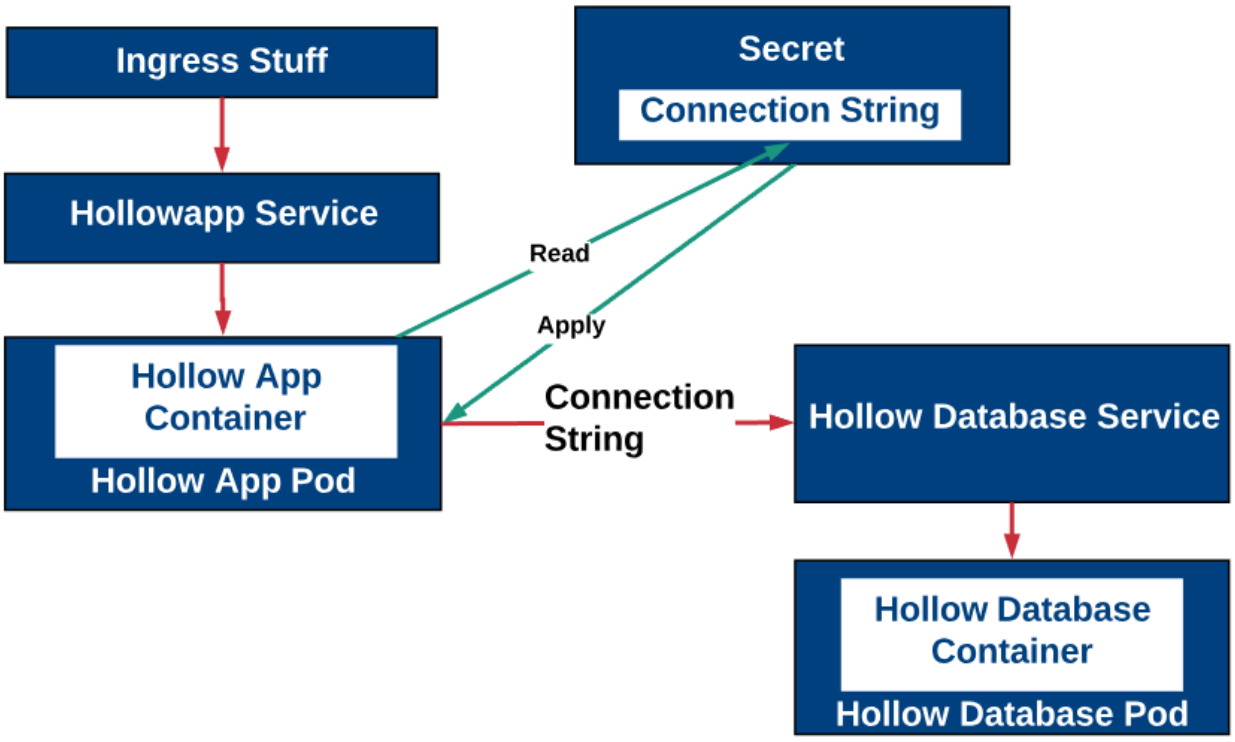
# 10. DNS


10.111.239.70
hollowapp service
10.111.239.71
testapp service
10.244.2.2
Pod 1
10.244.2.3
Pod 2
namespace1
namespace2
DNS Entries
hollowapp.namespace1.svc.cluster.local
testapp.namespace2.svc.cluster.local
10.244.2.2.namespace1.pod.cluster.local
10.244.2.3.namespace2.pod.cluster.local

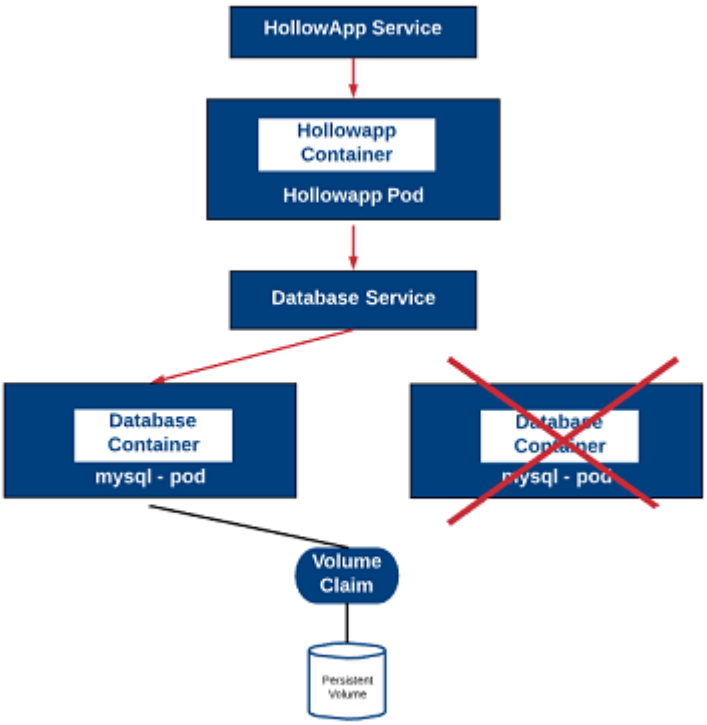
# 11. ConfigMaps



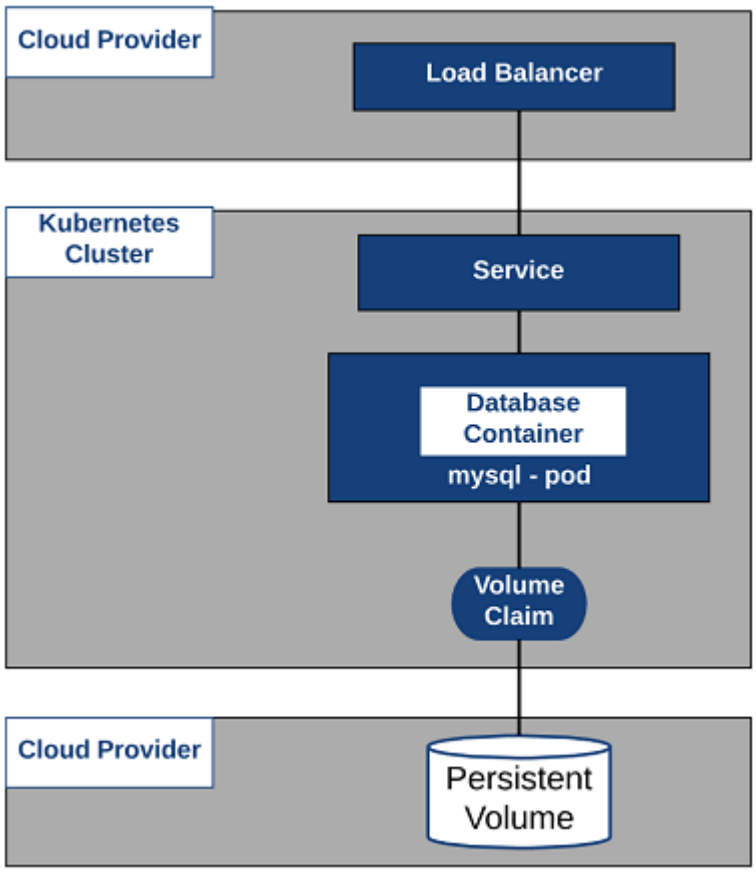
# 12. Secrets


Ingress Stuff
Hollowapp Service
Hollow App Container
Hollow App Pod
Secret
Connection String
Read
Apply
Connection String
Hollow Database Service
Hollow Database Container
Hollow Database Pod

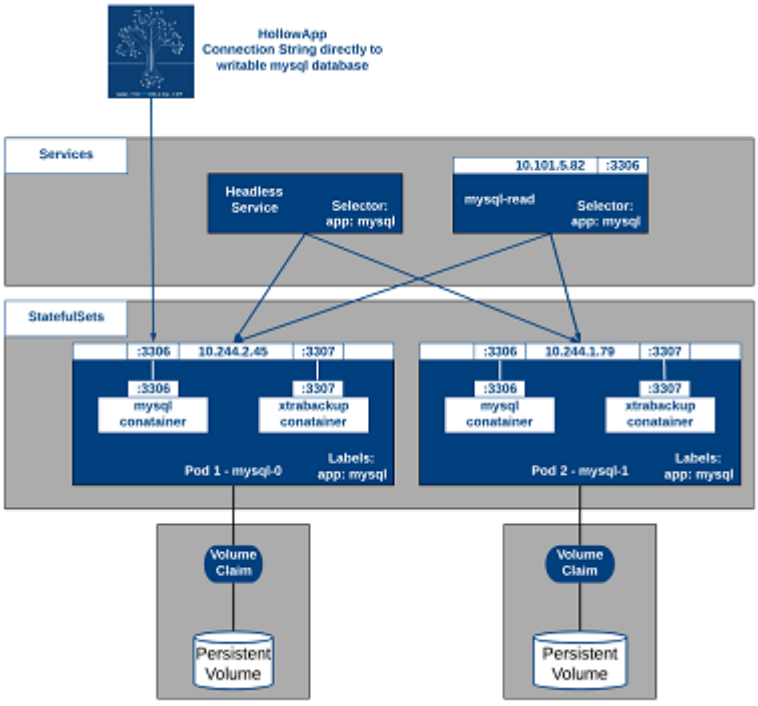
# 13. Persistent Volumes



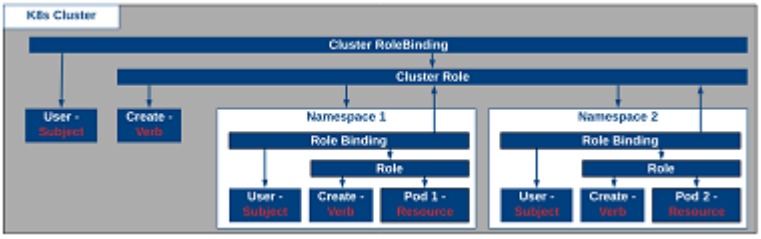
# 14. Cloud Providers and Storage Classes



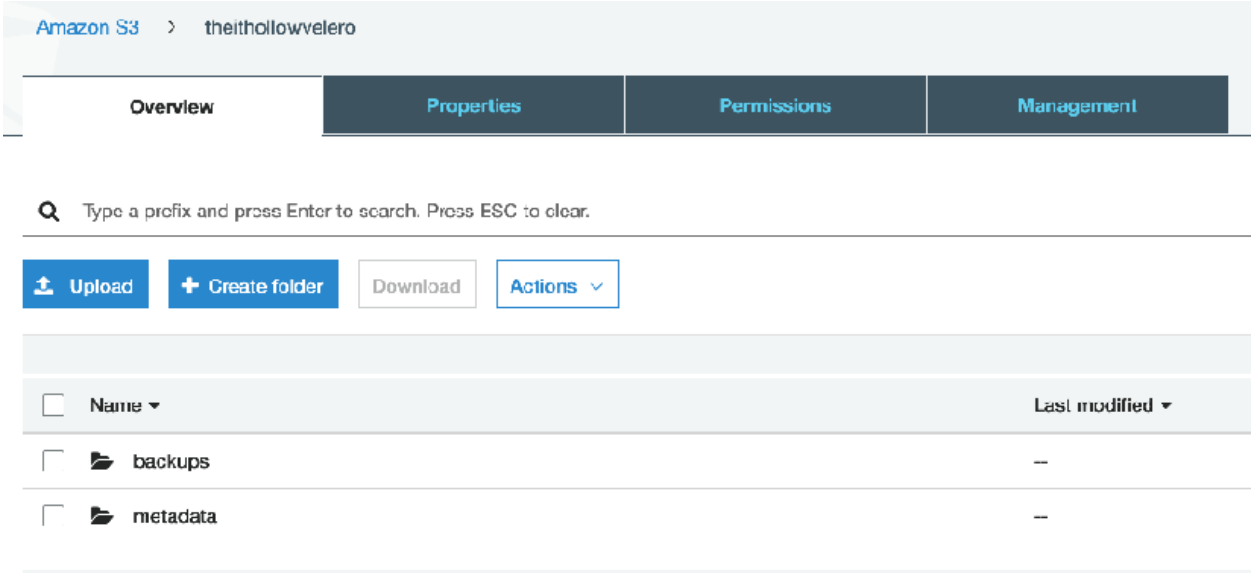
# 15. Stateful Sets



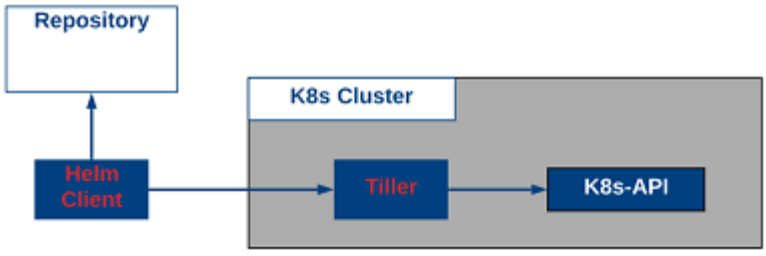

# 16. Role Based Access



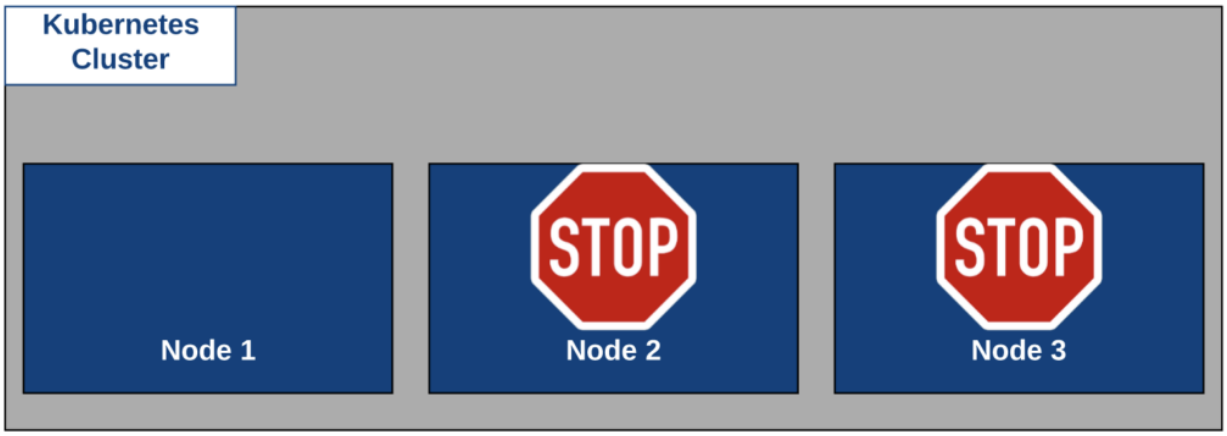
# 17. Pod Backups




# 18. Helm Charts



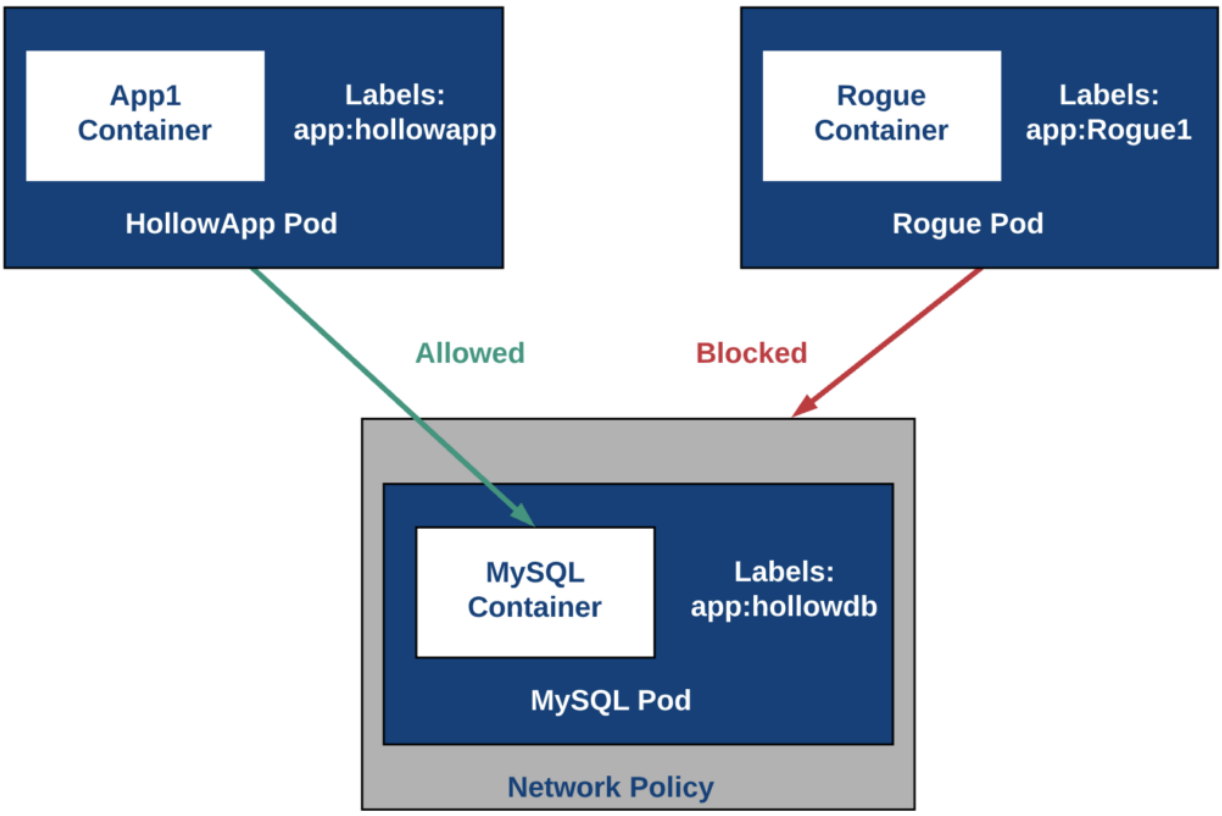
# 19. Taints and Tolerations




# 20. DaemonSets

| NAME | READY | STATUS | RESTARTS | AGE | IP | NODE |
|---|---|---|---|---|---|---|
| daemonset-example-bcnw4 | 1/1 | Running | 0 | 29m | 172.16.3.11 | k8s-worker-1 |
| daemonset-example-jqfxx | 1/1 | Running | 0 | 29m | 172.16.2.11 | k8s-master-2 |
| daemonset-example-k6cs9 | 1/1 | Running | 0 | 29m | 172.16.0.10 | k8s-master-0 |
| daemonset-example-k6pv7 | 1/1 | Running | 0 | 29m | 172.16.1.11 | k8s-master-1 |
| daemonset-example-mf8vj | 1/1 | Running | 0 | 45s | 172.16.6.2 | k8s-worker-3 |
| daemonset-example-p868c | 1/1 | Running | 0 | 29m | 172.16.4.13 | k8s-worker-0 |
| daemonset-example-xfpxn | 1/1 | Running | 0 | 29m | 172.16.5.13 | k8s-worker-2 |

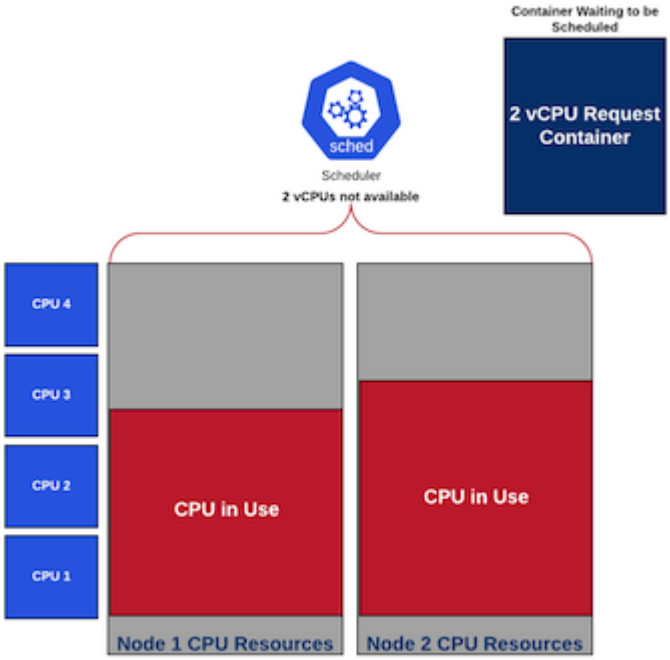
# 21. Network Policies



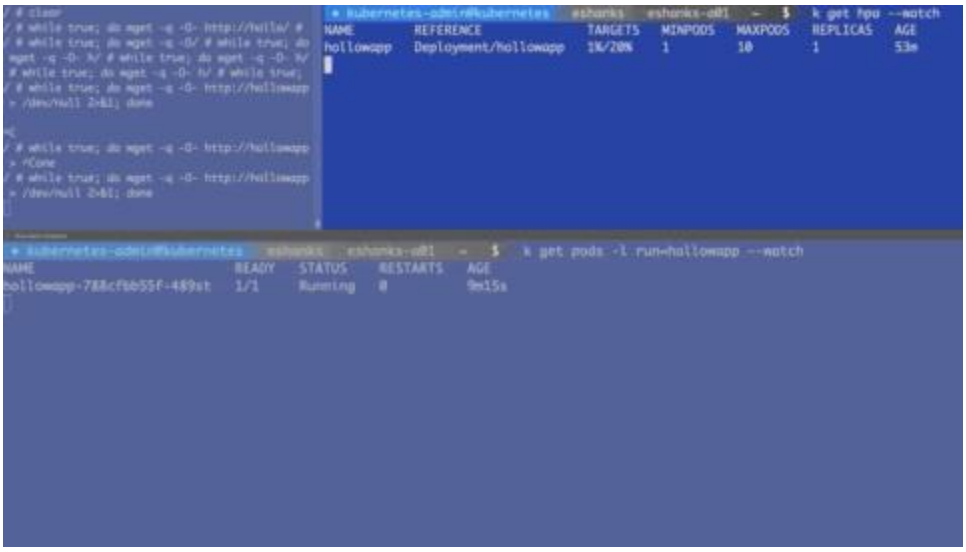

# 22. Pod Security Policies

```
eshanks-a01:securitypolicy eshanks$ kubectl create -f escalated.yaml
deployment.apps/escalated created
eshanks-a01:securitypolicy eshanks$ kubectl get replicasets
NAME                       DESIRED   CURRENT   READY   AGE
escalated-5c8fbfd466       1         0         0       13s
not-escalated-9b8c8454f    1         1         1       5m17s
```

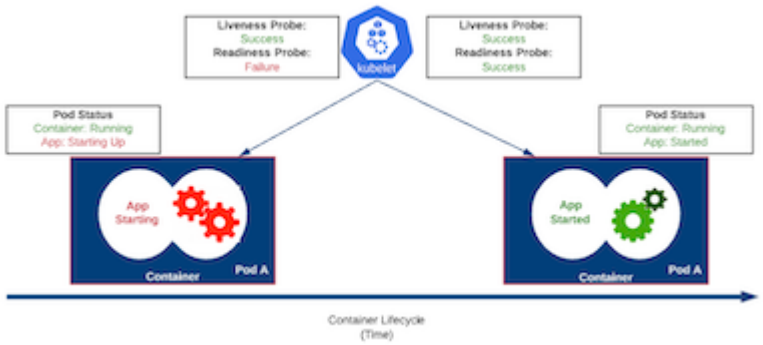
# 23. Resource Requests and Limits



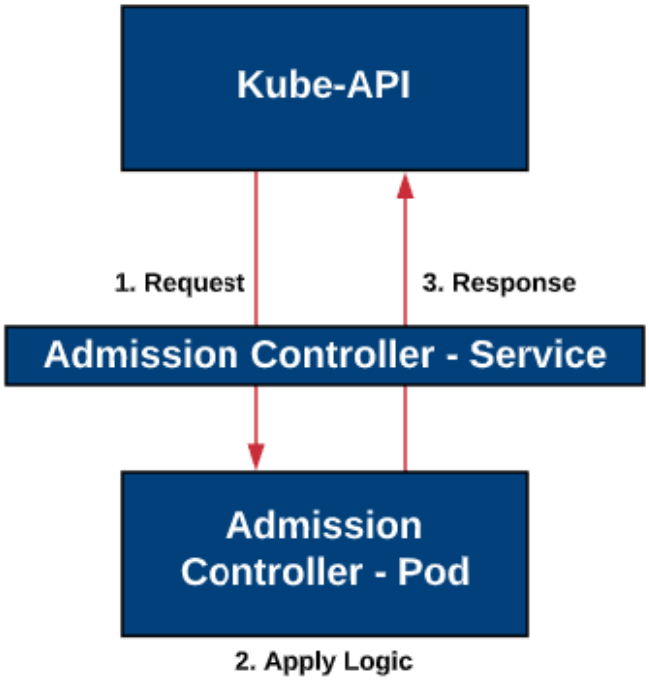

# 24. Pod Autoscaling



## 25. Liveness and Readiness Probes




## 26. Validating Admission Controllers




## 27. Jobs and CronJobs

```
eshanks-a01:k8s-guide eshanks$ k get cronjob --watch
NAME          SCHEDULE    SUSPEND   ACTIVE   LAST SCHEDULE   AGE
hollow-curl   0 * * * *   False     0        <none>          50m
hollow-curl   0 * * * *   False     1        9s              50m
hollow-curl   0 * * * *   False     0        19s             50m
```

# Kubernetes – Pods

*By* ERIC SHANKS

We've got a Kubernetes cluster setup and we're ready to start deploying some applications. Before we can deploy any of our containers in a kubernetes environment, we'll need to understand a little bit about pods.

## Pods – The Theory

In a docker environment, the smallest unit you'd deal with is a container. In the Kubernetes world, you'll work with a pod and a pod consists of one or more containers. You cannot deploy a bare container in Kubernetes without it being deployed within a pod.

The pod provides several things to the containers running within it. These include options on how the container should run, any storage resources and a network namespace. The pod encapsulates these for the containers that run inside them.

## Single Container Pods

The simplest way to get your containers deployed in Kubernetes is the one container per pod approach. When you deploy your applications and you've got a web container and an app container, each of them could be in their own pods. When you need to scale out either the web or app containers, you would then deploy additional pods.


## Multi-container Pods

Sometimes you'll want to deploy more than one container per pod. In general, this is done when the containers are very tightly coupled. One good reason to do this is if the containers are sharing the same storage resource between the two containers. If they are, this is a good time to use a multi-container pod. Another common reason to use multi-container pods is for things like a service mesh where a sidecar container is deployed alongside your application containers to link them together.

It is important to note however, that all containers that run within a pod will share an network namespace so this means that the containers are sharing an IP address and thus will need to be on different ports for them to be accessed. Containers within the same pod may access each other by using the localhost address.


It's also worth noting that a pod will have all containers up or none of them. Your pods will not be considered healthy until all containers that run within them are ready to go, so you can't have one healthy container and one unhealthy container in the same pod. The pod deployment will fail if this happens.

Those are the pod basics. Next up, let's look at how to deploy a container within a pod.

# Pods – In Action

Before we dive too far into this, you should know that this section shows some basics about deploying your first pods in a Kubernetes cluster, but these are considered "naked pods" and should be avoided in production. This section is to get you comfortable first. In a later post we'll discuss how to use pods with ReplicaSets or Deployments to make your pods more highly available and more useful.

I also want to mention that there are two ways to deploy objects in Kubernetes. The first way is through the command line and the second is through a manifest file. You will likely spend time in both of them where the manifest file is your configuration stored in version control and the command line would be used for troubleshooting purposes. The focus for this blog post will be on the manifests files.

### Deploy an Nginx pod from Manifest

To deploy nginx we first need to create a manifest file in YAML format. The code below will deploy our naked nginx container on our Kubernetes cluster. The file has comments for important lines so that you can see what each piece accomplishes.

```
apiVersion: v1 #version of the API to use
kind: Pod #What kind of object we're deploying
metadata: #information about our object we're deploying
  name: nginx-pod
spec: #specifications for our object
  containers:
  - name: nginx-container #the name of the container within the pod
    image: nginx #which container image should be pulled
    ports:
    - containerPort: 80 #the port of the container within the pod
```

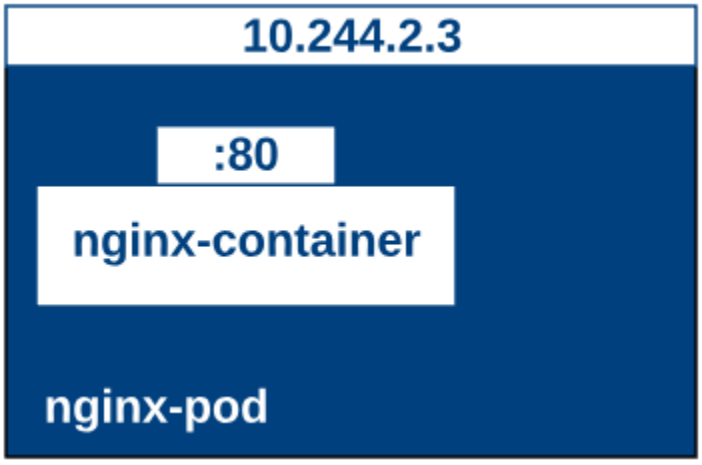
Code language: PHP (php)

Save the file and then we'll deploy it by running the following command from our cli:

```
kubectl apply -f [manifest file].yml
```

Code language: CSS (css)

Once deployed we should have a pod environment that looks similar to this although the IP address is likely different from mine.




To check on our pods, we can run:

```
kubectl get pods
```

Code language: JavaScript (javascript)

```
Erics-MacBook-Pro-2:git eshanks$ kubectl apply -f pod-manifest.yml
pod "nginx-pod" created
Erics-MacBook-Pro-2:git eshanks$ kubectl get pods
NAME        READY     STATUS    RESTARTS   AGE
nginx-pod   1/1       Running   0          6m
```

To get even more details on the pod that was deployed, we can run:

```
kubectl describe pod [pod name]
```

Code language: CSS (css)

The screenshot below omits a lot of the data (because it was difficult to fit on this page mainly) but it does show the events for the pod listed at the bottom. When you run this on your own you should see a wealth of knowledge about the pod.

# Kubernetes – Replica Sets

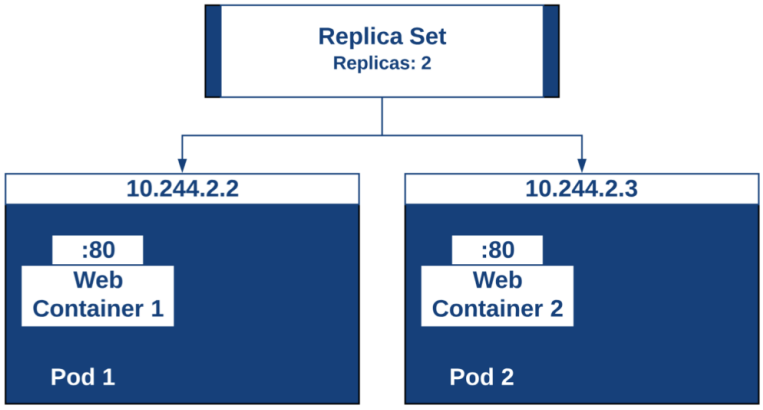
*By* ERIC SHANKS

In a previous post we covered the use of pods and deployed some "naked pods" in our Kubernetes cluster. In this post we'll expand our use of pods with Replica Sets.

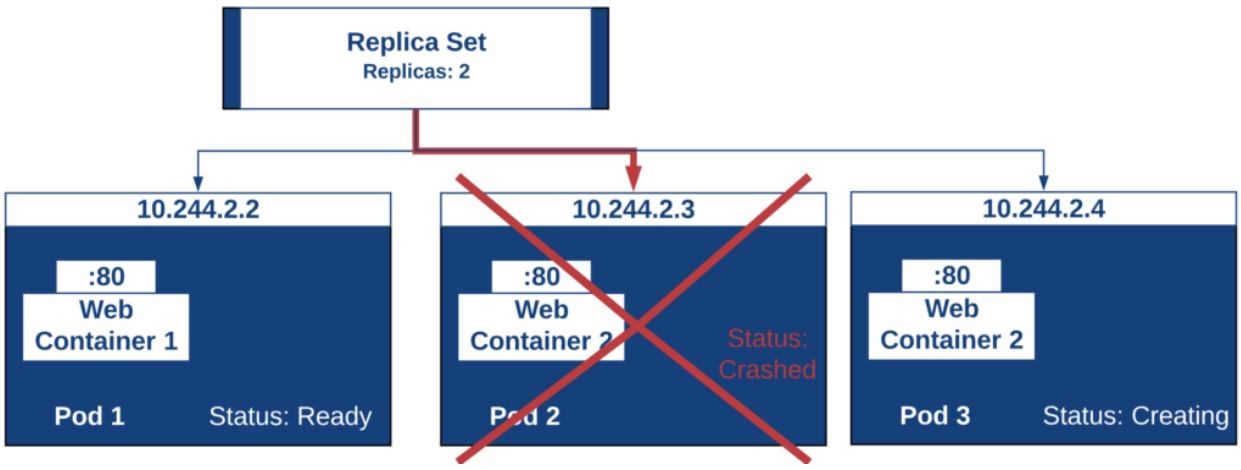
## Replica Sets – The Theory

One of the biggest reasons that we don't deploy naked pods in production is that they are not trustworthy. By this I mean that we can't count on them to always be running. Kubernetes doesn't ensure that a pod will continue running if it crashes. A pod could die for all kinds of reasons such as a node that it was running on had failed, it ran out of resources, it was stopped for some reason, etc. If the pod dies, it stays dead until someone fixes it which is not ideal, but with containers we should expect them to be short lived anyway, so let's plan for it.

Replica Sets are a level above pods that ensures a certain number of pods are always running. A Replica Set allows you to define the number of pods that need to be running at all times and this number could be "1". If a pod crashes, it will be recreated to get back to the desired state. For this reason, replica sets are preferred over a naked pod because they provide some high availability.




If one of the pods thats is part of a replica set crashes, one will be created to take its place.




# Replica Sets – In Action

Let's deploy a Replica Set from a manifest file so we can see what happens during the deployment. First, we'll need a manifest file to deploy. The manifest below will deploy nginx just like we did with the pods, except this time we'll use a Replica Set and specify that we should always have 2 replicas running in our cluster.

```
apiVersion: apps/v1 #version of the API to use
kind: ReplicaSet #What kind of object we're deploying
metadata: #information about our object we're deploying
  name: nginx-replicaset
spec: #specifications for our object
  replicas: 2 #The number of pods that should always be running
  selector: #which pods the replica set should be responsible for
    matchLabels:
      app: nginx #any pods with labels matching this I'm responsible
for.
  template: #The pod template that gets deployed
    metadata:
      labels: #A tag on the pods created
        app: nginx
    spec:
      containers:
      - name: nginx-container #the name of the container within the pod
        image: nginx #which container image should be pulled
        ports:
        - containerPort: 80 #the port of the container within the pod
```

Code language: PHP (php)

Let's deploy the Replica Sets from the manifest file by running a kubectl command and then afterwards we'll run a command to list our pods.

```
kubectl apply -f [manifest file].yml
kubectl get pods
```

Code language: CSS (css)

As you can see from my screenshot, we now have two pods running as we were expecting. Note: if you run the command too quickly, the pods might still be in a creating state. If this happens wait a second and run the get pod command again.

```
Erics-MacBook-Pro-2:git eshanks$ kubectl get pods
NAME                      READY     STATUS    RESTARTS   AGE
nginx-replicaset-5ml9k    1/1       Running   0          11m
nginx-replicaset-l94zd    1/1       Running   0          11m
```

I wonder what would happen if we manually deleted one of those pods that was in the Replica Set? Let's try it by running:

```
kubectl delete pod [pod name]
```

Code language: JavaScript (javascript)

Run another "get pod" command quickly to see what happens after you delete one of your pods.

```
Erics-MacBook-Pro-2:git eshanks$ kubectl delete pod nginx-replicaset-l94zd
pod "nginx-replicaset-l94zd" deleted
Erics-MacBook-Pro-2:git eshanks$ kubectl get pods
NAME                      READY     STATUS        RESTARTS   AGE
nginx-replicaset-5ml9k    1/1       Running       0          15m
nginx-replicaset-l94zd    0/1       Terminating   0          15m
nginx-replicaset-qrvvl    1/1       Running       0          6m
```

As you can see from my screenshot, I deleted a pod and then immediately ran another get pods command. You can see that one of my pods is in a Terminating status, but there is a new pod that is now running and thats because our Replica Set specified that two pods should be available. It is ensuring that we have that many at all times, even if one fails.

**Summary**

Now you know the basics of Replica Sets and why you'd use them over naked pods. Again, don't worry that you can't access your nginx container yet, we still haven't gotten to that yet but we're getting there. We're going to learn a few more things before this happens but we're well on our way now.

If you're done with your testing you can remove the Replica Set we created by running:

```
kubectl delete -f [mainfest file].yml
```

```
Events:
  Type    Reason     Age   From                Message
  ----    ------     ----  ----                -------
  Normal  Scheduled  7m    default-scheduler   Successfully assigned default/ng
inx-pod to kube-node2
  Normal  Pulling    55s   kubelet, kube-node2  pulling image "nginx"
  Normal  Pulled     55s   kubelet, kube-node2  Successfully pulled image "nginx
"
  Normal  Created    55s   kubelet, kube-node2  Created container
  Normal  Started    54s   kubelet, kube-node2  Started container
```

**Summary**

That's it for pods. I should note that you probably can't connect to your container. Don't worry, you're not supposed to be able to yet. We'll continue this conversation in a future post where we learn about other objects including Deployments, ReplicaSets and Services.

If you're all done, you can delete your pod from your cluster by running:

```
kubectl delete -f [mainfest file].yml
```

# Kubernetes – Deployments

*By* ERIC SHANKS

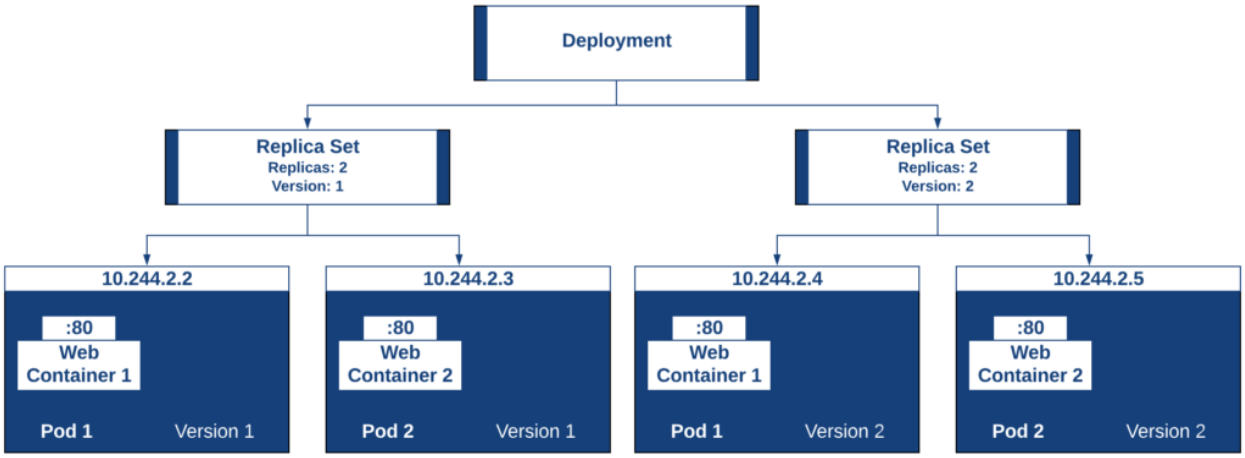
After following the previous posts, we should feel pretty good about deploying our pods and ensuring they are highly available. We've learned about naked pods and then replica sets to make those pods more HA, but what about when we need to create a new version of our pods? We don't want to have an outage when our pods are replaced with a new version do we? This is where "Deployments" comes into play.

# Deployments – The Theory

If you couldn't tell from the introduction to this post, Deployments are an amazing object for us to handle changes to our applications. While replica sets are used to ensure a desired number of pods are always running and handle our high availability concerns, Deployments ensure that we can safely rollout new versions of our pods safely and without outages. They also make it possible to rollback a deployment if there is some terrible issue with the new version.

Now, I want to make sure that we don't think that Deployments replace replica sets because they don't. Deployments are a construct a level above replica sets and actually manage the replica set objects.

So Deployments manage replica sets and replica sets manage pods and pods manage containers.




# Deployments – In Action

Just as we have done with the other posts in this series we'll start with creating a manifest file of our desired state configuration. Kubernetes will ensure that this desired state is applied and any items that need to be orchestrated will be handled to meet this configuration.

We'll start by adding Deployment information to the replica set manifest we built previously. Remember that a Deployment sits at a level above replica sets so we can add the new construct to our manifest file we created in the replica set post. As usual, comments have been added to the file so it's easier to follow.

```
apiVersion: apps/v1 #version of the API to use
kind: Deployment #What kind of object we're deploying
metadata: #information about our object we're deploying
  name: nginx-deployment #Name of the deployment
  labels: #A tag on the deployments created
    app: nginx
spec: #specifications for our object
  replicas: 2 #The number of pods that should always be running
  selector: #which pods the replica set should be responsible for
    matchLabels:
      app: nginx #any pods with labels matching this I'm responsible
for.
  template: #The pod template that gets deployed
    metadata:
      labels: #A tag on the replica sets created
        app: nginx
    spec:
      containers:
      - name: nginx-container #the name of the container within the pod
        image: nginx #which container image should be pulled
        ports:
```

```
        - containerPort: 80 #the port of the container within the pod
```
Code language: PHP (php)

Now we'll deploy this configuration to our kubernetes cluster by running:

```
kubectl apply -f [manifest file].yml
```
Code language: CSS (css)

Once the deployment has been applied to our cluster, we'll run:

```
kubectl get deployments
```
Code language: JavaScript (javascript)

The results of our deployment should look similar to the following screenshot.

```
Erics-MacBook-Pro-2:git eshanks$ kubectl apply -f Deployment-manifest.yml
deployment.apps "nginx-deployment" created
Erics-MacBook-Pro-2:git eshanks$ kubectl get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   2         2         2            2           6m
```

There are several columns listed here but the gist of it is:

- DESIRED – 2 replicas of the application were in our configuration.
- CURRENT – 2 replicas are currently running.
- UP-TO-DATE – 2 replicas that have been updated to get to the configuration we specified.
- AVAILABLE – 2 replicas are available for use

This information may not seem to interesting at the moment, but remember that Deployments can take an existing set and perform a rolling update on them. When this occurs the information shown here may be more important.

Let's try to update our deployment and see what happens. Let's modify our deployment manifest file to increase the number of replicas and also change the version of nginx that is being deployed. The file below makes those changes for you.

```
apiVersion: apps/v1 #version of the API to use
kind: Deployment #What kind of object we're deploying
metadata: #information about our object we're deploying
  name: nginx-deployment #Name of the deployment
  labels: #A tag on the deployments created
    app: nginx
spec: #specifications for our object
  strategy:
    type: RollingUpdate
    rollingUpdate: #Update Pods a certain number at a time
      maxUnavailable: 1 #Total number of pods that can be unavailable at
once
      maxSurge: 1 #Maximum number of pods that can be deployed above
desired state
  replicas: 6 #The number of pods that should always be running
  selector: #which pods the replica set should be responsible for
    matchLabels:
```

```
      app: nginx #any pods with labels matching this I'm responsible
for.
  template: #The pod template that gets deployed
    metadata:
      labels: #A tag on the replica sets created
        app: nginx
    spec:
      containers:
      - name: nginx-container #the name of the container within the pod
        image: nginx:1.7.9 #which container image should be pulled
        ports:
        - containerPort: 80 #the port of the container within the pod
```

Code language: PHP (php)

We can apply the configuration file again and then check our Deployment status using some familiar commands:

```
kubectl apply -f [new manifest file].yml
kubectl get deployments
```

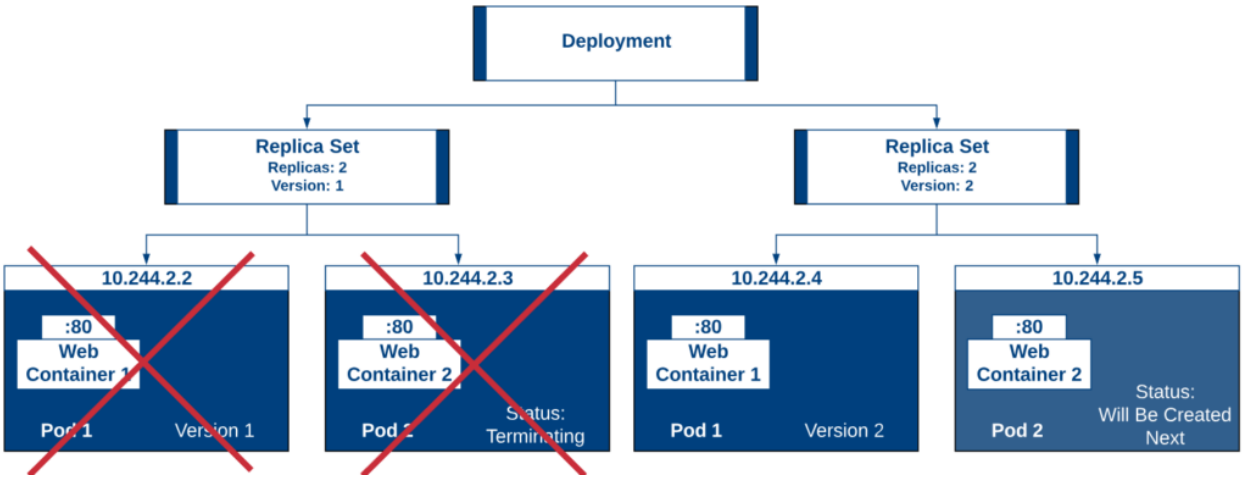
Code language: JavaScript (javascript)

Now we see some more interesting information. Depending on how fast you were with your commands, you might get different results so use the screenshot below for this discussion.

```
Erics-MacBook-Pro-2:git eshanks$ kubectl apply -f Deployment-manifest2.yml
deployment.apps "nginx-deployment" configured
Erics-MacBook-Pro-2:git eshanks$ kubectl get deployment
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   6         7         2            2           7m
```

Since I ran the get deployments command before the whole thing was finished, my desired state doesn't match the current state. In fact, the current state has MORE replicas than are desired. The reason for this is that the new pods are deployed first and then the old pods are removed.

In the example below, you can see that we're going to remove the version 1 containers while we spin up version 2. To do this we spin up one pod with the new version and then start terminating the old pods. When this is done, the next pod will be created in version 2 and eventually the last pod in version 1 will be removed.




The number of pods being deployed above desired state is configured in our manifest in the maxSurge specification. We could have made this number 2 and then two pods would be created at a time and two removed.

Now, sometimes you have a bigger update than the one we used for a demonstration. In that case you might not want to keep running the get deployments command. You can run:

```
kubectl rollout status deployment [deployment name]
```

Code language: CSS (css)

This command will show you a running list of whats happening with the rollout. I ran this during the deploy and here is an example of what you might see.

```
Erics-MacBook-Pro-2:git eshanks$ kubectl rollout status deployment nginx-deployment
Waiting for rollout to finish: 3 out of 6 new replicas have been updated...
Waiting for rollout to finish: 3 out of 6 new replicas have been updated...
Waiting for rollout to finish: 3 out of 6 new replicas have been updated...
Waiting for rollout to finish: 3 out of 6 new replicas have been updated...
Waiting for rollout to finish: 4 out of 6 new replicas have been updated...
Waiting for rollout to finish: 4 out of 6 new replicas have been updated...
Waiting for rollout to finish: 1 old replicas are pending termination...
Waiting for rollout to finish: 1 old replicas are pending termination...
Waiting for rollout to finish: 1 old replicas are pending termination...
Waiting for rollout to finish: 1 old replicas are pending termination...
Waiting for rollout to finish: 1 old replicas are pending termination...
Waiting for rollout to finish: 5 of 6 updated replicas are available...
Waiting for rollout to finish: 5 of 6 updated replicas are available...
Waiting for rollout to finish: 5 of 6 updated replicas are available...
deployment "nginx-deployment" successfully rolled out
```

**Summary**

Deployments are just another step along the way in learning Kubernetes. We've gotten to a pretty good point here and now we know how we can deploy and update our deployments in our cluster. We STILL haven't accessed our containers yet, but just hold on, we're at the cusp of having a working container.

When you're done messing around with your cluster you can delete the deployment by running:

```
kubectl delete -f [manifest file].yml
```

# Kubernetes – Services and Labels

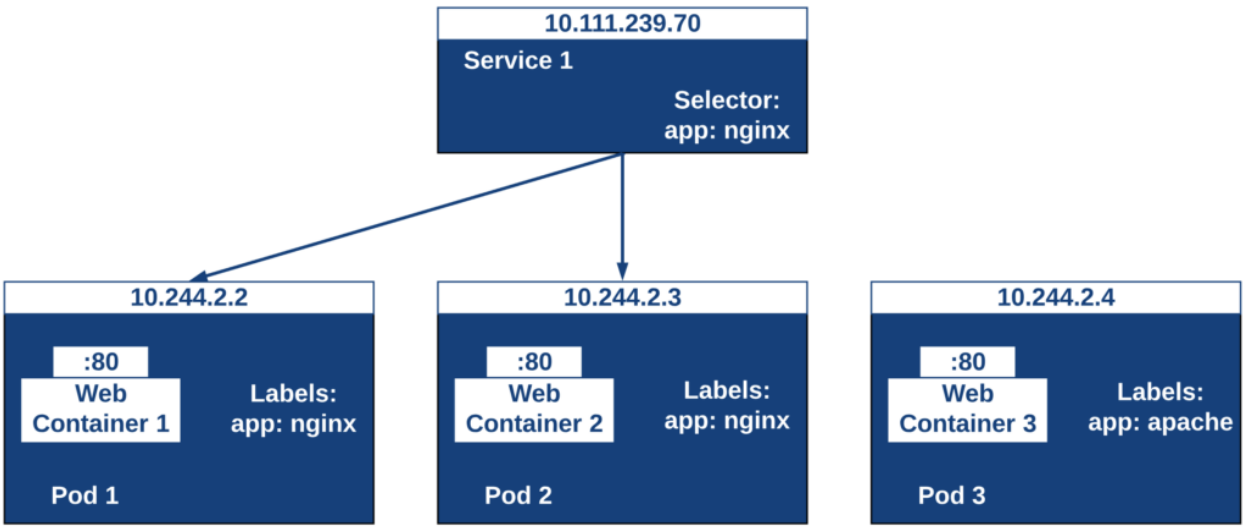
If you've been following the series, you may be thinking that we've built ourselves a problem. You'll recall that we've now learned about Deployments so that we can roll out new pods when we do upgrades, and replica sets can spin up new pods when one dies. Sounds great, but remember that each of those containers has a different IP address. Now, I know we haven't accessed any of those pods yet, but you can imagine that it would be a real pain to have to go lookup an IP Address every time a pod was replaced, wouldn't it? This post covers Kubernetes Services and how they are used to address this problem, and at the end of this post, we'll access one of our pods … finally.

## Services – The Theory

In the broadest sense, Kubernetes Services tie our pods together and provide a front end resource to access. You can think of them like a load balancer that automatically knows which servers it is trying to load balance. Since our pods may be created and destroyed even without our intervention, we'll need a stable way to access them at a single address every time. Services give us a static resource to access that abstracts the pods behind them.

OK, we probably don't have a hard time understanding that a Service sits in front of our pods and distributes requests to them, but we might be asking how the service knows which pods it should be providing a front end for. I mean of course we'll have pods for different reasons running in our cluster and we'll need to assign services to our pods somehow. Well that introduces a really cool specification called labels. If you were paying close attention in previous posts, we had labels on some of our pods already. Labels are just a key value pair, or tag, that provides metadata on our objects. A Kubernetes Service can select the pods it is supposed to abstract through a label selector. Neat huh?

Take a look at the example diagram below. Here we have a single Service that is front-ending two of our pods. The two pods have labels named "app: nginx" and the Service has a label selector that is looking for those same labels. This means that even though the pods might change addresses, as long as they are labeled correctly, the service, which stays with a constant address, will send traffic to them.




You might also notice that there is a Pod3 that has a different label. The Service 1 service won't front end that pod so we'd need another service that would take care of that for us. Now we'll use a service to access our nginx pod later in this post, but remember that many apps are multiple tiers. Web talks to app which talks to database. In that scenario all three of those tiers may need a consistent service for the others to communicate properly all while pods are spinning up and down.

The way in which services are able to send traffic to the backend pods is through the use of the kube-proxy. Every node of our Kubernetes cluster runs a proxy called the kube-proxy. This proxy listens to the master node API for services as well as endpoints (covered in a later post). Whenever it finds a new service, the kube-proxy opens a random port on the node in which it belongs. This port proxies connections to the backend pods.

# Services and Labels – In Action

I know you're eager to see if your cluster is really working or not, so let's get to deploying our deployment manifest like we built in a previous post and then a Service to front-end that deployment. When we're done, we'll pull it open in a web browser to see an amazing webpage.

Let's start off by creating a new manifest file and deploying it to our Kubernetes cluster. The file is below and has two objects, Deployment & Service within the same file.

```
apiVersion: apps/v1 #version of the API to use
kind: Deployment #What kind of object we're deploying
metadata: #information about our object we're deploying
  name: nginx-deployment #Name of the deployment
  labels: #A tag on the deployments created
    app: nginx
spec: #specifications for our object
  replicas: 2 #The number of pods that should always be running
  selector: #which pods the replica set should be responsible for
```

```
    matchLabels:
      app: nginx #any pods with labels matching this I'm responsible
for.
  template: #The pod template that gets deployed
    metadata:
      labels: #A tag on the replica sets created
        app: nginx
    spec:
      containers:
      - name: nginx-container #the name of the container within the pod
        image: nginx #which container image should be pulled
        ports:
        - containerPort: 80 #the port of the container within the pod
---
apiVersion: v1 #version of the API to use
kind: Service #What kind of object we're deploying
metadata: #information about our object we're deploying
  name: ingress-nginx #Name of the service
spec: #specifications for our object
  type: NodePort #Ignore for now discussed in a future post
  ports: #Ignore for now discussed in a future post
  - name: http
    port: 80
    targetPort: 80
    nodePort: 30001
    protocol: TCP
  selector: #Label selector used to identify pods
    app: nginx
```

Code language: PHP (php)

We can deploy this by running a command we should be getting very familiar with at this point.

```
kubectl apply -f [manifest file].yml
```

Code language: CSS (css)

After the the manifest has been deployed we can look at the pods, replica sets or deployments like we have before and now we can also look at our services by running:

```
kubectl get services
```

Code language: JavaScript (javascript)

```
Erics-MacBook-Pro-2:git eshanks$ kubectl get services
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
ingress-nginx   NodePort    10.110.210.31   <none>        80:30001/TCP     8m
kubernetes      ClusterIP   10.96.0.1       <none>        443/TCP          10d
```

Neat! Now we have a Service named ingress-nginx which we defined in our manifest file. We also have a Kubernetes Service which, for now, we'll ignore. Just know this is used to run the cluster. But do take a second to notice the ports column. Our ingress-ngnix service shows

80:30001/TCP. This will be discussed further in a future post, but the important thing is that the port after the colon ":" is the port we'll access the service on from our computer.

Here's the real test, can we open a web browser and put in an IP Address of one of our Kubernetes nodes on port 30001 and get an nginx page?


**Summary**

Well the result isn't super exciting. We have a basic nginx welcome page which isn't really awe inspiring, but we did finally access an app on our Kubernetes cluster and it was by using Services and Labels coupled with our pods that we've been learning. Stay tuned for the next post where we dive deeper into Kubernetes.

# Kubernetes – Services and Labels

*By* ERIC SHANKS

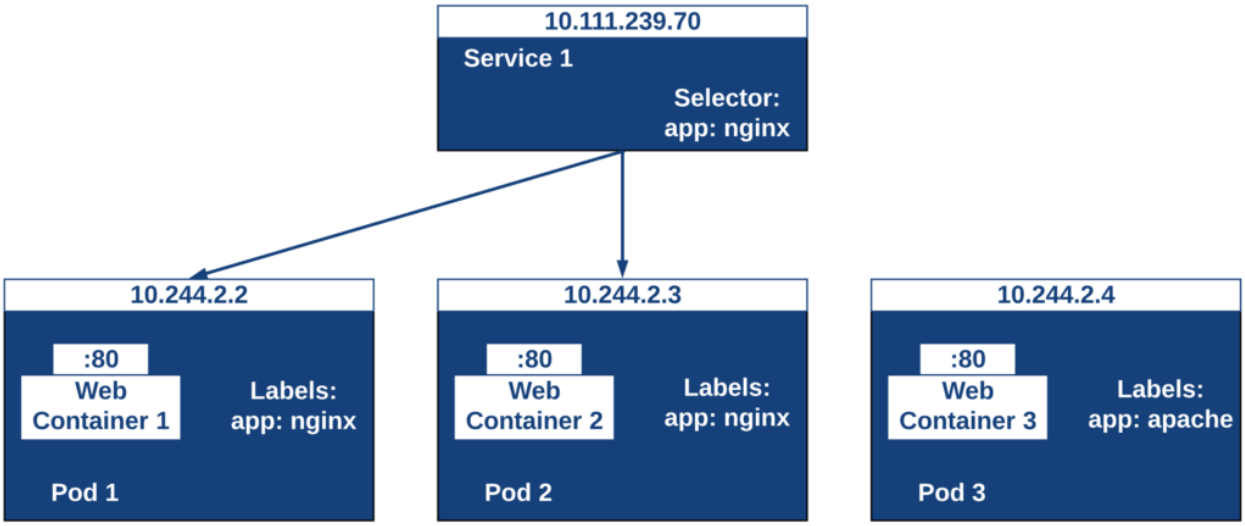
If you've been following the series, you may be thinking that we've built ourselves a problem. You'll recall that we've now learned about Deployments so that we can roll out new pods when we do upgrades, and replica sets can spin up new pods when one dies. Sounds great, but remember that each of those containers has a different IP address. Now, I know we haven't accessed any of those pods yet, but you can imagine that it would be a real pain to have to go lookup an IP Address every time a pod was replaced, wouldn't it? This post covers Kubernetes Services and how they are used to address this problem, and at the end of this post, we'll access one of our pods … finally.

## Services – The Theory

In the broadest sense, Kubernetes Services tie our pods together and provide a front end resource to access. You can think of them like a load balancer that automatically knows which servers it is trying to load balance. Since our pods may be created and destroyed even without our intervention, we'll need a stable way to access them at a single address every time. Services give us a static resource to access that abstracts the pods behind them.

OK, we probably don't have a hard time understanding that a Service sits in front of our pods and distributes requests to them, but we might be asking how the service knows which pods it should be providing a front end for. I mean of course we'll have pods for different reasons running in our cluster and we'll need to assign services to our pods somehow. Well that introduces a really cool specification called labels. If you were paying close attention in previous posts, we had labels on some of our pods already. Labels are just a key value pair, or tag, that provides metadata on our objects. A Kubernetes Service can select the pods it is supposed to abstract through a label selector. Neat huh?

Take a look at the example diagram below. Here we have a single Service that is front-ending two of our pods. The two pods have labels named "app: nginx" and the Service has a label selector that is looking for those same labels. This means that even though the pods might change addresses, as long as they are labeled correctly, the service, which stays with a constant address, will send traffic to them.




You might also notice that there is a Pod3 that has a different label. The Service 1 service won't front end that pod so we'd need another service that would take care of that for us. Now we'll use a service to access our nginx pod later in this post, but remember that many apps are multiple tiers. Web talks to app which talks to database. In that scenario all three of those tiers may need a consistent service for the others to communicate properly all while pods are spinning up and down.

The way in which services are able to send traffic to the backend pods is through the use of the kube-proxy. Every node of our Kubernetes cluster runs a proxy called the kube-proxy. This proxy listens to the master node API for services as well as endpoints (covered in a later post). Whenever it finds a new service, the kube-proxy opens a random port on the node in which it belongs. This port proxies connections to the backend pods.

# Services and Labels – In Action

I know you're eager to see if your cluster is really working or not, so let's get to deploying our deployment manifest like we built in a previous post and then a Service to front-end that deployment. When we're done, we'll pull it open in a web browser to see an amazing webpage.

Let's start off by creating a new manifest file and deploying it to our Kubernetes cluster. The file is below and has two objects, Deployment & Service within the same file.

```
apiVersion: apps/v1 #version of the API to use
kind: Deployment #What kind of object we're deploying
metadata: #information about our object we're deploying
  name: nginx-deployment #Name of the deployment
  labels: #A tag on the deployments created
    app: nginx
spec: #specifications for our object
  replicas: 2 #The number of pods that should always be running
  selector: #which pods the replica set should be responsible for
    matchLabels:
      app: nginx #any pods with labels matching this I'm responsible
for.
  template: #The pod template that gets deployed
    metadata:
      labels: #A tag on the replica sets created
        app: nginx
    spec:
      containers:
      - name: nginx-container #the name of the container within the pod
        image: nginx #which container image should be pulled
        ports:
        - containerPort: 80 #the port of the container within the pod
---
apiVersion: v1 #version of the API to use
kind: Service #What kind of object we're deploying
metadata: #information about our object we're deploying
  name: ingress-nginx #Name of the service
spec: #specifications for our object
  type: NodePort #Ignore for now discussed in a future post
  ports: #Ignore for now discussed in a future post
  - name: http
    port: 80
    targetPort: 80
    nodePort: 30001
    protocol: TCP
  selector: #Label selector used to identify pods
```

```
    app: nginx
```
Code language: PHP (php)

We can deploy this by running a command we should be getting very familiar with at this point.

```
kubectl apply -f [manifest file].yml
```
Code language: CSS (css)

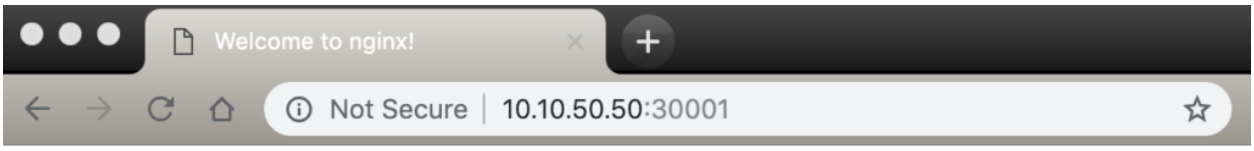
After the the manifest has been deployed we can look at the pods, replica sets or deployments like we have before and now we can also look at our services by running:

```
kubectl get services
```
Code language: JavaScript (javascript)

```
Erics-MacBook-Pro-2:git eshanks$ kubectl get services
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
ingress-nginx   NodePort    10.110.210.31   <none>        80:30001/TCP     8m
kubernetes      ClusterIP   10.96.0.1       <none>        443/TCP          10d
```

Neat! Now we have a Service named ingress-nginx which we defined in our manifest file. We also have a Kubernetes Service which, for now, we'll ignore. Just know this is used to run the cluster. But do take a second to notice the ports column. Our ingress-ngnix service shows 80:30001/TCP. This will be discussed further in a future post, but the important thing is that the port after the colon ":" is the port we'll access the service on from our computer.

Here's the real test, can we open a web browser and put in an IP Address of one of our Kubernetes nodes on port 30001 and get an nginx page?




# Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

*Thank you for using nginx.*

**Summary**

Well the result isn't super exciting. We have a basic nginx welcome page which isn't really awe inspiring, but we did finally access an app on our Kubernetes cluster and it was by using Services and Labels coupled with our pods that we've been learning. Stay tuned for the next post where we dive deeper into Kubernetes.

# Kubernetes – Endpoints

*By* ERIC SHANKS

It's quite possible that you could have a Kubernetes cluster but never have to know what an endpoint is or does, even though you're using them behind the scenes. Just in case you need to use one though, or if you need to do some troubleshooting, we'll cover the basics of Kubernetes endpoints in this post.

## Endpoints – The Theory

During the post where we first learned about Kubernetes Services, we saw that we could use labels to match a frontend service with a backend pod automatically by using a selector. If any new pods had a specific label, the service would know how to send traffic to it. Well the way that the service knows to do this is by adding this mapping to an endpoint. Endpoints track the IP Addresses of the objects the service send traffic to. When a service selector matches a pod label, that IP Address is added to your endpoints and if this is all you're doing, you don't really need to know much about endpoints. However, you can have Services where the endpoint is a server outside of your cluster or in a different namespace (which we haven't covered yet).

What you should know about endpoints is that there is a list of addresses your services will send traffic and its managed through endpoints. Those endpoints can be updated automatically through labels and selectors, or you can manually configure your endpoints depending on your use case.

## Endpoints – In Action

Let's take a look at some endpoints that we've used in our previous manifests. Lets deploy this manifest as we did in our previous posts and take a look to see what our endpoints are doing.

```
apiVersion: apps/v1 #version of the API to use
kind: Deployment #What kind of object we're deploying
metadata: #information about our object we're deploying
  name: nginx-deployment #Name of the deployment
  labels: #A tag on the deployments created
    app: nginx
spec: #specifications for our object
  replicas: 2 #The number of pods that should always be running
  selector: #which pods the replica set should be responsible for
    matchLabels:
      app: nginx #any pods with labels matching this I'm responsible
for.
  template: #The pod template that gets deployed
```

```
    metadata:
      labels: #A tag on the replica sets created
        app: nginx
    spec:
      containers:
      - name: nginx-container #the name of the container within the pod
        image: nginx #which container image should be pulled
        ports:
        - containerPort: 80 #the port of the container within the pod
---
apiVersion: v1 #version of the API to use
kind: Service #What kind of object we're deploying
metadata: #information about our object we're deploying
  name: ingress-nginx #Name of the service
spec: #specifications for our object
  type: NodePort #Ignore for now discussed in a future post
  ports: #Ignore for now discussed in a future post
  - name: http
    port: 80
    targetPort: 80
    nodePort: 30001
    protocol: TCP
  selector: #Label selector used to identify pods
    app: nginx
```

Code language: PHP (php)

We can deploy this manifest running:

```
kubectl apply -f [manifest file].yml
```

Code language: CSS (css)

Now that its done we should be able to see the endpoints that were automatically created when our service selector matched our pod label. To see this we can query our endpoints through kubectl.

```
kubectl get endpoints
```

Code language: JavaScript (javascript)

Your results will likely have different addresses than mine, but for reference, here are the endpoints that were returned.

```
Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl get endpoints
NAME            ENDPOINTS                          AGE
ingress-nginx   10.244.1.143:80,10.244.2.204:80   23h
kubernetes      10.10.50.50:6443                   11d
```

The ingress-nginx endpoint is the one we're focusing on and you can see it has two endpoints listed, both on port 80.

Now those endpoints should be the IP addresses of our pods that we deployed in our manifest. To test this, lets use the get pods command with the -o wide switch to show more output.

```
kubectl get pods -o wide
```

Code language: JavaScript (javascript)

```
Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl get pods -o wide
NAME                                READY     STATUS    RESTARTS   AGE       IP             NODE
nginx-deployment-6d99f57cb4-ngc4g   1/1       Running   0          1d        10.244.2.204   kube-node2
nginx-deployment-6d99f57cb4-x5gsm   1/1       Running   0          1d        10.244.1.143   kube-node1
```

You can see that the IP addresses associated with the pods matches the endpoints. So we proved that the endpoints are matching under the hood.

How about if we want to manually edit our endpoints if we don't have a selector? Maybe we're trying to have a resource to access an external service that doesn't live within our Kubernetes cluster? We could create our own endpoint to do this for us. A great example might be an external database service for our web or app containers.

Let's look at an example where we're using an endpoint to access an external resource from a container. In this case we'll access a really simple web page just for a test. For reference, I accessed this service from my laptop first to prove that it's working. If you're doing this in your lab, you'll need to spin up a web server and modify the IP Addresses accordingly.


Hello Kube Users Welcome to the Test Web Page!!!! ----------

I know that it's not very exciting, but it'll get the job done. Next up we'll deploy our Endpoint and a service with no selector. The following manifest should do the trick. Notice the ports used and the IP Address specified in the endpoint.

```
  kind: "Service"
  apiVersion: "v1"
  metadata:
    name: "external-web"
  spec:
    ports:
      -
        name: "apache"
        protocol: "TCP"
        port: 80
        targetPort: 80
```

```
---
  kind: "Endpoints"
  apiVersion: "v1"
  metadata:
    name: "external-web"
  subsets:
    -
      addresses:
        -
          ip: "10.10.50.53" #The IP Address of the external web server
      ports:
        -
          port: 80
          name: "apache"
```
Code language: PHP (php)

Then we'll deploy the manifest file to get our service and endpoint built.

```
kubectl apply -f [manifest file]
```
Code language: CSS (css)

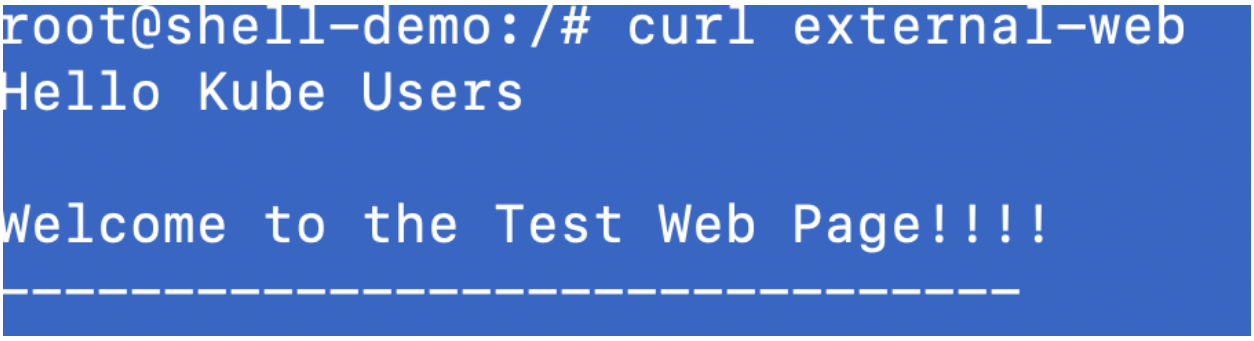
Let's check the endpoint just to make sure it looks correct.

```
kubectl get endpoints
```
Code language: JavaScript (javascript)

```
Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl get endpoints
NAME            ENDPOINTS                          AGE
external-web    10.10.50.53:80                     13m
ingress-nginx   10.244.1.143:80,10.244.2.204:80    1d
kubernetes      10.10.50.50:6443                   12d
```

Once the service and endpoint are deployed, we'll deploy a quick container into our cluster so we can use it to curl our web page as a test. We'll use the imperative commands instead of a manifest file this time.

```
kubectl create -f https://k8s.io/examples/application/shell-demo.yaml

kubectl exec -it shell-demo -- /bin/bash

apt-get update
apt-get install curl
curl external-web
```
Code language: JavaScript (javascript)




The curl command worked when performing a request against the "external-web" service, so we know it's working!

**Summary**

If you've been following the series so far, you've already been using endpoints but just didn't know it. You can add your own endpoints manually if you have a use case such as accessing a remote service like a database but for now its probably enough just to get the idea that they exist and that you can modify them if necessary.

# Kubernetes – Service Publishing

*By* ERIC SHANKS

A critical part of deploying containers within a Kubernetes cluster is understanding how they use the network. In previous posts we've deployed pods and services and were able to access them from a client such as a laptop, but how did that work exactly? I mean, we had a bunch of ports configured in our manifest files, so what do they mean? And what do we do if we have more than one pod that wants to use the same port like 443 for https?

This post will cover three options for publishing our services for accessing our applications.

# ClusterIP – The Theory

Whenever a service is created a unique IP address is assigned to the service and that IP Address is called the "ClusterIP". Now since we need our Services to stay consistent, the IP address of that Service needs to stay the same. Remember that pods come and go, but services will need to be pretty consistent so that we have an address to always access for our applications or users.

Ok, big deal right. Services having an IP assigned to them probably doesn't surprise anyone, but what you should know is that this ClusterIP isn't accessible from outside of the Kubernetes cluster. This is an internal IP only meaning that other pods can use a Services Cluster IP to communicate between them but we can't just put this IP address in our web browser and expect to get connected to the service in our Kubernetes cluster.

# NodePort – The Theory

NodePort might seem familiar to you. Thats because we used NodePort when we deployed our sample deployment in the Services and Labels post. A NodePort exposes a service on each node's IP address on a specific port. NodePort doesn't replace ClusterIP however, all it does is direct traffic to the ClusterIP from outside the cluster.

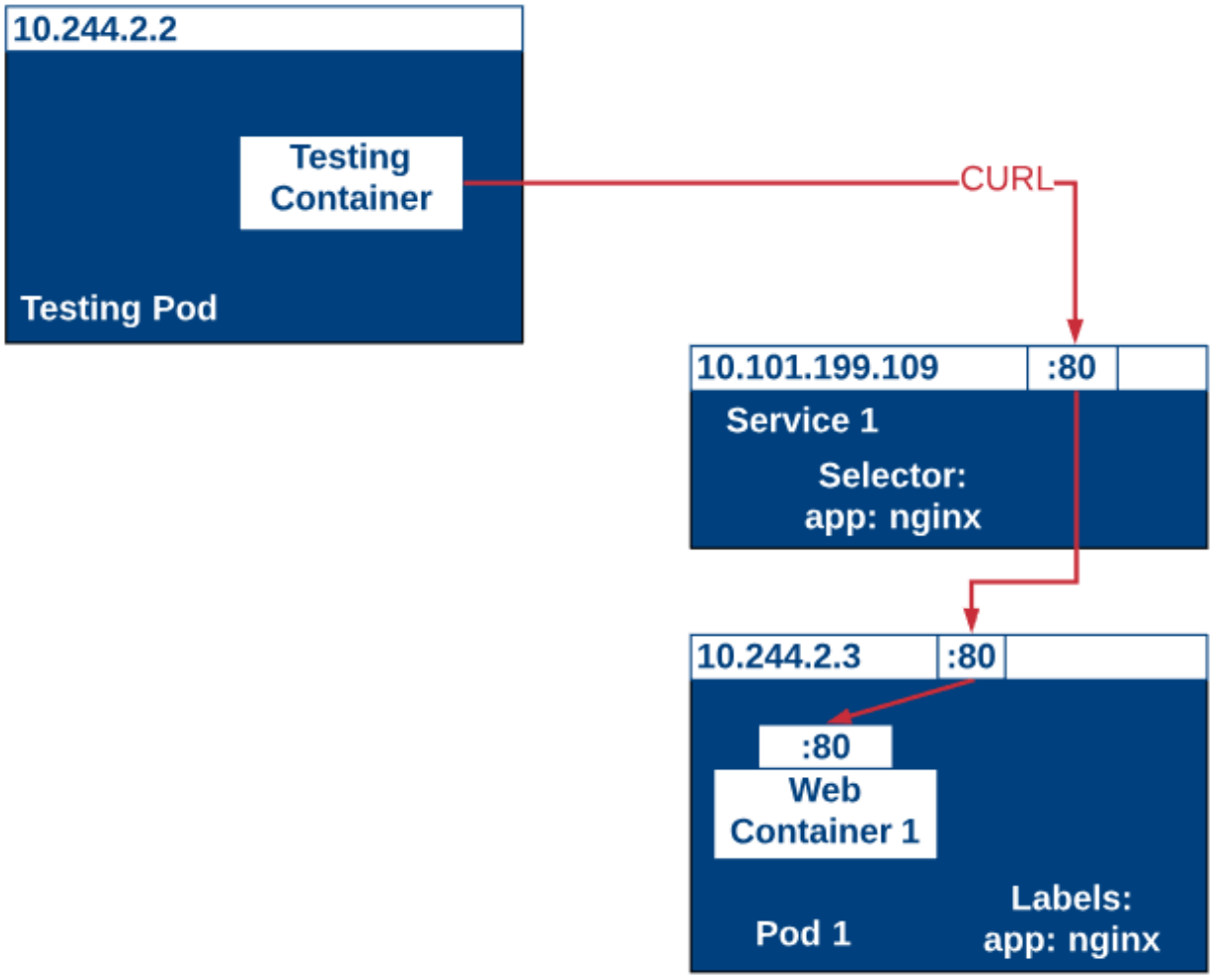
Some important information about NodePorts is that you can just publish your service on any port you want. For example if you have a web container deployed you'd likely be tempted to use a NodePort of 443 so that you can use a standard https port. This won't work since NodePort must be within the port range 30000-32767. You can specify which port you want to use as long as its in this range, which is what we did in our previous post, but be sure it doesn't conflict with another service in your cluster. If you don't care what port it is, don't specify one and your cluster will randomly assign one for you.

# LoadBalancer – The Theory

LoadBalancers are not going to be covered in depth in this post. What you should know about them right now is that they won't work if your cluster is build on-premises, like on a vSphere enviornment. If you're using a cloud service like Amazon Elastic Container Service for Kubernetes (EKS) or other cloud provider's k8s solution, then you can specify a load balancer in your manifest file. What it would do is spin up a load balancer in the cloud and point the load balancer to your service. This would allow you to use port 443 for example on your load balancer and direct traffic to one of those 30000 or higher ports.

# ClusterIP – In Action

As we mentioned, we can't access our containers from outside our cluster by using just a ClusterIP so we'll deploy a test container within the Kubernetes cluster and run a curl command against our nginx service. The picture below describes the process we'll be testing.




First, lets deploy our manifest which includes our service and nginx pod and container.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
```

```
      containers:
      - name: nginx-container
        image: nginx
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: ingress-nginx
spec:
  type: ClusterIP
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
  selector:
    app: nginx
```

We can deploy the manifest via:

```
kubectl apply -f [manifest file].yml
```

Code language: CSS (css)

Once our pod and service has been deployed we can look at the service information to find the ClusterIP

```
Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl get services
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
ingress-nginx   ClusterIP   10.101.199.109   <none>        80/TCP
kubernetes      ClusterIP   10.96.0.1        <none>        443/TCP
```

We can see that the ClusterIP for our ingress-nginx service is 10.101.199.109. So next, we're going to deploy another container just to run a curl command from within the cluster. To do this we'll run some imperative commands instead of the declarative manifest files.

```
kubectl create -f https://k8s.io/examples/application/shell-demo.yaml
kubectl exec -it shell-demo -- /bin/bash
```

Code language: JavaScript (javascript)

Once you've run the two commands above, you'll have a new pod named shell-demo and you've gotten an interactive terminal session into the container. Now we need to update the container and install curl.

```
apt-get update
apt-get install curl
curl [CLUSTERIP]
```

Code language: JavaScript (javascript)

```
root@shell-demo:/# curl 10.101.199.109
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial,
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
```

As you can see, we can communicate with this service by using the ClusterIP from within the cluster.

To stop this test you can exit the interactive shell by ctrl+d. Then we can delete the test pod by running:

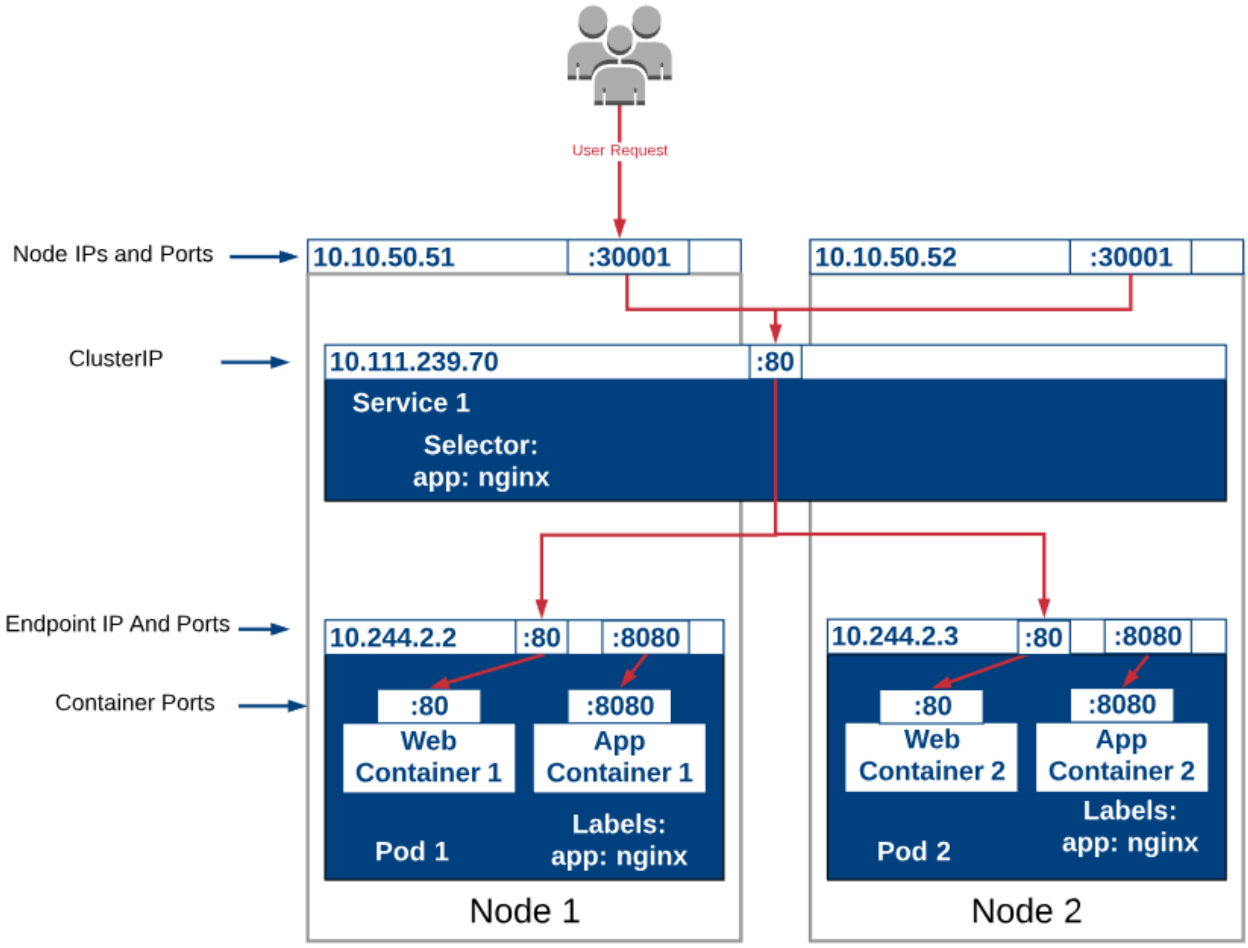
```
kubectl delete pod shell-demo
```
Code language: JavaScript (javascript)

Then we can remove our service and pod by running:

```
kubectl delete -f [manifest file].yml
```
Code language: CSS (css)

# NodePort – In Action

In this test, we'll access our backend pods through a NodePort from outside our Kubernetes cluster. The diagram below should give you a good idea of where the ClusterIP, NodePort and Containers fit in. Additional containers (app) were added to help better understand how they might fit in.




Lets deploy another service and pod where we'll also specify a NodePort of 30001. Below is another declarative manifest files.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx-container
```

```
        image: nginx
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: ingress-nginx
spec:
  type: NodePort
  ports:
  - name: http
    port: 80
    targetPort: 80
    nodePort: 30001
    protocol: TCP
  selector:
    app: nginx
```

We can deploy our manifest file with:

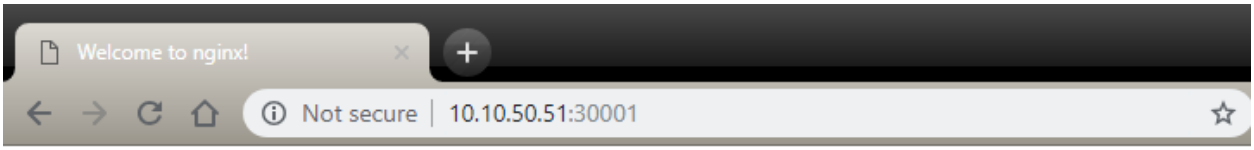
```
kubectl apply -f [manifest file].yml
```

Code language: CSS (css)

Once it's deployed we can look at the services again. You'll notice this is the same command as we ran earlier but this time there are two ports listed. The second port is whats mapped to our NodePort.

```
Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl get service
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
ingress-nginx   NodePort    10.101.248.193   <none>        80:30001/TCP
kubernetes      ClusterIP   10.96.0.1        <none>        443/TCP
```

If we access the IP Address of one of our nodes with the port we specified, we can see our nginx page.




# Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

*Thank you for using nginx.*

And thats it. Now we know how we can publish our services both externally and internally. To delete your pods run:

```
kubectl delete -f [manifest file].yml
```
Code language: CSS (css)

**Summary**

There are different ways to publish your services depending on what your goals are. We'll learn about other ways to expose your services externally in a future post, but for now we've got a few weapons in our arsenal to expose our pods to other pods or externally to the clients.

# Kubernetes – Namespaces

*By* ERIC SHANKS

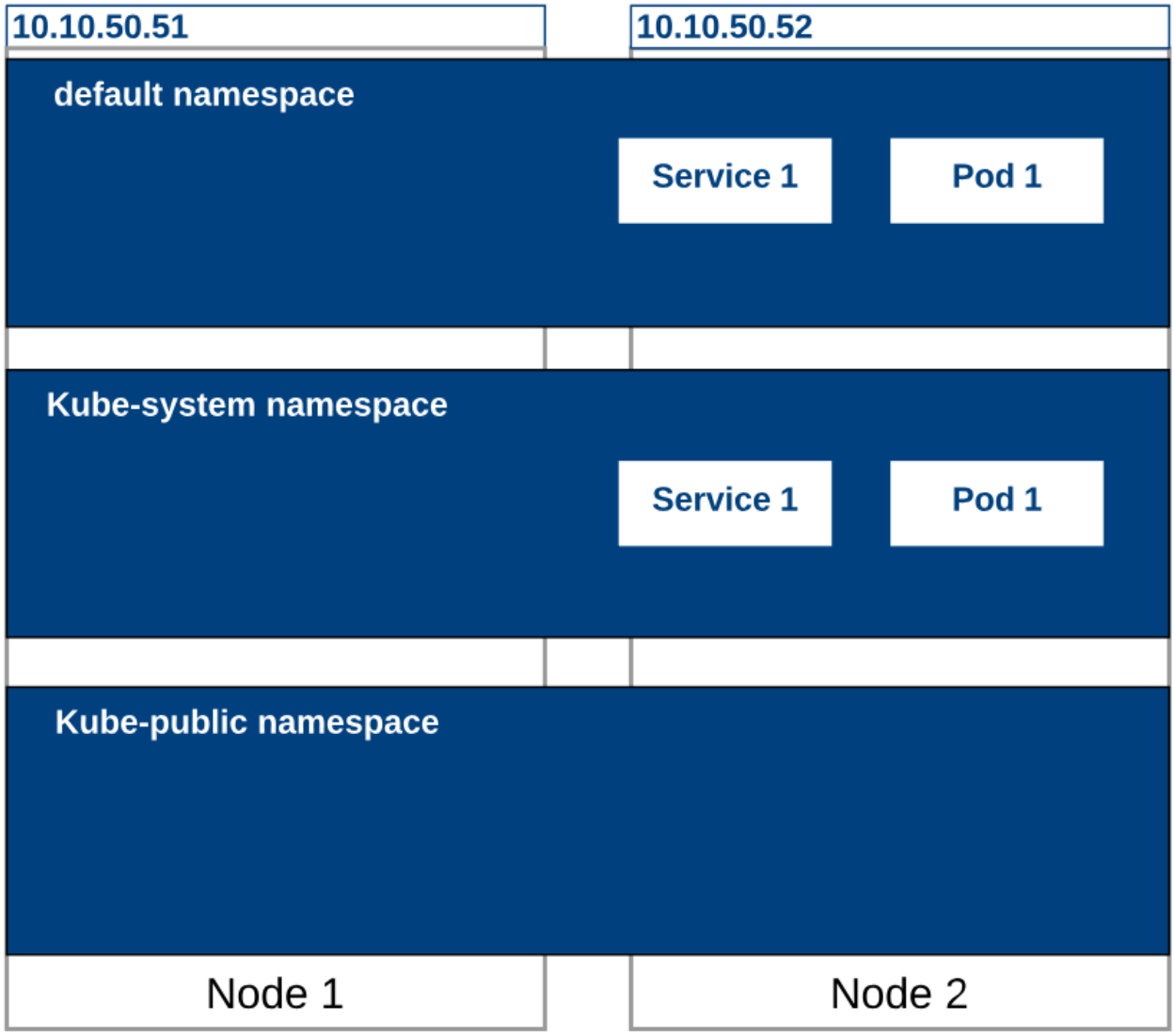
In this post we'll start exploring ways that you might be able to better manage your Kubernetes cluster for security or organizational purposes. Namespaces become a big piece of how your Kubernetes cluster operates and who sees what inside your cluster.

## Namespaces – The Theory

The easiest way to think of a namespace is that its a logical separation of your Kubernetes Cluster. Just like you might have segmented a physical server into several virtual severs, we can segment our Kubernetes cluster into namespaces. Namespaces are used to isolate resources within the control plane. For example if we were to deploy a pod in two different namespaces, an administrator running the "get pods" command may only see the pods in one of the namespaces. The pods could communicate with each other across namespaces however.

The Kubernetes cluster we've been working with should have a few namespaces built in to our default deployment. These include:

- Default – This is the namespace where all our pods and services run unless we specify a different one. This is the namespace our work has been completed in up until this point.
- kube-public – The public namespace is available to everyone with access to the Kubernetes cluster
- Kube-System – The system namespace is being used by your cluster right now. It stores objects related to the management of the Kubernetes cluster. It's probably smart to leave the kube-system namespace alone. Here be dragons.
-




So why would we create additional namespaces? Namespaces can be used for security purposes so that you can couple it with role based access control (RBAC) for your users. Instead of building multiple Kubernetes clusters which might waste resources, we can build a single cluster and then carve it up into namespaces if we need to give different teams their own space to work.

Your cluster might also be carved up into namespaces by environment, such as Production and Development. Service Names can be re-used if they are placed in different namespaces so your code can be identical between the namespaces and not conflict with each other.

It's also possible that you just want to segment some of your containers so that not everyone sees them. Maybe you've got a shared service that would be used between teams and don't want it to show up in each team's work space. Namespaces can be a great tool for Kubernetes hygiene.

## Namespaces – In Action

To get our hands dirty, lets start by listing the namespaces that currently exist in our Kubernetes cluster. We can do this by running:

```
kubectl get namespaces
```

Code language: JavaScript (javascript)

```
Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl get namespaces
NAME          STATUS    AGE
default       Active    17d
kube-public   Active    17d
kube-system   Active    17d
```

There are the three out of the box namespaces we discussed in our theory section. Now let's create a new namespace with our desired state manifest files as we have in previous posts. We'll also deploy a naked pod within this container to show that we can do it, and to later show how namespaces segment our resources.

```
kind: Namespace
apiVersion: v1
metadata:
  name: hollow-namespace
  labels:
    name: hollow-namespace
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: hollow-namespace
  labels:
    name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
```

Code language: PHP (php)

Deploy the manifest file above with the:

```
kubectl apply -f [manifest file].yml
```

Code language: CSS (css)

Once the namespace and pod have been created, lets run a quick get pod command as well, just to see what's happening.

```
Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl apply -f namespace.yml
namespace "hollow-namespace" created
pod "nginx" created
Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl get pods
No resources found.
```

The screenshot shows that we created a new namespace and we created a new pod, but when we did our get pods command, nothing was listed. Thats because our context is still set to

the default namespace, as it is by default. To see our pods in other namespaces we have a couple of options.

First we can list all pods across all namespaces if our permissions allow. We can do this by running:

```
kubectl get pod --all-namespaces
```

Code language: JavaScript (javascript)

```
Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl get pod --all-namespaces
NAMESPACE           NAME                                    READY     STATUS
hollow-namespace    nginx                                   1/1       Running
kube-system         coredns-86c58d9df4-2rrpn                1/1       Running
kube-system         coredns-86c58d9df4-8w5mz                1/1       Running
kube-system         etcd-kube-master                        1/1       Running
kube-system         kube-apiserver-kube-master              1/1       Running
kube-system         kube-controller-manager-kube-master     1/1       Running
kube-system         kube-flannel-ds-amd64-b88br             1/1       Running
kube-system         kube-flannel-ds-amd64-m5mbp             1/1       Running
kube-system         kube-flannel-ds-amd64-spt98             1/1       Running
kube-system         kube-proxy-8pw9c                        1/1       Running
kube-system         kube-proxy-9hvjc                        1/1       Running
kube-system         kube-proxy-km6lc                        1/1       Running
kube-system         kube-scheduler-kube-master              1/1       Running
```

Wow, there are a lot of other pods running besides the ones we've deployed! Most of these pods are running in the kube-system namespace though and we'll leave them alone.

Running our commands across all namespaces can be too much to deal with, what if we just want to see all pods in a single namespace instead? For this we can run:

```
kubectl get pod --namespace=[namespace name]
```

Code language: JavaScript (javascript)

```
Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl get pod --namespace=hollow-namespace
NAME      READY     STATUS    RESTARTS   AGE
nginx     1/1       Running   0          14m
```

**Summary**

Now we've seen how we can logically segment our cluster into different areas for our teams to work. In future posts we'll discuss namespaces more including how to set the current context so that we are using the correct namespace when we log in to our cluster.

To delete the pod and namespace used in this post run the following command:

```
kubectl delete -f [manifest file].yml
```

# Kubernetes – KUBECONFIG and Context

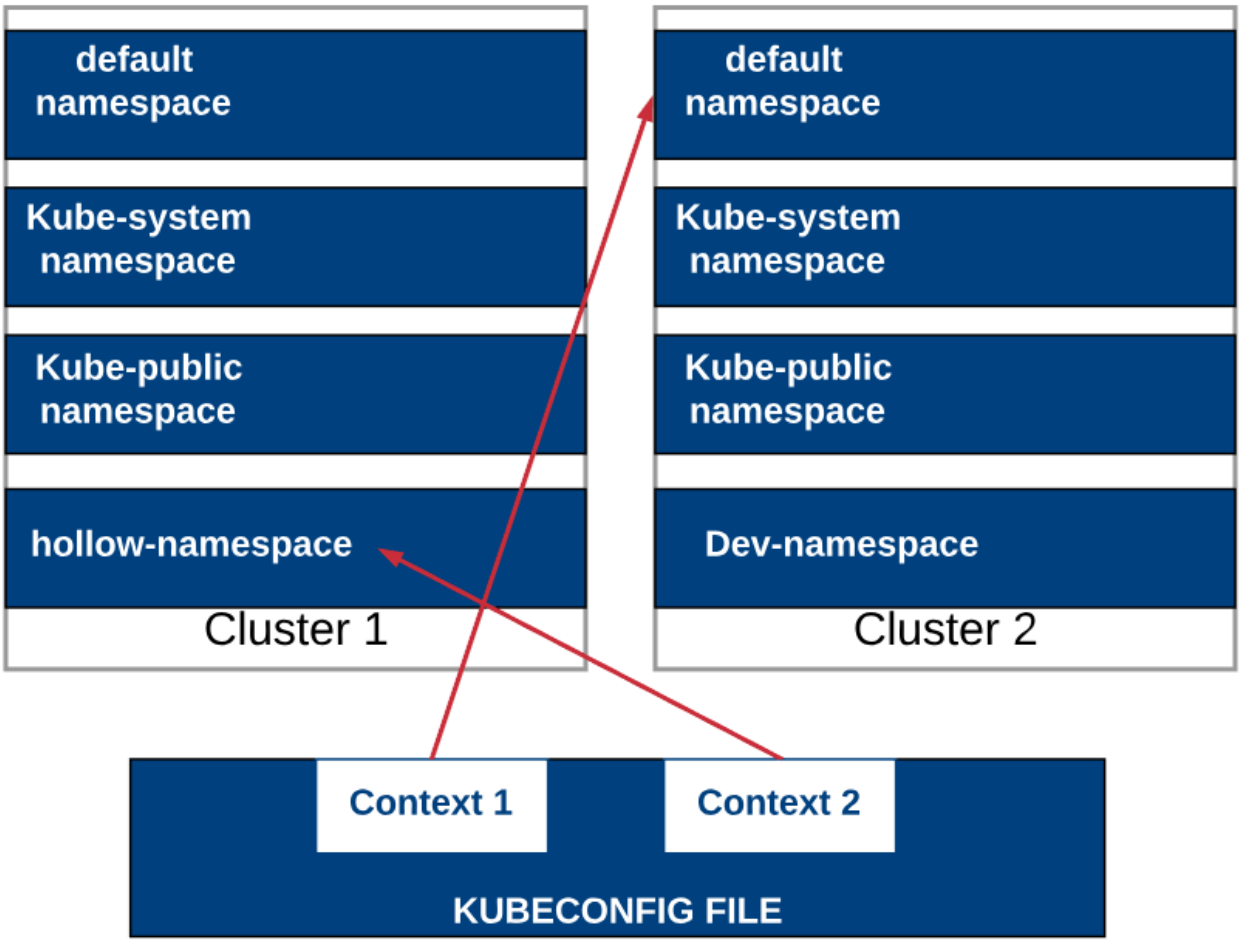
You've been working with Kubernetes for a while now and no doubt you have lots of clusters and namespaces to deal with now. This might be a good time to introduce Kubernetes KUBECONFIG files and context so you can more easily use all of these different resources.

# KUBECONFIG and Context – The Theory

When you first setup your Kubernetes cluster you created a config file likely stored in your $HOME/.kube directory. This is the KUBECONFIG file and it is used to store information about your connection to the Kubernetes cluster. When you use kubectl to execute commands, it gets the correct communication information from this KUBECONFIG file. This is why you would've needed to add this file to your $PATH variable so that it could be used correctly by the kubectl commands.

The KUBECONFIG file contains several things of interest including the cluster information so that kubectl is executing commands on the correct cluster. It also stores authentication information such as username/passwords, certificates or tokens. Lastly, the KUBECONFIG file stores contexts.

Contexts group access information under an easily recognizable name. So a context would include a cluster, a user and a namespace. Remember in the previous post where we talked about namespaces and how we could logically separate our Kubernetes cluster? Now we can use KUBECONFIG and context to set our default namespaces. No more logging in and being dumped into the default namespace.




# KUBECONFIG and Context – In Action

Lets start by deploying a new namespace and an example pod so that we have something to work with.

```
kind: Namespace
apiVersion: v1
metadata:
  name: hollow-namespace
  labels:
    name: hollow-namespace
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: hollow-namespace
  labels:
    name: nginx
```

```
spec:
  containers:
  - name: nginx
    image: nginx
```

Code language: PHP (php)

```
kubectl apply -f {manifest file].yml
```

Now we've got a new namespace named "hollow-namespace" that we can use to test out setting up our context in our KUBECONFIG file. Before we do that, lets take a look at our current context by running:

```
kubectl config current-context
```

```
Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl config current-context
kubernetes-admin@kubernetes
```

Based on my setup my context is named kubernetes-admin@kubernetes. Lets take a closer look at the configuration by running:

```
kubectl config view
```

```
Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: REDACTED
    server: https://10.10.50.50:6443
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
users:
- name: kubernetes-admin
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

The output from the previous command shows us a good deal of detail about our current configuration. In fact, you can probably find this same information if you open your KUBECONFIG file in a text editor. **NOTE:** the KUBECONFIG file is probably not named "kubeconfig". Mine was named admin.conf but the name isn't really important.

We can see that in our config file, we have a "Contexts" section. This section sets our default cluster, namespace and user that kubectl will use with its commands.

Now let's update our KUBECONFIG with our new namespace that we created. We can do this via kubectl by running the following command:

```
kubectl config set-context theithollow --namespace=hollow-namespace --cluster=kubernetes --user=kubernetes-admin
```

Code language: JavaScript (javascript)

You can create your own context name and change the namespace as you see fit. Since this is for theITHollow, I've used that as the context.

Once we've set a new context, we can re-run the "config view" command to see if our context changed at all. You should see the context section has been updated.

```
Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: REDACTED
    server: https://10.10.50.50:6443
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes
- context:
    cluster: kubernetes
    namespace: hollow-namespace
    user: kubernetes-admin
  name: theithollow
current-context: theithollow
kind: Config
preferences: {}
users:
- name: kubernetes-admin
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

So if we run our get pods right now, we shouldn't see any pods because we're still in the default namespace. Let's change our context so that we're using the new namespace. In my case the hollow-namespace. Then we'll view our current-context again after changing the context in use.

```
kubectl config use-context theithollow
kubectl config current-context
```

Code language: PHP (php)

Lets run a get pods command and if we did everything right, we should see our pod from the "hollow-namespace" instead of the pods from the default namespace.

```
Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl config set-context theithollow --namespace=hol
low-namespace --cluster=kubernetes --user=kubernetes-admin
Context "theithollow" modified.
Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
nginx     1/1       Running   0          2h
```

**Summary**

After this post you should feel more comfortable not only with namespaces and how they can be set to default, but how to configure connections to multiple Kubernetes clusters. If you'd like to reset your cluster we can run the following commands to delete our namespace and pods, as well as removing the context information that was added and setting our context back to normal.

```
kubectl config use-context kubernetes-admin@kubernetes
kubectl config unset contexts.theithollow #replace theithollow with your context name
kubectl delete -f [manifest name].yml #Manifest is the file used to deploy the namespace and the naked pod.
```

# Kubernetes – Ingress

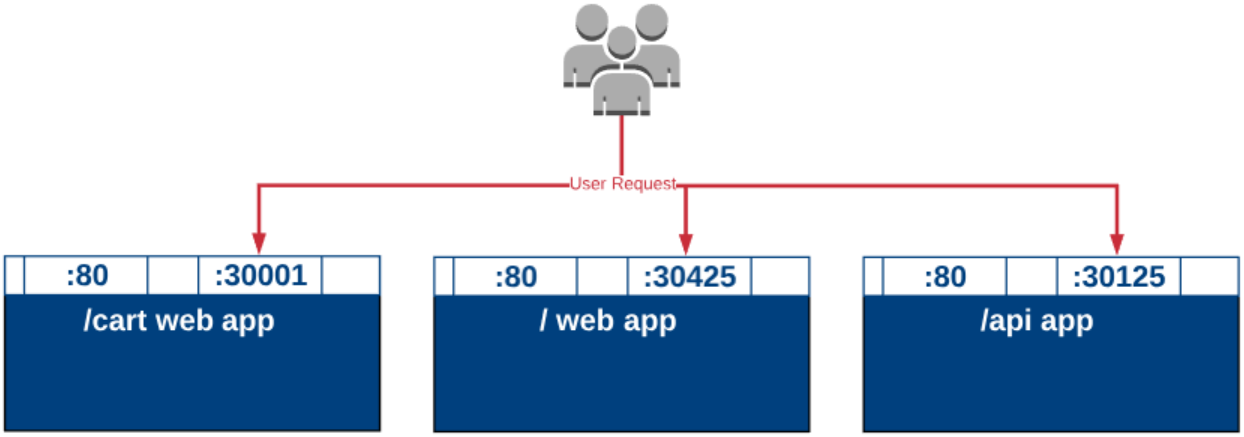
*By* ERIC SHANKS

It's time to look closer at how we access our containers from outside the Kubernetes cluster. We've talked about Services with NodePorts, LoadBalancers, etc., but a better way to handle ingress might be to use an ingress-controller to proxy our requests to the right backend service. This post will take us through how to integrate an ingress-controller into our Kubernetes cluster.

## Ingress Controllers – The Theory

Lets first talk about why we'd want to use an ingress controller in the first place. Take an example web application like you might have for a retail store. That web application might have an index page at "http://store-name.com/" and a shopping cart page at "http://store-name.com/cart" and an api URI at "http://store-name.com/api". We could build all these in a single container, but perhaps each of those becomes their own set of pods so that they can all scale out independently. If the API needs more resources, we can just increase the number of pods and nodes for the api service and leave the / and the /cart services alone. It also allows for multiple groups to work on different parts simultaneously but we're starting to drift off the point which hopefully you get now.

OK, so assuming we have these different services, if we're in an on-prem Kubernetes service we'd need to expose each of those services to the external networks. We've done this in the past with a node port. The problem is, we can't expose each app individually and have it work well because each service would need a different port. Imagine your users having to know which ports to use for each part of your application. It would be unusable like the example below.




A much simpler solution would be to have a single point of ingress (you see where I'm going) and have this "ingress-controller" define where the traffic should be routed.

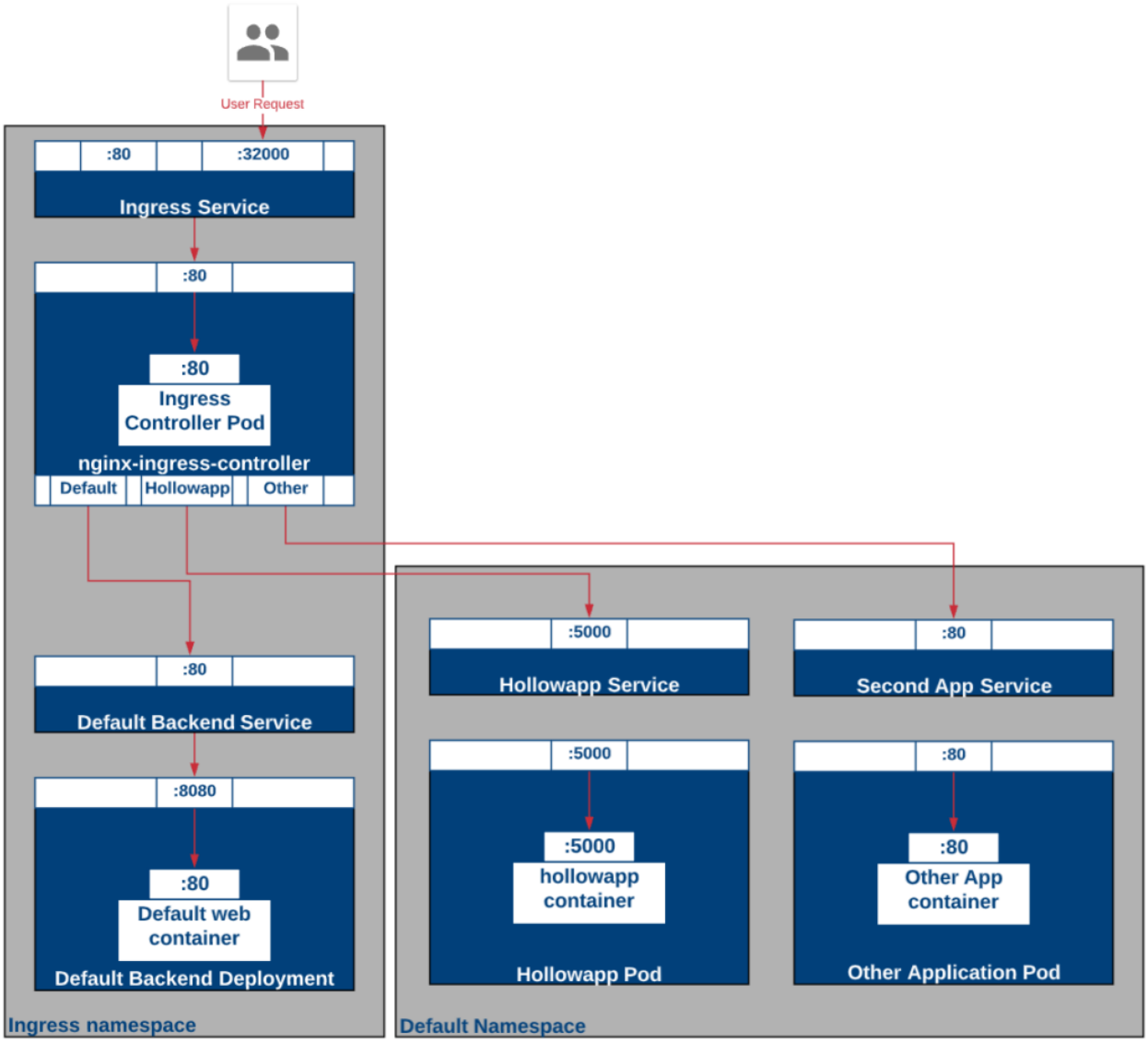
Ingress with a Kubernetes cluster comes in two parts.

1. Ingress Controller
2. Ingress Resource

The ingress-controller is responsible for doing the routing to the right places and can be thought of like a load balancer. It can route requests to the right place based on a set of rules applied to it. These rules are called an "ingress resource." If you see ingress as part of a Kubernetes manifest, it's likely not an ingress controller, but a rule that should be applied on the controller to route new requests. So the ingress controller likely is running in the cluster all the time and when you have new services, you just apply the rule so that the controller knows where to proxy requests for that service.

# Ingress Controllers – In Action

For this scenario we're going to deploy the objects depicted in this diagram. We'll have an ingress controller, a default backend, and two apps have different host names. Also notice, that we'll be using an NGINX ingress controller and that it is setup in a different namespace which helps keep the cluster secure and clean for anyone that shouldn't need to see that controller deployment.




First, we'll set the state by deploying our namespace where the ingress controller will live. To do that lets first start with a manifest file for our namespace.

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: ingress
```

Code language: PHP (php)

Once you deploy your namespace, we can move on to our configmap that has information about our environment, used by our ingress controller when it starts up.

```
---
apiVersion: v1
kind: ConfigMap
metadata:
```

```
  name: nginx-ingress-controller-conf
  labels:
    app: nginx-ingress-lb
  namespace: ingress
data:
  enable-vts-status: 'true'
```

Code language: JavaScript (javascript)

Lastly, before we get to our ingress objects, we need to deploy a service account with permissions to deploy and read information from the Kubernetes cluster. Here is a manifest that can be used.

**NOTE:** If you're following this series, you may not know what a service account is yet. For now, think of the service account as an object with permissions attached to it. It's a way for the Ingress controller to get permissions to interact with the Kubernetes API.

```
---
apiVersion: v1
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: nginx
  namespace: ingress
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: nginx-role
  namespace: ingress
rules:
  - apiGroups:
      - ""
    resources:
      - configmaps
      - endpoints
      - nodes
      - pods
      - secrets
    verbs:
      - list
      - watch
  - apiGroups:
      - ""
    resources:
      - nodes
```

```
    verbs:
      - get
  - apiGroups:
      - ""
    resources:
      - services
    verbs:
      - get
      - list
      - update
      - watch
  - apiGroups:
      - extensions
    resources:
      - ingresses
    verbs:
      - get
      - list
      - watch
  - apiGroups:
      - ""
    resources:
      - events
    verbs:
      - create
      - patch
  - apiGroups:
      - extensions
    resources:
      - ingresses/status
    verbs:
      - update
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: nginx-role
  namespace: ingress
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: nginx-role
subjects:
  - kind: ServiceAccount
    name: nginx
    namespace: ingress
```

Code language: JavaScript (javascript)

Now that those prerequisites are deployed, we'll deploy a default backend for our controller. This is just a pod that will handle any requests where there is an unknown route. If someone accessed our cluster at http://clusternodeandport/**bananas** we wouldn't have a route that handled that so we'd point it to the default backend with a 404 error in it. NGINX has a sample backend that you can use and that code is listed below. Just deploy the manifest to the cluster.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: default-backend
  namespace: ingress
spec:
  replicas: 1
  selector:
    matchLabels:
      app: default-backend
  template:
    metadata:
      labels:
        app: default-backend
    spec:
      terminationGracePeriodSeconds: 60
      containers:
      - name: default-backend
        image: gcr.io/google_containers/defaultbackend:1.0
        livenessProbe:
          httpGet:
            path: /healthz
            port: 8080
            scheme: HTTP
          initialDelaySeconds: 30
          timeoutSeconds: 5
        ports:
        - containerPort: 8080
        resources:
          limits:
            cpu: 10m
            memory: 20Mi
          requests:
            cpu: 10m
            memory: 20Mi
---
apiVersion: v1
kind: Service
metadata:
  name: default-backend
```

```
  namespace: ingress
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    app: default-backend
```

```
kubectl apply -f [manifest file].yml
```

Code language: CSS (css)

Now that the backend service and pods are configured, it's time to deploy the ingress controller through another deployment file. Here is a deployment of the Ingress Controller for NGINX.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-ingress-controller
  namespace: ingress
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-ingress-lb
  revisionHistoryLimit: 3
  template:
    metadata:
      labels:
        app: nginx-ingress-lb
    spec:
      terminationGracePeriodSeconds: 60
      serviceAccount: nginx
      containers:
        - name: nginx-ingress-controller
          image:     quay.io/kubernetes-ingress-controller/nginx-
ingress-controller:0.9.0
          imagePullPolicy: Always
          readinessProbe:
            httpGet:
              path: /healthz
              port: 10254
              scheme: HTTP
          livenessProbe:
            httpGet:
              path: /healthz
              port: 10254
              scheme: HTTP
```

```
            initialDelaySeconds: 10
            timeoutSeconds: 5
          args:
            - /nginx-ingress-controller
            - --default-backend-service=$(POD_NAMESPACE)/default-
backend
            -         --configmap=\$(POD_NAMESPACE)/nginx-ingress-
controller-conf
            - --v=2
          env:
            - name: POD_NAME
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: POD_NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
          ports:
            - containerPort: 80
            - containerPort: 18080
```

To deploy your manifest, it's again the:

```
kubectl apply -f [manifest file].yml
```

Code language: CSS (css)

Congratulations, your controller is deployed, lets now deploy a Service that exposes the ingress controller to the outside world and in our case through a NodePort.

```
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: ingress
spec:
  type: NodePort
  ports:
    - port: 80
      name: http
      nodePort: 32000
    - port: 18080
      name: http-mgmt
  selector:
    app: nginx-ingress-lb
```

Code language: PHP (php)

Deploy the Service with:

```
kubectl apply -f [manifest file].yml
```

Code language: CSS (css)

The service just deployed publishes itself on port 32000. This was hard coded into the yaml manifest that was just deployed. If you'd like to check the service to make sure, run:

```
kubectl get svc --namespace=ingress
```

Code language: JavaScript (javascript)

```
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                       AGE
default-backend   ClusterIP   10.98.233.173   <none>        80/TCP                        125m
nginx             NodePort    10.100.8.99     <none>        80:32000/TCP,18080:31071/TCP  118m
```

As you can see from my screenshot, the outside port for http traffic is 32000. This is because I've hard coded the 32000 port as a NodePort into the ingress-svc manifest. Be careful doing this as this is the only service that can use this port. You can remove the nodeport but you will need to lookup the port assigned to this service if you do. Hard coding this port into the manifest was used to simplify these instructions and make things more clear as you follow along.

Now we'll deploy our application called hollowapp. The manifest file below has standard Deployments and a Service (that isn't exposed externally). It also has a new resource of kind "ingress" which is our ingress rule to be applied on our controller. The main thing is anyone who access the ingress-controller with a host name of hollowapp.hollow.local will route traffic to our service. This means we need to setup a DNS record to point at our Kubernetes cluster for this resource. I've done this in my lab and you can change this to meet your own needs in your lab with your own app.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
      app: hollowapp
  name: hollowapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hollowapp
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: hollowapp
    spec:
      containers:
      - name: hollowapp
        image: theithollow/hollowapp-blog:allin1-v2
        imagePullPolicy: Always
        ports:
```

```
        - containerPort: 5000
        env:
        - name: SECRET_KEY
          value: "my-secret-key"
---
apiVersion: v1
kind: Service
metadata:
  name: hollowapp
  labels:
    app: hollowapp
spec:
  type: ClusterIP
  ports:
  - port: 5000
    protocol: TCP
    targetPort: 5000
  selector:
    app: hollowapp
---
apiVersion: networking.k8s.io/v1beta1
kind: Ingress #ingress resource
metadata:
  name: hollowapp
  labels:
    app: hollowapp
spec:
  rules:
  -   host:   hollowapp.hollow.local   #only   match   connections   to
hollowapp.hollow.local.
    http:
      paths:
      - path: / #root path
        backend:
          serviceName: hollowapp
          servicePort: 5000
```

Code language: PHP (php)

Apply the manifest above, or a modified version for your own environment with the command below:

```
kubectl apply -f [manifest file].yml
```

Code language: CSS (css)

Now, we can test out our deployment in a web browser. First lets make sure that something is being returned when we hit the Kubernetes ingress controller via the web browser and I'll use the IP Address in the URL.

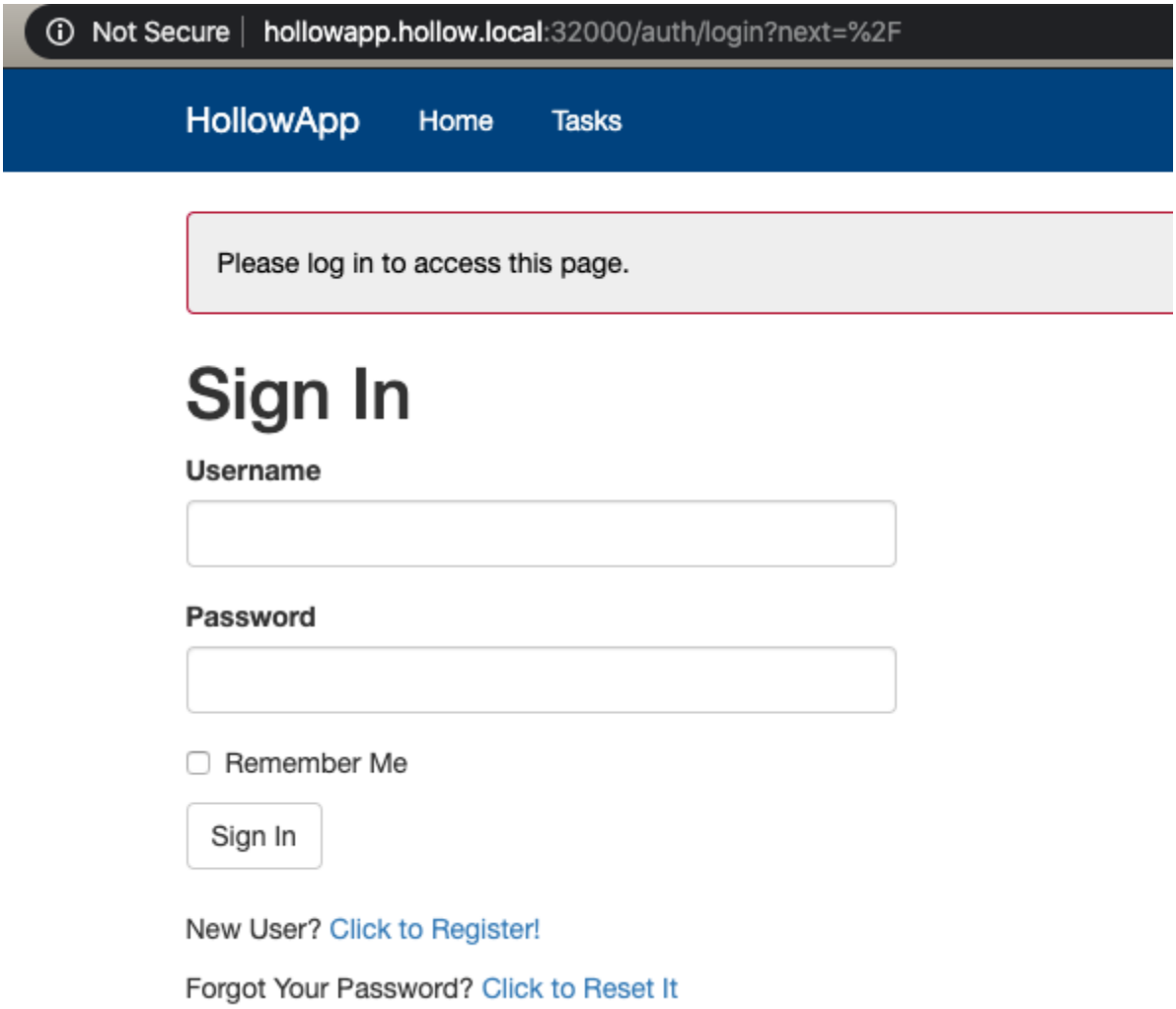

```
default backend - 404
```

So that is the default backend providing a 404 error, as it should. We don't have an ingress rule for access the controller via an IP Address so it used the default backend. Now try that again by using the hostname that we used in the manifest file which in my case was http://hollowapp.hollow.local.

And before we do that here's a screenshot showing that the host name maps to the EXACT same IP address we used to test the 404 error.

```
PING hollowapp.hollow.local (10.10.50.176): 56 data bytes
64 bytes from 10.10.50.176: icmp_seq=0 ttl=63 time=0.767 ms
64 bytes from 10.10.50.176: icmp_seq=1 ttl=63 time=0.846 ms
```

When we access the cluster by the hostname, we get our new application we've deployed.




**Summary**

The examples shown in this post can be augmented by adding a load balancer outside the cluster but you should get a good idea of what an ingress controller can do for you. Once you've got it up and running it can provide a single resource to access from outside the cluster and many service running behind it. Other advanced controllers exist as well, so this post should just serve as an example. Ingress controllers can do all sorts of things including handling TLS, monitoring, handling session persistence and others. Feel free to checkout all your ingress options from all sorts of sources including NGINX, Heptio, Traefik and others.

# Kubernetes – DNS

*By* ERIC SHANKS

DNS is a critical service in any system. Kubernetes is no different, but Kubernetes will implement its own domain naming system that's implemented within your Kubernetes cluster. This post explores the details that you need to know to operate a k8s cluster properly.

# Kubernetes DNS – The theory

I don't want to dive into DNS too much since it's a core service most should be familiar with. But at a really high level, DNS translates an IP address that might be changing, with an easily remember-able name such as "theithollow.com". Every network has a DNS server, but Kubernetes implements their own DNS within the cluster to make connecting to containers a simple task.

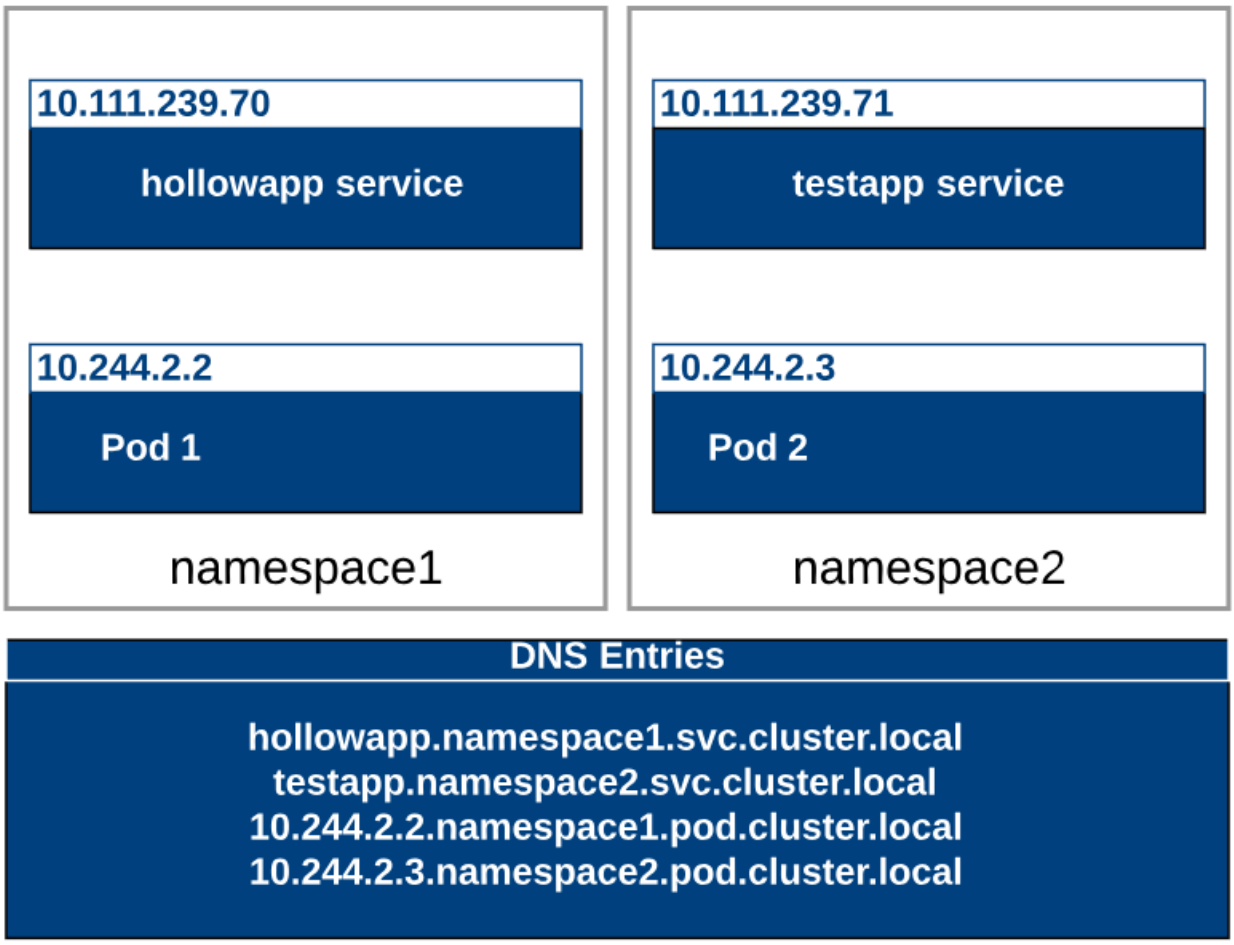
There are two implementations of DNS found within Kubernetes clusters. Kube-dns and CoreDNS. The default used with kubeadm after version 1.13 is to use CoreDNS which is managed by the Cloud Native Computing Foundation. Since I used kubeadm to setup our cluster and it's the default version, this post will focus on CoreDNS.

This topic can be pretty detailed, but lets distill it into the basics. Each time you deploy a new service or pod, the DNS service sees the calls made to the Kube API and adds DNS entries for the new object. Then other containers within a Kubernetes cluster can use these DNS entries to access the services within it by name.

There are two objects that you could access via DNS within your Kubernetes cluster and those are Services and Pods. Now Pods can get a dns entry, but you'd be wondering why, since we've learned that accessing pods should be done through a Service. For this reason, pods don't get a DNS entry by default with CoreDNS.

Services get a DNS entry of the service name appended to the namespace and then appended to "svc.cluster.local". Similarly pods get entries of PodIPAddress appended to the namespace and then appended to .pod.cluster.local if Pod DNS is enabled in your cluster.

The diagram below shows a pair of services and a pair of pods that are deployed across two different namespaces. Below this are the DNS entries that you can expect to be available for these objects, again assuming Pod DNS is enabled for your cluster.




Any pods looking for a service within the same namespace can just use the common name of the "Service" instead of the FQDN. Kubernetes will add the proper DNS suffix to the request if one is not given. In fact, when you deploy new pods, Kubernetes specifies your DNS server for you unless you override it within a manifest file.

## Kubernetes DNS – In Action

Lets take a quick look at how DNS is working in our own Kubernetes cluster. Feel free to deploy any Kubernetes manifest that has a service and pod in the deployment. If you need one, you can use mine below.

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

```
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx-container
        image: nginx
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginxsvc
spec:
  type: NodePort
  ports:
  - name: http
    port: 80
    targetPort: 80
    nodePort: 30001
    protocol: TCP
  selector:
    app: nginx
```

You can deploy the file with:

```
kubectl apply -f [manifest file].yml
```

Code language: CSS (css)

Now lets look at some information about our service which we'll check later in this post.

```
kubectl get svc
```

Code language: JavaScript (javascript)

```
Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl get svc
NAME                  TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
default-http-backend  ClusterIP   10.110.176.215   <none>        80/TCP         4d
kubernetes            ClusterIP   10.96.0.1        <none>        443/TCP        22d
nginxsvc              NodePort    10.103.134.28    <none>        80:30001/TCP   2h
```

Once our example pods and services are deployed, let's deploy another container where we can run some DNS lookups. To do this we'll deploy the shell-demo container and get an interactive shell by using the commands below. We'll also install busybox once we've got a shell.

```
Kubectl create -f https://k8s.io/examples/application/shell-demo.yaml
```

```
kubectl exec -it shell-demo -- /bin/bash
```

```
Apt-get update
Apt-get install busybox -y
```
Code language: JavaScript (javascript)

Now that busybox is installed and we're in an interactive shell session, lets check to see what happens when we run a dns lookup on our example service.

```
busybox nslookup nginxsvc
```

The results of our nslookup on the nginxsvc that we deployed early on in this post shows the ClusterIP address of the Service. Also, pay close attention to the FQDN. The FQDN is the [service name] +"." + [namespace name] + "." + svc.cluster.local.

```
root@shell-demo:/# busybox nslookup nginxsvc
Server:    10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:      nginxsvc
Address 1: 10.103.134.28 nginxsvc.default.svc.cluster.local
```

**Summary**

DNS is a basic service that we rely on for almost all systems. Kubernetes is no different and it makes accessing other services very simple and straightforward. You can manipulate your DNS configs by update manifest files if you need, but for the basic operations, you should be good to go without even having to configure much. And remember… it's ALWAYS DNS.

To clean up your cluster run:

```
kubectl delete -f [manifest file].yml #manifest used during the deploy of the nginxsvc
kubectl delete pod shell-demo
```

# Kubernetes – ConfigMaps

*By* ERIC SHANKS

Sometimes you need to add additional configurations to your running containers. Kubernetes has an object to help with this and this post will cover those ConfigMaps.

## ConfigMaps – The Theory

Not all of our applications can be as simple as the basic nginx containers we've deployed earlier in this series. In some cases, we need to pass configuration files, variables, or other information to our apps.

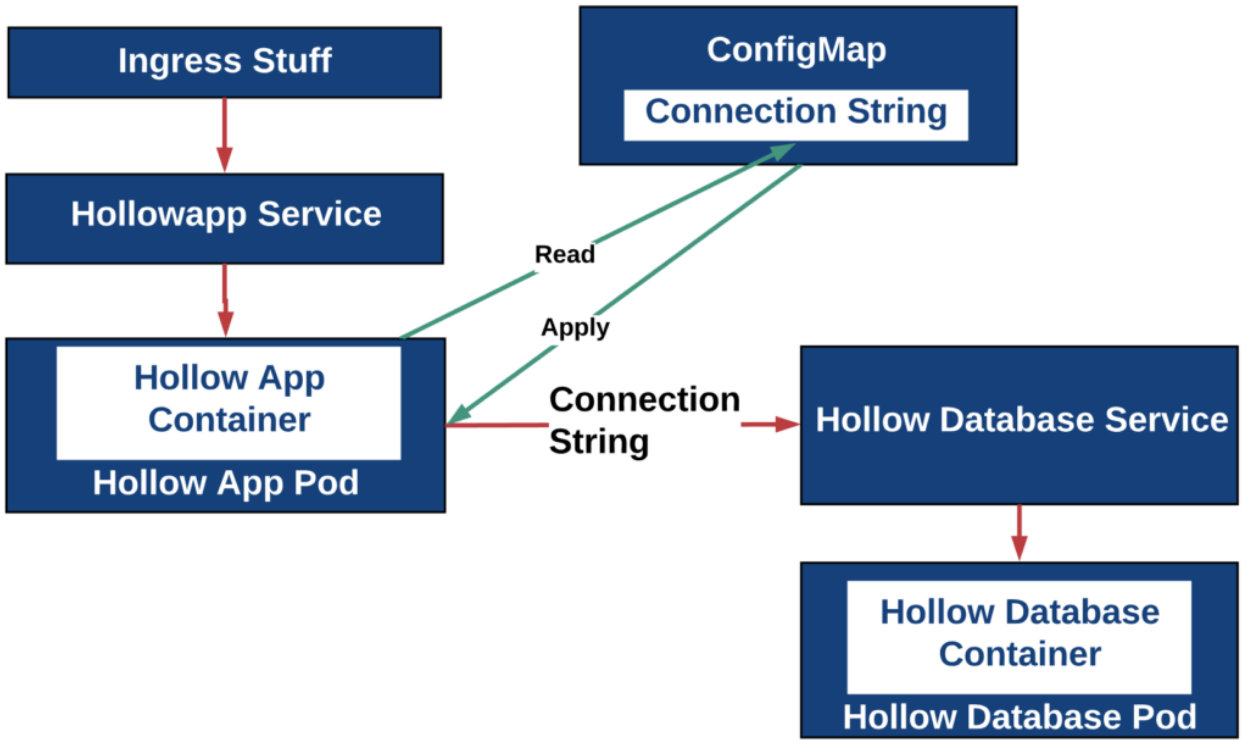
The theory for this post is pretty simple, ConfigMaps store key/value pair information in an object that can be retrieved by your containers. This configuration data can make your applications more portable.

For example, you could have a key value pair of "environment:dev" in a configmap for your development Kubernetes cluster. When you deploy your apps, you can key some of your logic off of the "environment" variable and do different things for production vs. development environments. I'm not sure this is necessary, but it's just an example to get you thinking about it. Let's see these ConfigMaps in action and I think you'll get the picture.

# ConfigMaps – In Action

For a ConfigMap example, we'll take a simple two tier app (lovingly called hollowapp) and we'll configure the database connection string through a ConfigMap object. Try to put out of your mind how insecure this is for the moment, it's just an example to prove the point.

Here is a high level diagram of the lab we'll be building. As you can see some of the items we've talked about in previous posts have been generalized. The main part is the App container using the ConfigMap to configure the connection string to the database service.




First, we'll deploy the database container and service. The database container has already been configured with new database with the appropriate username and password. The manifest file to deploy the DB and service is listed below.

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: hollowdb
  labels:
    app: hollowdb
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hollowdb
  template:
    metadata:
      labels:
        app: hollowdb
    spec:
      containers:
      - name: mysql
        image: theithollow/hollowapp-blog:dbv1
        imagePullPolicy: Always
        ports:
        - containerPort: 3306
---
apiVersion: v1
kind: Service
metadata:
  name: hollowdb
spec:
  ports:
  - name: mysql
    port: 3306
    targetPort: 3306
    protocol: TCP
  selector:
    app: hollowdb
```

To deploy the manifests, we run our familiar command:

```
kubectl apply -f [manifest file].yml
```

Code language: CSS (css)

OK, the Database is deployed and ready to go. Now we need to deploy a ConfigMap with the database connection string.

The ConfigMap is listed below in another manifest file. Most of the configuration in the manifest should look familiar to you. The kind has been updated to a type of "ConfigMap", but the important part is the data field. We have a key named. db.string and the value of that key is our connection string (yes, with the clear text super secret password).

```
apiVersion: v1
kind: ConfigMap
metadata:
```

```
  name: hollow-config
data:
  db.string:   "mysql+pymysql://hollowapp:Password123@hollowdb:3306/hollowapp"   #Key Value pair Value being a database connection string
```
Code language: PHP (php)

Deploy the ConfigMap using the same apply command as before and change the manifest file name.

```
kubectl apply -f [manifest file].yml
```
Code language: CSS (css)

Now you can run a get command to list the ConfigMap.

```
kubectl get configmap
```
Code language: JavaScript (javascript)

```
root@shell-demo:/# Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl get configmap
NAME            DATA      AGE
hollow-config   1         5h
```

Now, before we deploy our app, lets deploy a test container to prove that we can read that key value pair from the ConfigMap.

The manifest below has an environment variable named DATABASE_URL and we're telling it to get the value of that environment variable from the ConfigMap named hollow-config. Within the hollow-config ConfigMap, we're looking for the key named db.string. The result that will be the value stored in our DATABASE_URL variable.

```
apiVersion: v1
kind: Pod
metadata:
  name: shell-demo
spec:
  containers:
  - name: nginx
    image: nginx
    env:
    - name: DATABASE_URL
      valueFrom:
        configMapKeyRef:
          name: hollow-config
          key: db.string
```
Code language: CSS (css)

Deploy the test shell-demo container with the kubectl apply command.

```
kubectl apply -f [manifest file].yml
```
Code language: CSS (css)

Once it's deployed, you can get an interactive shell into that container by running the following command.

```
kubectl exec -it shell-demo -- /bin/bash
```
Code language: JavaScript (javascript)

Once we have a shell session, we can do an echo on our DATABASE_URL environment variable and it should show the string from our ConfigMap.

```
Erics-MacBook-Pro-2:k8s-series eshanks$ kubectl exec -it shell-demo -- /bin/bash
root@shell-demo:/# echo $DATABASE_URL
mysql+pymysql://hollowapp:Password123@hollowdb:3306/hollowapp
```

You can exit the shell session and then we're ready to deploy our app. The manifest for the app is shown below. NOTE: it does require that you have your ingress controller running if you plan to access it through a browser.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
      app: hollowapp
  name: hollowapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hollowapp
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: hollowapp
    spec:
      containers:
      - name: hollowapp
        image: eshanks16/k8s-hollowapp:v2
        imagePullPolicy: Always
        ports:
        - containerPort: 5000
        env:
        - name: SECRET_KEY
          value: "my-secret-key"
        - name: DATABASE_URL
          valueFrom:
            configMapKeyRef:
              name: hollow-config
              key: db.string
---
apiVersion: v1
kind: Service
metadata:
  name: hollowapp
  labels:
    app: hollowapp
spec:
  type: ClusterIP
  ports:
  - port: 5000
```

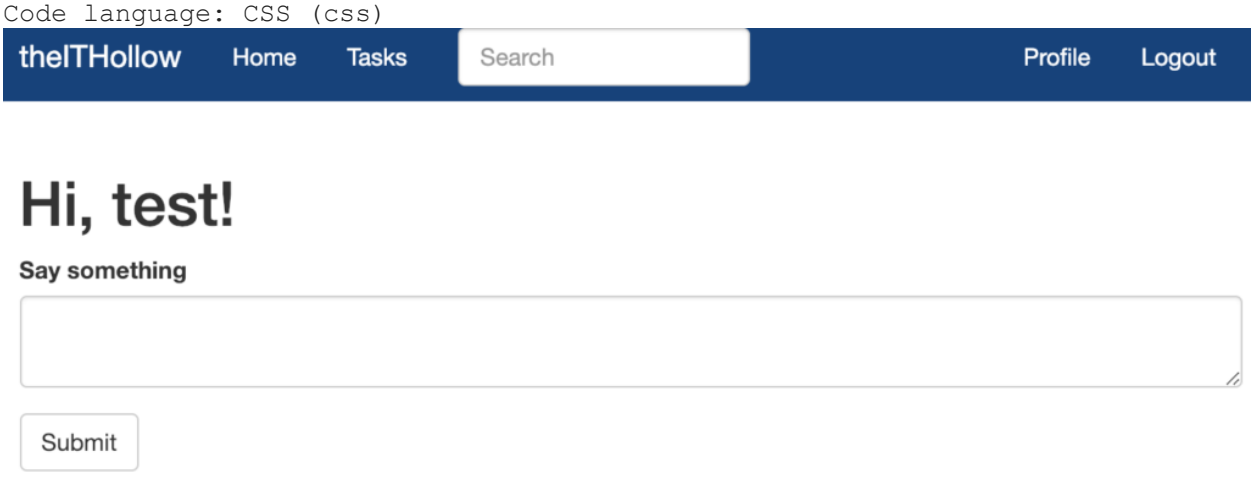
```
    protocol: TCP
    targetPort: 5000
  selector:
    app: hollowapp
---
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: hollowapp
  labels:
    app: hollowapp
spec:
  rules:
  - host: hollowapp.hollow.local
    http:
      paths:
      - path: /
        backend:
          serviceName: hollowapp
          servicePort: 5000
```

Code language: PHP (php)

Deploy the app with the same command.

```
kubectl apply -f [manifest file].yml
```

Code language: CSS (css)




The result is that our app is communicating correctly with our back end database container all because of our ConfigMap.

**Summary**

So this isn't the most secure way to pass connection information to your containers, but it is a pretty effective way of storing parameters that you might need for your applications. What will you think of to use ConfigMaps for?

# Kubernetes – Secrets

*By* ERIC SHANKS

Secret, Secret, I've got a secret! OK, enough of the Styx lyrics, this is serious business. In the previous post we used ConfigMaps to store a database connection string. That is probably not the best idea for something with a sensitive password in it. Luckily Kubernetes provides a way to store sensitive configuration items and its called a "secret".

## Secrets – The Theory

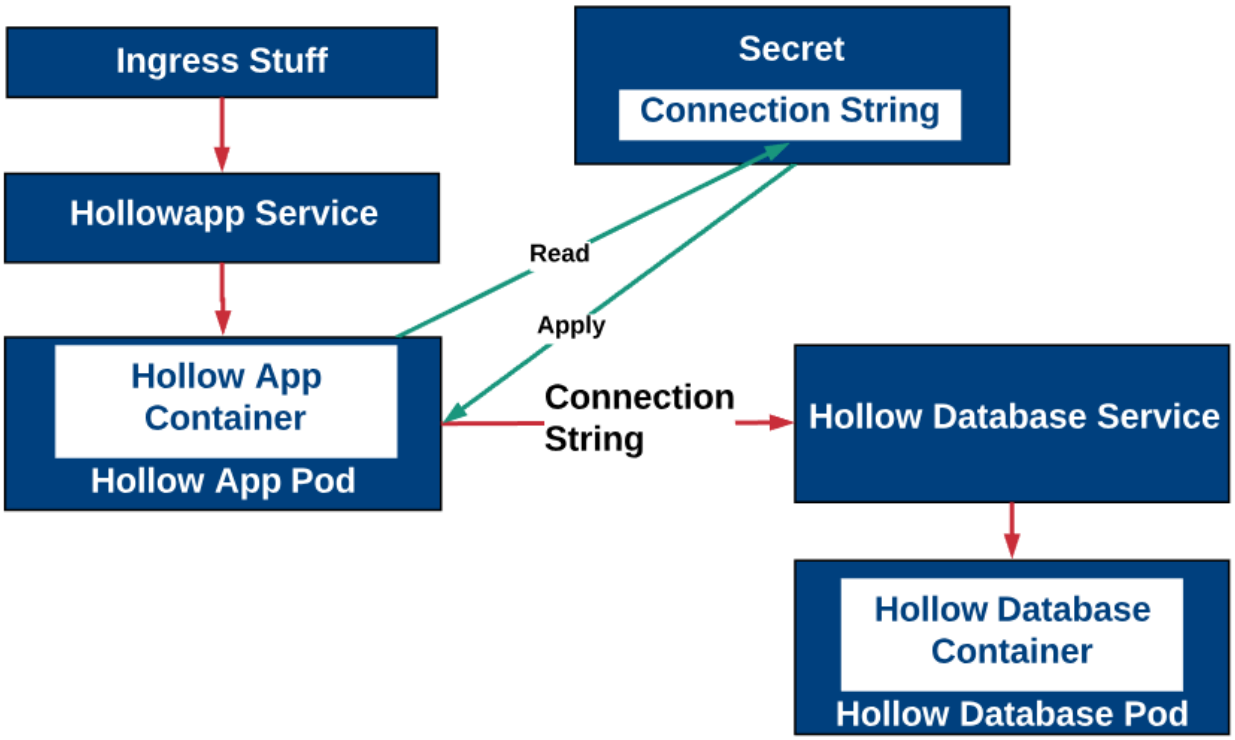
The short answer to understanding secrets would be to think of a ConfigMap, which we have discussed in a previous post in this series, but with non-clear text.

ConfigMaps would be good to store configuration data that is not sensitive. If it is sensitive information that not everyone should see, then a "secret" should be chosen over a ConfigMap. Password, keys, or private information should be stored as a secret instead of a ConfigMap.

One thing to note is that secrets can be stored a either "data" or "stringData" maps. Data would be used to store a secret in base64 format which you would provide. You can also use stringData which you'd provide an unencoded string but it would be stored as a base64 string when the secret is create. This is a valuable tool when your deployment creates a secret as part of the build process.

## Secrets – In Action

Since a secret is just a more secured version of a ConfigMap, the demo will be the same as the last post, with the exception that we'll use a secret over a ConfigMap to store our connection string. This is a better way to store a connection string over a ConfigMap because it does have a password in it which should be protected.




First, we'll deploy the database container and service. The database container has already been configured with new database with the appropriate username and password. The manifest file to deploy the DB and service is listed below.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hollowdb
  labels:
    app: hollowdb
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hollowdb
  template:
    metadata:
      labels:
        app: hollowdb
    spec:
      containers:
      - name: mysql
        image: theithollow/hollowapp-blog:dbv1
        imagePullPolicy: Always
        ports:
        - containerPort: 3306
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: hollowdb
spec:
  ports:
  - name: mysql
    port: 3306
    targetPort: 3306
    protocol: TCP
  selector:
    app: hollowdb
```

We'll deploy a new secret from a manifest after we take our connection string and convert it to base64. I took the following connection string:

```
mysql+pymysql://hollowapp:Password123@hollowdb:3306/hollowapp
```

Code language: JavaScript (javascript)

then ran it through a bash command of:

```
echo -n 'mysql+pymysql://hollowapp:Password123@hollowdb:3306/hollowapp'
| base64
```

Code language: PHP (php)

I took the result of that command and placed int in the Secret manifest under the db.string map.

```
apiVersion: v1
kind: Secret
metadata:
  name: hollow-secret
data:
  db.string:
bXlzcWwrcHlteXNxbDovL2hvbGxvd2FwcDpQYXNzd29yZDEyM0Bob2xsb3dkYjoz
MzA2L2hvbGxvd2FwcA==
```

Deploy the secret via a familiar command we've used for deploying our manifests.

```
kubectl apply -f [manifest file].yml
```

Code language: CSS (css)

Now that the secret is deployed, we can deploy our application pods which will connect to the backend database just as we did with a ConfigMap in a previous post. The important difference between the Deployment manifest using a ConfigMap and the Deployment manifest using a secret is this section.

```
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: hollow-secret
              key: db.string
```

Code language: CSS (css)

The full deployment file that will read from the secret and use that secret is listed below.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
      app: hollowapp
  name: hollowapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hollowapp
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: hollowapp
    spec:
      containers:
      - name: hollowapp
        image: theithollow/hollowapp-blog:allin1
        imagePullPolicy: Always
        ports:
        - containerPort: 5000
        env:
        - name: SECRET_KEY
          value: "my-secret-key"
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: hollow-secret
              key: db.string
---
apiVersion: v1
kind: Service
metadata:
  name: hollowapp
  labels:
    app: hollowapp
spec:
  type: ClusterIP
  ports:
  - port: 5000
    protocol: TCP
```

```
    targetPort: 5000
  selector:
    app: hollowapp
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hollowapp
  labels:
    app: hollowapp
spec:
  rules:
  - host: hollowapp.hollow.local
    http:
      paths:
      - path: /
        backend:
          serviceName: hollowapp
          servicePort: 5000
```

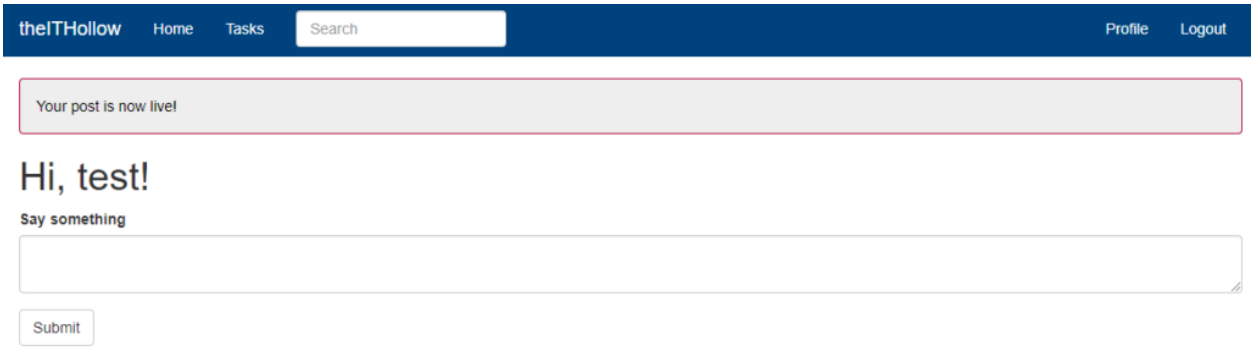
Code language: PHP (php)

Deploy the manifest using:

```
kubectl apply -f [manifest file].yml
```

Code language: CSS (css)

The result is that the app reads from the secret file, and uses that string as part of the connection to the backend database.




**Summary**

Sometimes you'll want to store non-sensitive data and a ConfigMap is the easy way to store that data. In other cases, the data shouldn't be available to everyone, like in the case of a password or a connection string. When those situations occur, a secret may be your method to store this data.

# Kubernetes – Persistent Volumes

Containers are often times short lived. They might scale based on need, and will redeploy when issues occur. This functionality is welcomed, but sometimes we have state to worry about and state is not meant to be short lived. Kubernetes persistent volumes can help to resolve this discrepancy.

## Volumes – The Theory

In the Kubernetes world, persistent storage is broken down into two kinds of objects. A Persistent Volume (PV) and a Persistent Volume Claim (PVC). First, lets tackle a Persistent Volume.

### Persistent Volumes

Persistent Volumes are simply a piece of storage in your cluster. Similar to how you have a disk resource in a server, a persistent volume provides storage resources for objects in the cluster. At the most simple terms you can think of a PV as a disk drive. It should be noted that this storage resource exists independently from any pods that may consume it. Meaning, that if the pod dies, the storage should remain intact assuming the claim policies are correct. Persistent Volumes are provisioned in two ways, Statically or Dynamically.

**Static Volumes –** A static PV simply means that some k8s administrator provisioned a persistent volume in the cluster and it's ready to be consumed by other resources.

**Dynamic Volumes –** In some circumstances a pod could require a persistent volume that doesn't exist. In those cases it is possible to have k8s provision the volume as needed if storage classes were configured to demonstrate where the dynamic PVs should be built. This post will focus on static volumes for now.

### Persistent Volume Claims

Pods that need access to persistent storage, obtain that access through the use of a Persistent Volume Claim. A PVC, binds a persistent volume to a pod that requested it.

When a pod wants access to a persistent disk, it will request access to the claim which will specify the size , access mode and/or storage classes that it will need from a Persistent Volume. Indirectly the pods get access to the PV, but only through the use of a PVC.

### Claim Policies

We also reference claim policies earlier. A Persistent Volume can have several different claim policies associated with it including:

**Retain –** When the claim is deleted, the volume remains.

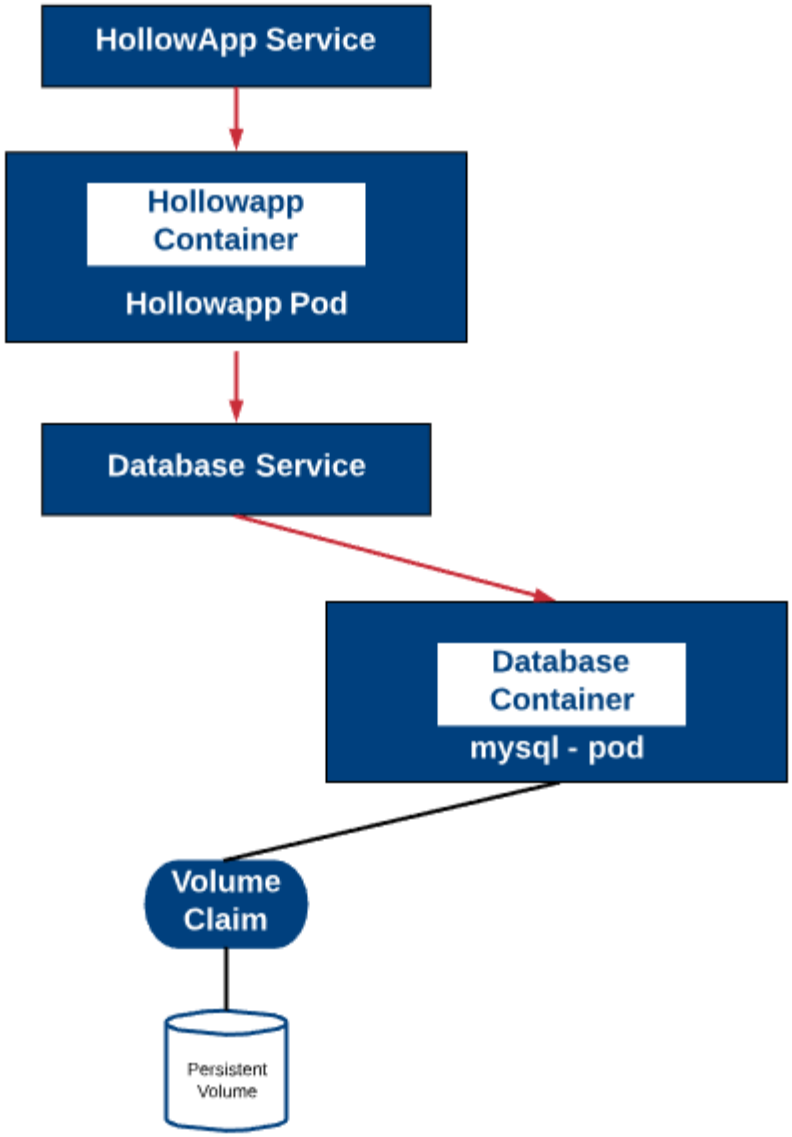
**Recycle –** When the claim is deleted the volume remains but in a state where the data can be manually recovered.

**Delete –** The persistent volume is deleted when the claim is deleted.

The claim policy (associated at the PV and not the PVC) is responsible for what happens to the data on when the claim has been deleted.

# Volumes – In Action

For the demonstration in the lab, we'll begin by deploying something that looks like the diagram below. The application service and pod won't change from what we've done before, but we need a front end to our application. However, the database pod will use a volume claim and a persistent volume to store the database for our application. Also, if you're following my example exactly, I'm using an ingress controller for the application, but however you present your application outside of the Kubernetes cluster is fine.




First, we'll start by deploying a persistent volume through a manifest file. Remember that you can deploy these manifest files by running:

```
kubectl apply -f [manifest file].yml
```

Code language: CSS (css)

Here is a sample manifest file for the persistent volume. This is a static persistent volume.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysqlvol
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi #Size of the volume
```

```
  accessModes:
    - ReadWriteOnce #type of access
  hostPath:
    path: "/mnt/data" #host location
```
Code language: PHP (php)

After you deploy your persistent volume, you can view it by running:

```
kubectl get pv
```
Code language: JavaScript (javascript)

Now that the volume has been deployed, we can deploy our claim.

**NOTE:** you can deploy the pv, pvc, pods, services, etc within the same manifest file, but for the purposes of this blog I'll often break them up so we can focus on one part over the other.

```
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysqlvol
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Once your claim has been created, we can look for those claims by running:

```
kubectl get pvc
```
Code language: JavaScript (javascript)

Great, the volume is setup and a claim ready to be used. Now we can deploy our database pod and service. The database pod will mount the volume via the claim and we're specifying in our pod code, that the volume will be mounted in the /var/lib/mysql directory so it can store our database for mysql.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hollowdb
  labels:
    app: hollowdb
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hollowdb
  strategy:
    type: Recreate
  template:
```

```
    metadata:
      labels:
        app: hollowdb
    spec:
      containers:
      - name: mysql
        image: theithollow/hollowapp-blog:dbv1
        imagePullPolicy: Always
        ports:
        - containerPort: 3306
        volumeMounts:
        - name: mysqlstorage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysqlstorage
        persistentVolumeClaim:
          claimName: mysqlvol
---
apiVersion: v1
kind: Service
metadata:
  name: hollowdb
spec:
  ports:
  - name: mysql
    port: 3306
    targetPort: 3306
    protocol: TCP
  selector:
    app: hollowdb
```

Code language: JavaScript (javascript)

And now that we've got a working mysql container with persistent storage for the database, we can deploy our app.

NOTE: In this example, my application container, checks to see if there is a database for the app created already. If there is, it will use that database, if there isn't, it will create a database on the mysql server.

Also, I'm using a secret for the connection string as we've discussed in a previous post.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
      app: hollowapp
  name: hollowapp
spec:
```

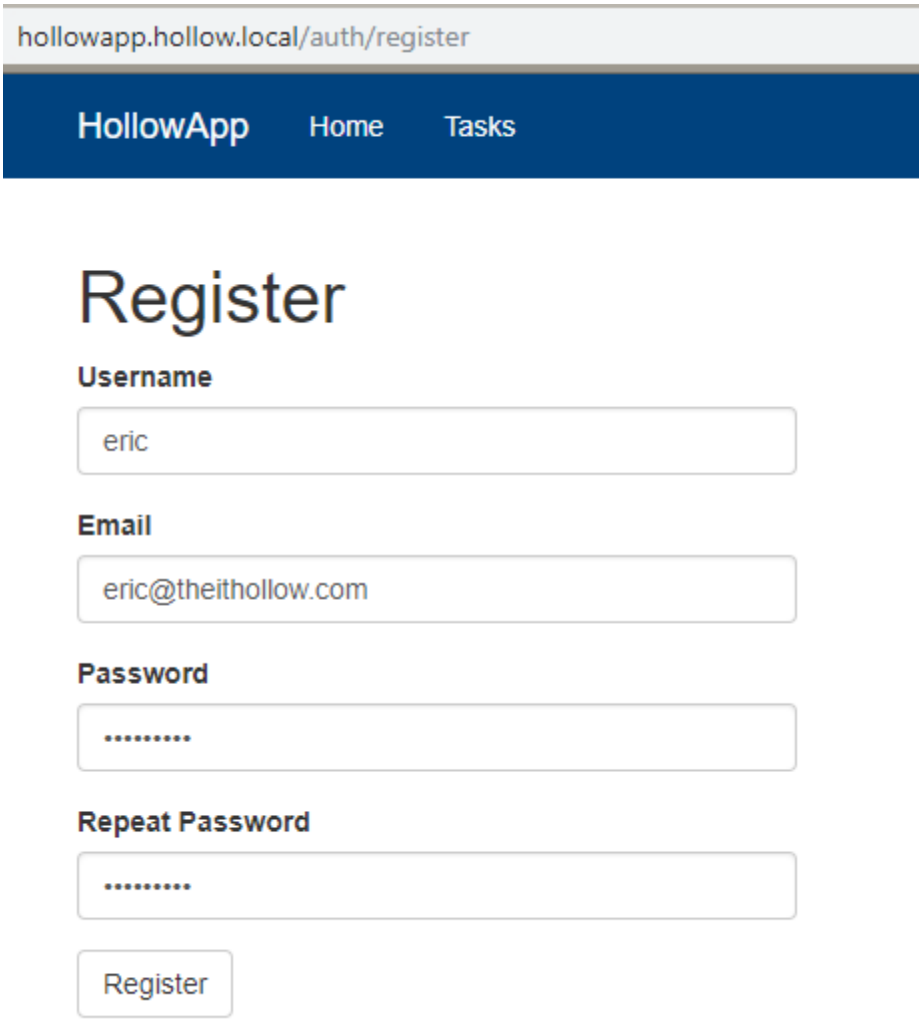
```
  replicas: 3
  selector:
    matchLabels:
      app: hollowapp
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: hollowapp
    spec:
      containers:
      - name: hollowapp
        image: theithollow/hollowapp-blog:allin1
        imagePullPolicy: Always
        ports:
        - containerPort: 5000
        env:
        - name: SECRET_KEY
          value: "my-secret-key"
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: hollow-secret
              key: db.string
---
apiVersion: v1
kind: Service
metadata:
  name: hollowapp
  labels:
    app: hollowapp
spec:
  type: ClusterIP
  ports:
  - port: 5000
    protocol: TCP
    targetPort: 5000
  selector:
    app: hollowapp
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hollowapp
  labels:
```

```
    app: hollowapp
spec:
  rules:
  - host: hollowapp.hollow.local
    http:
      paths:
      - path: /
        backend:
          serviceName: hollowapp
          servicePort: 5000
```

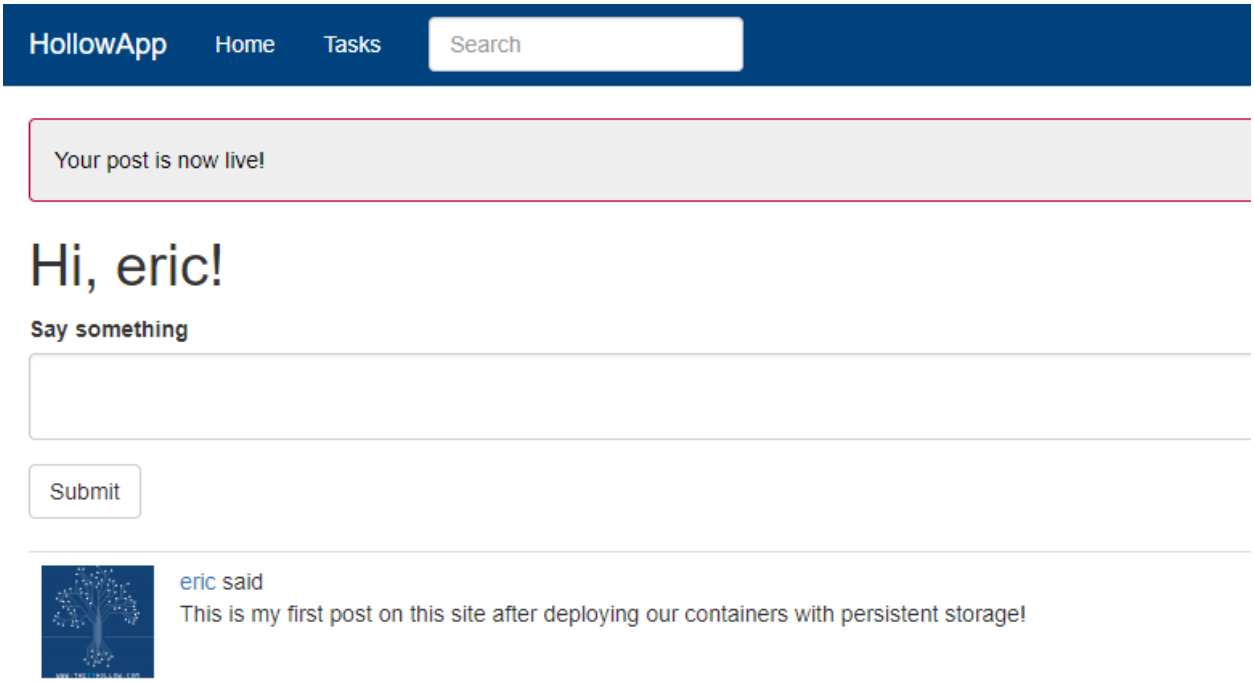
Code language: PHP (php)

Once the application has been deployed, it should connect to the database pod and set it up and then start presenting our application. Let's check by accessing it from our ingress controller.

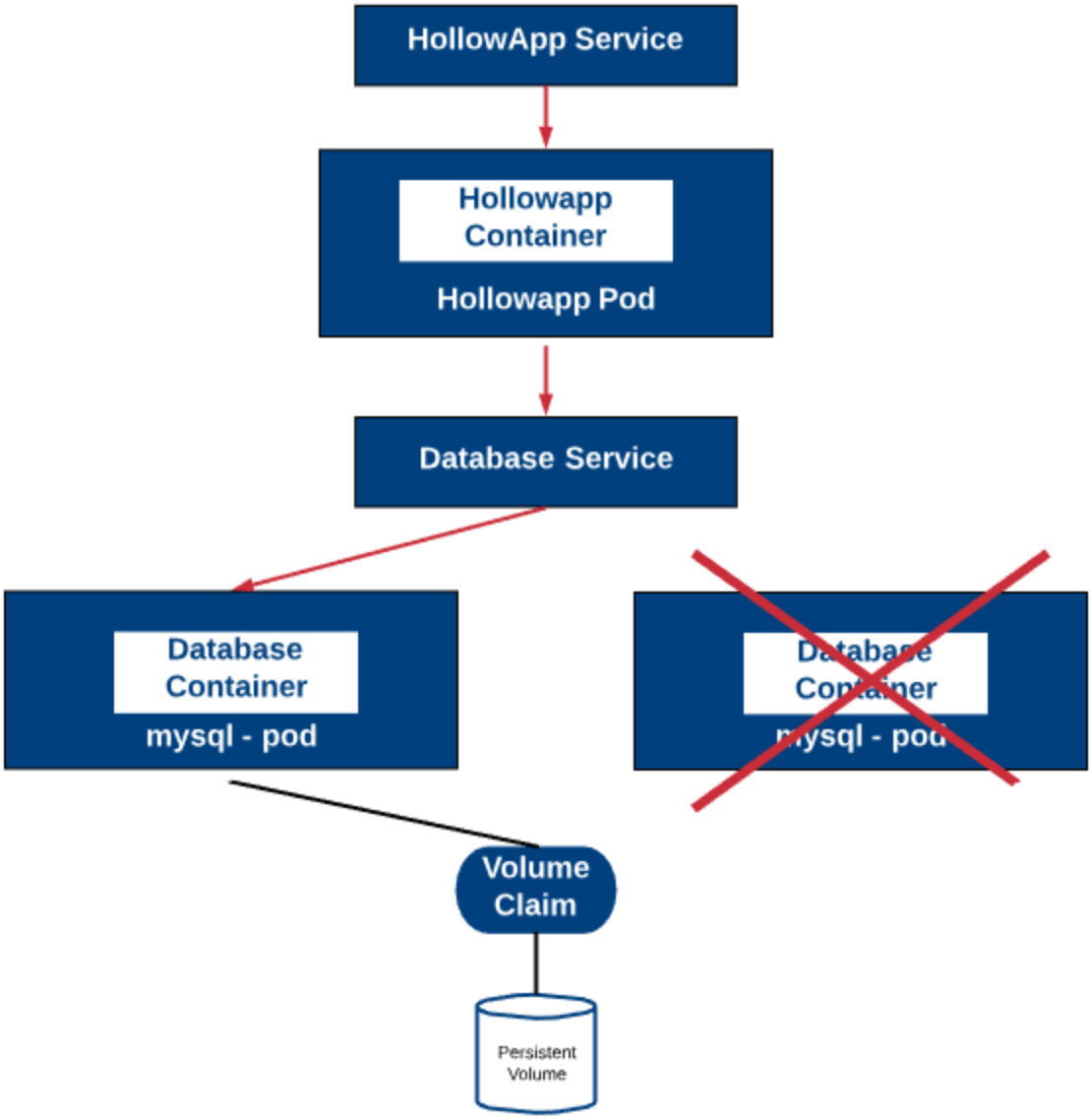
As you can see from the screenshot below, my app came up and I'm registering a user within my application as a test. This proves that I can submit data to my form and have it stored in the database pod.




After I register a user I can then submit a post just to show that we're posting data and displaying it through the application. You can see that the first post is successful and it foreshadows our next step.




Now that the app works, lets test the database resiliency. Remember that with replica set, Kubernetes will make sure that we have a certain number of pods always running. If one fails, it will be rebuilt. Great when there is no state involved. Now we have a persistent volume with our database in it. Therefore, we should be able to kill that database pod and a new one will take its place and attach to the persistent storage. The net result will be an outage, but when it comes back up, our data should still be there. The diagram below demonstrates what will happen.




So lets break some stuff!

Lets kill our database pod from the command line.

```
kubectl delete pod [database pod name]
```

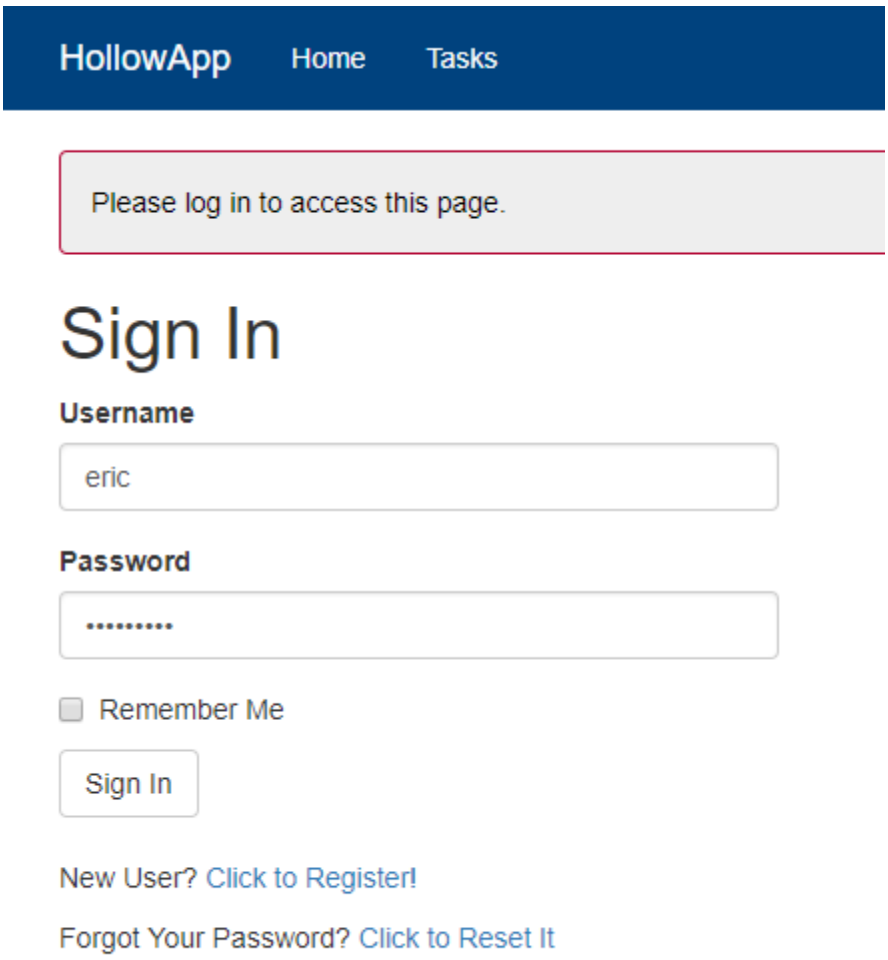
Code language: JavaScript (javascript)

```
Erics-MacBook-Pro-2:persistentvolumes eshanks$ kubectl delete pod hollowdb-7d7c68
b4d5-8xjrq
pod "hollowdb-7d7c68b4d5-8xjrq" deleted
Erics-MacBook-Pro-2:persistentvolumes eshanks$ kubectl get pods
NAME                                  READY     STATUS        RESTARTS   AGE
default-http-backend-5d4f569658-hjltf   1/1       Running       0          12d
hollowapp-846b5cf565-vc5sk            1/1       Running       0          44m
hollowdb-7d7c68b4d5-66qx9             1/1       Running       0          6m
hollowdb-7d7c68b4d5-8xjrq             1/1       Terminating   0          45m
```

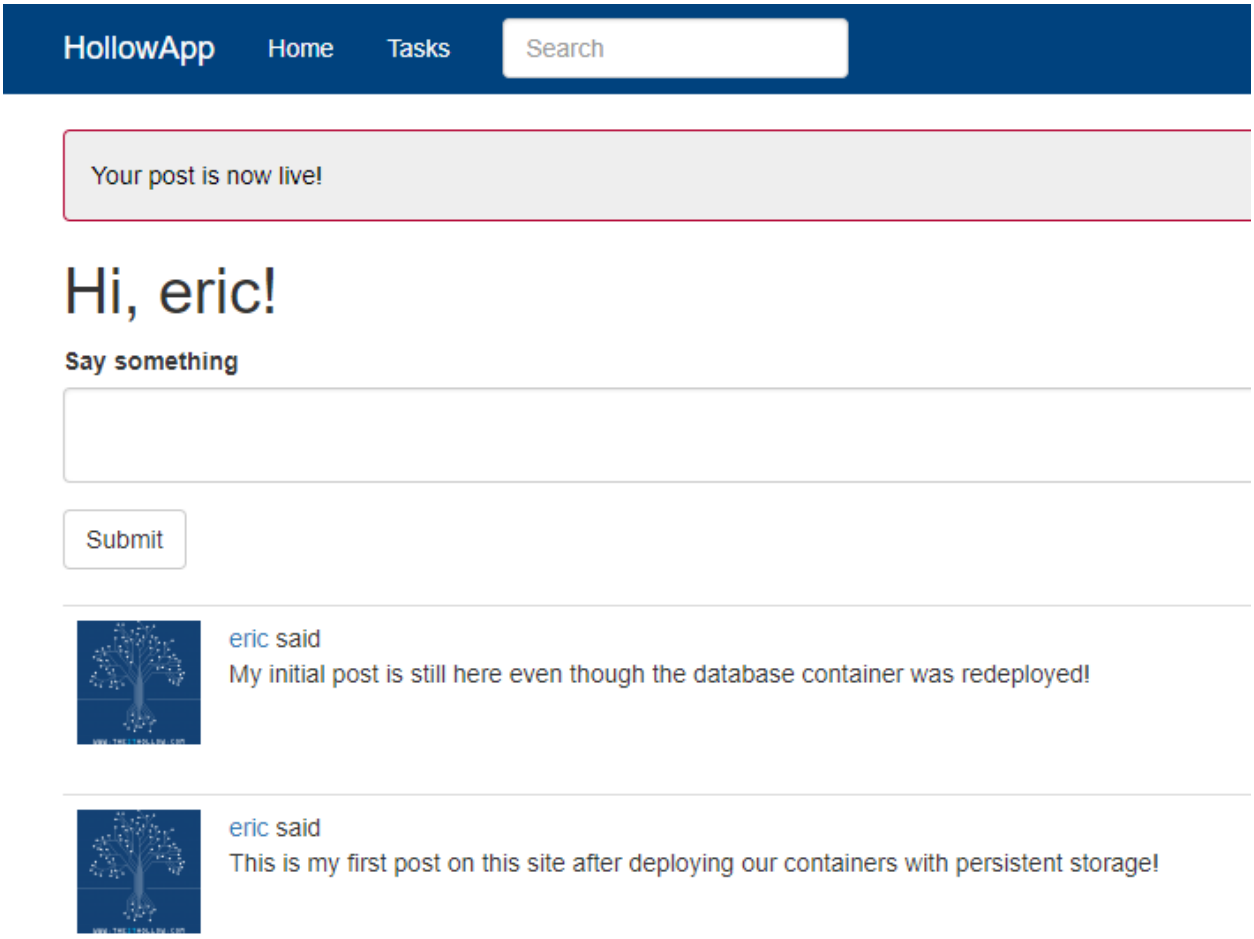
In the screenshot above you see that the pod was delete, and then I ran a “get pod” command to see whats happening. my DB pod is Terminating and a new one is in a running status already.

Let’s check the state of our application. NOTE: depending on what app you have here, it may or may not handle the loss of a database connection well or not. Mine did fine in this case.

Back in my application, I’m able to login with the user that I registered earlier, which is a good sign.




And once I’m logged in, I can see my previous post which means my database is functioning even though its in a new pod. The volume still stored the correct data and was re-attached to the new pod.




**Summary**

Well persistent volumes aren't the most interesting topic to cover around Kubernetes, but if state is involved, they are critical to the resiliency of your applications. If you're designing your applications, consider whether a pod with persistent volumes will suffice, or if maybe an external service like a cloud database is the right choice for your applications.

# Kubernetes – Cloud Providers and Storage Classes

In the previous post we covered Persistent Volumes (PV) and how we can use those volumes to store data that shouldn't be deleted if a container is removed. The big problem with that post is that we have to manually create the volumes and persistent volume claims. It would sure be nice to have those volumes spun up automatically wouldn't it? Well, we can do that with a storage class. For a storage class to be really useful, we'll have to tie our Kubernetes

cluster in with our infrastructure provider like AWS, Azure or vSphere for example. This coordination is done through a cloud provider.

## Cloud Providers – The Theory

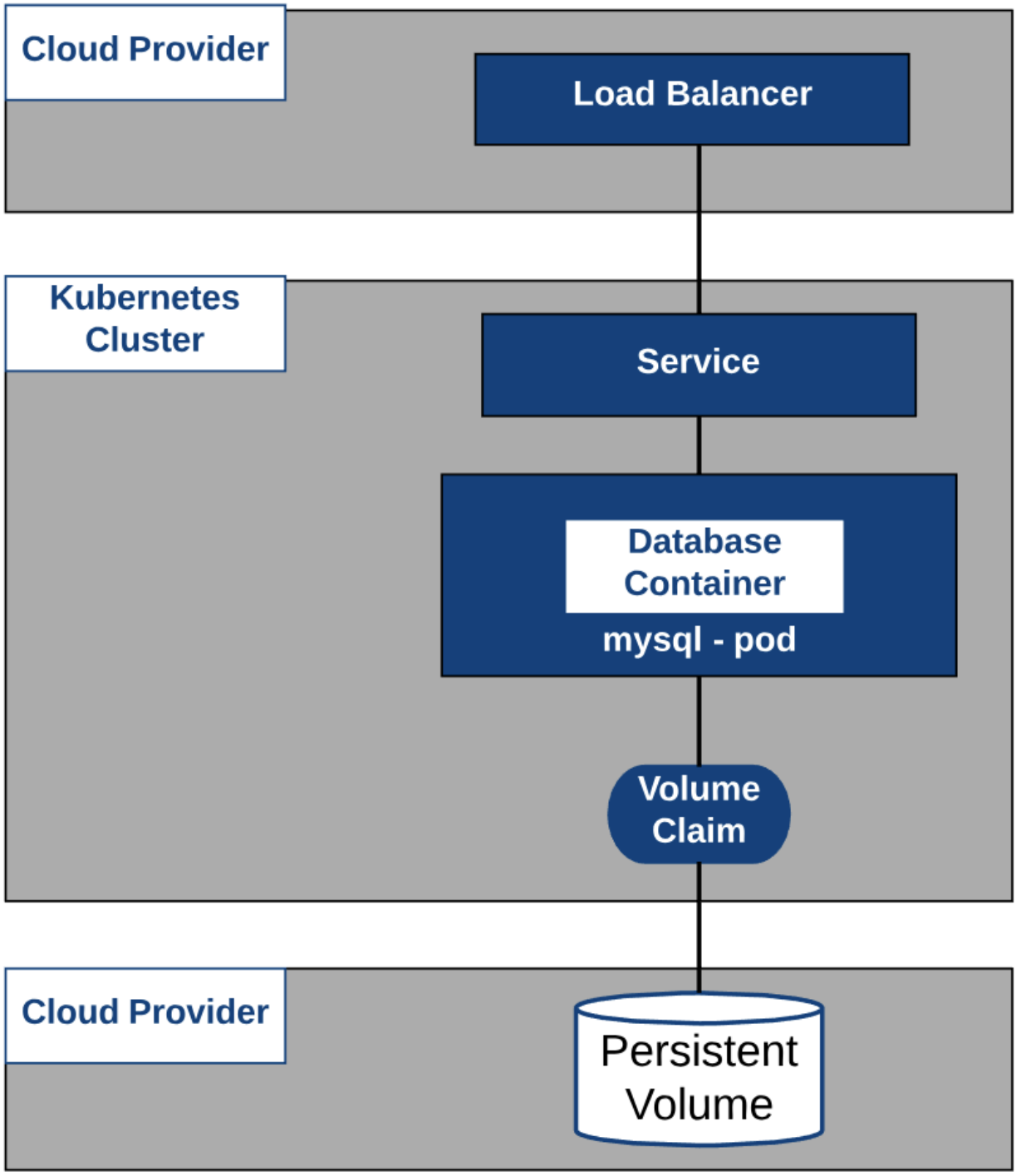
As we've learned in some of our other posts within this series, there are some objects that you can deploy through the kubectl client that don't exist inside the Kubernetes cluster. Things like a load balancer for example. If your k8s cluster is running on EC2 instances within AWS, you can have AWS deploy a load balancer that points to your ingress controller. When it comes to storage, we can take that a step further and request EBS volumes be attached to our Kubernetes cluster for our persistent volumes.

This configuration can all be controlled from the kubectl client, if you setup a cloud provider. Later in this post we'll see cloud providers in action when we setup our Cloud Provider for our vSphere environment. We won't be able to use load balancers, but we will be able to use our vSphere storage.

## Storage Classes – The Theory

Storage Classes are a nice way of giving yourself a template for the volumes that might need to be created in your Kubernetes cluster. Instead of creating a new PV for every pod, its much nicer to have a Storage Class that defines what a PV might look like. Then when the pods spin up with a persistent volume claim (PVC), it will create those PVs as they are needed from a template. Storage Classes are the mechanism that lets us deploy PVs from a template. In our Kubernetes cluster we've deployed PVs in a previous post, but we can't do this with a storage class at this time because dynamic provisioning isn't available yet. However, if we use a storage class connected with a Cloud Provider, we can use a storage class to automatically provision new vSphere storage for our cluster.

When we combine storage classes and Cloud Providers together, we can get some pretty neat results.




## Cloud Provider – In Action

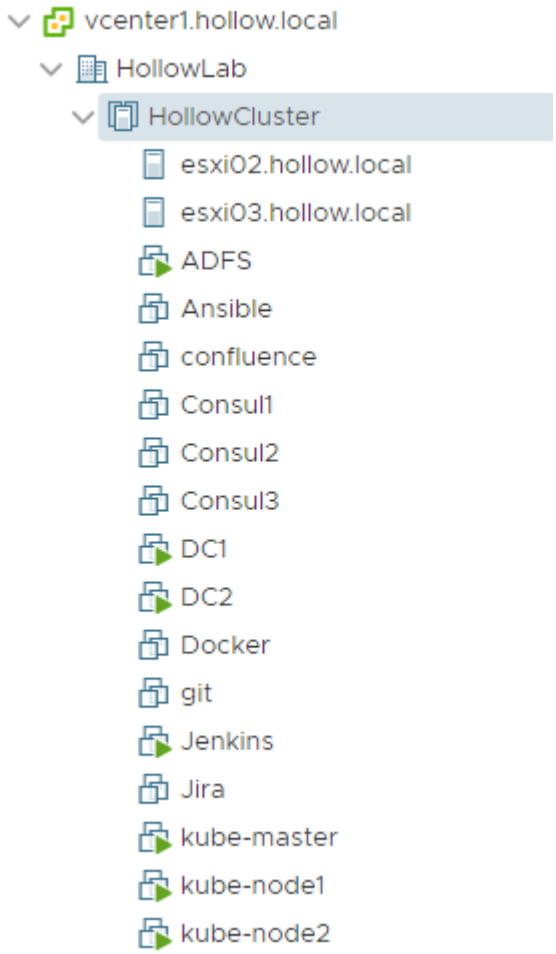
Before we can do much with our cloud provider, we need to set it up. I will be doing this on my vSphere hosted k8s cluster that was deployed in this previous post. To setup the cloud

provider, I'll be rebuilding this cluster with the kubeadm reset command to reset the cluster and then performing the kubeadm init process from scratch.

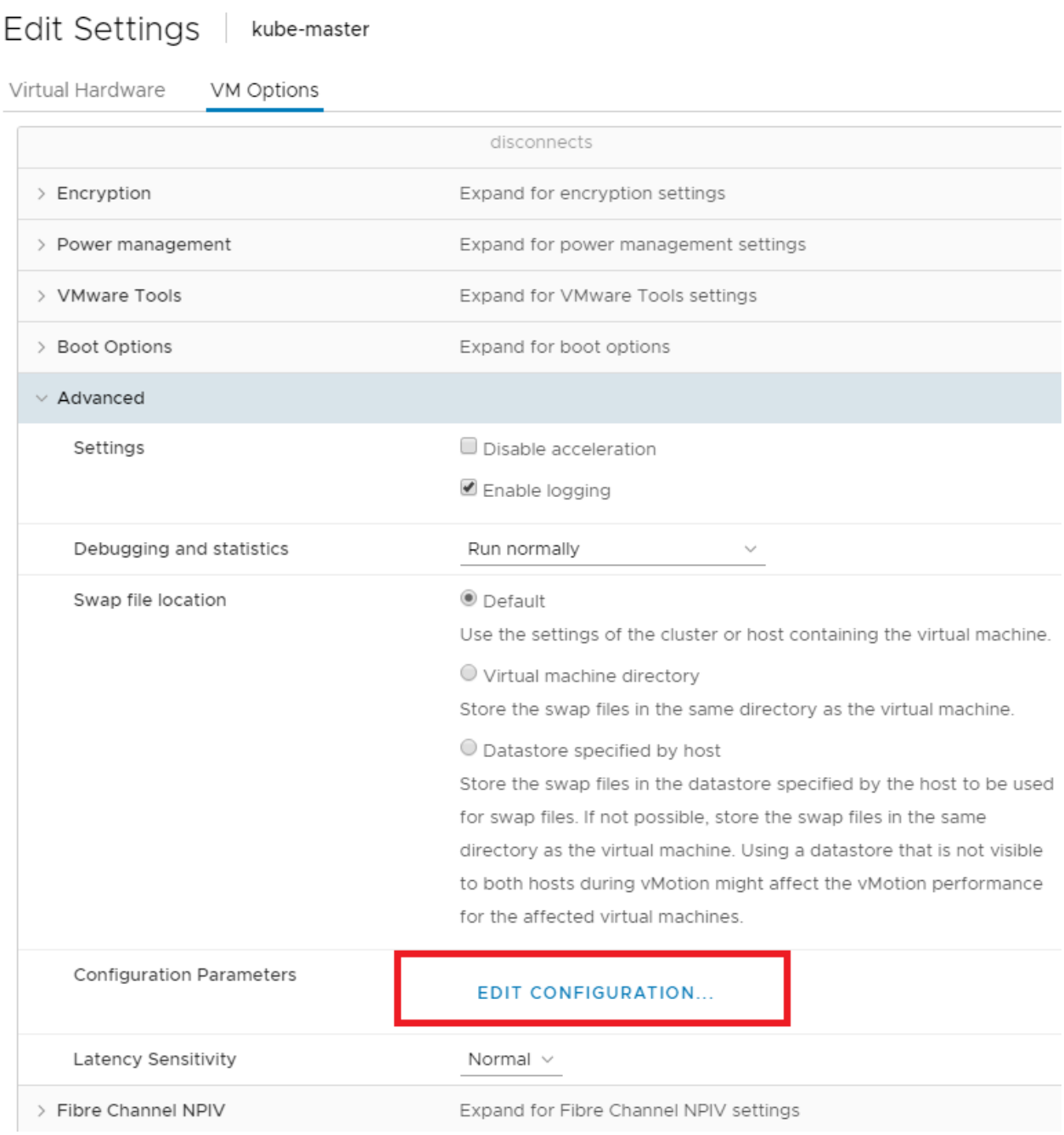
**Sidebar:** I had some help with this configuration and wanted to send a shout-out to Tim Carr who is a Kubernetes Architect at VMware as well as an all around good guy.

### vSphere Setup

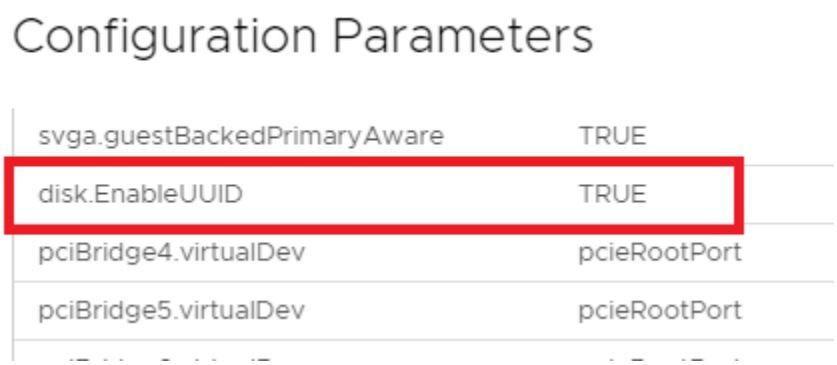
Before creating any Kubernetes configs, we need to make sure our vSphere environment is ready to be used by k8s. I've created an administrator accounts which permissions to the cluster, VMs, and datastores. Here is a look at my hosts and clusters view so you can see how I created config files later in the post.



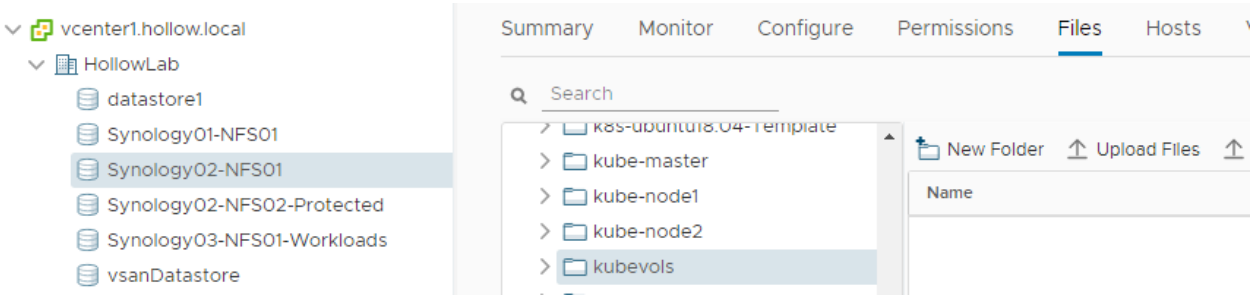

Now don't go to far, you'll need to make sure that an advanced configuration setting is added to your Kubernetes nodes. This setting ensures that Kubernetes can identify the disks that it might need to add to a pod from vSphere. To do this, power off your k8s cluster and modify the vm properties of the nodes. Go to VM Options and click the Edit Configurations link next to Configuration Parameters in the Advanced section.



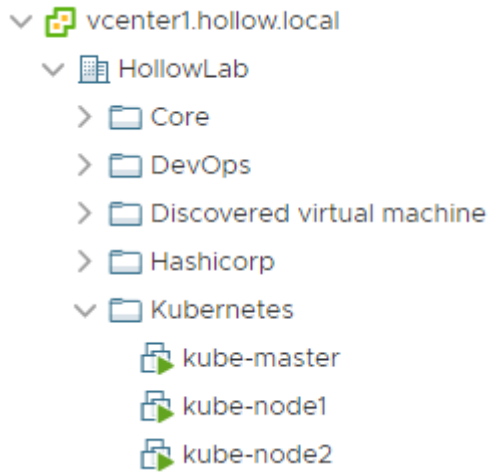

From there, add the key “disk.EnableUUID” and the corresponding value of “True”.




I've also created a folder called "kubevol" in one of my vSphere datastores where new Kubernetes persistent volumes will live.




Lastly, all of my k8s nodes live within a single virtual machine folder, which is important.




**Kubeadm init Setup**

First, we need to create a config file that will have configuration information for the connection to the vSphere environment. It will include the location of the Kubernetes hosts, datastores and connection strings shown in the above screenshots. Here is the example file I used in my lab. I saved it as vsphere.conf for now and placed it in the /etc/kubernetes directory on my master node.

**NOTE:** It is possible to use a Base64 encoded password, which I didn't do for this post. Just note that it can be done and should be used over plain text for any production environments. This is a getting stated post so we've eliminated some of the complexity here.

```
[Global]
user = "k8s@hollow.local"
password = "Password123"
port = "443"
insecure-flag = "1"
datacenters = "HollowLab"

[VirtualCenter "vcenter1.hollow.local"]

[Workspace]
server = "vcenter1.hollow.local"
datacenter = "HollowLab"
default-datastore="Synology02-NFS01"
resourcepool-path="HollowCluster/Resources"
folder = "Kubernetes"

[Disk]
scsicontrollertype = pvscsi
```

Code language: PHP (php)

If you're wondering what this config file contains, here are some descriptions of the fields.

- **user –** Username with admin access to the vSphere cluster to create storage
- **password –** The credentials for the user to login
- **port –** The port used to connect to the vCenter server
- insecure-flag – Setting this to "1" means it will accept the certificate if it isn't trusted
- **datacenters –** The datacenter within vCenter where Kubernetes nodes live
- **server –** The vCenter to connect to
- default datastore – Datastore where persistent volumes will be created
- **resourcepool-path –** The resource pool where the Kubernetes nodes live.
- **folder –** the folder where your Kubernetes nodes live. They should be in their own folder within vSphere.
- **scsicontrollertype –** Which type of vSphere scsi controller type should be used

The config file you created in the previous step is the connection information for vSphere. However, when you run the Kubeadm init configuration, you need a configuration file that's used to bootstrap your cluster.

We'll need to create a yaml file that will be used during the kubeadm init process. Here is the config file for the kubeadm init. Note that the cloud provider is "vsphere" and and I've updated the cloud-config value to the path to the vsphere.conf file we just created in the previous step. I've placed both this config file and the vsphere.conf file in /etc/kubernetes/ directory on the k8s master.

```
apiVersion: kubeadm.k8s.io/v1beta1
kind: InitConfiguration
bootstrapTokens:
      - groups:
        - system:bootstrappers:kubeadm:default-node-token
        token: cba33r.3f565pcpa8vbxdpu #generate your own token here
prior or use an autogenerated token
        ttl: 0s
        usages:
        - signing
        - authentication
nodeRegistration:
  kubeletExtraArgs:
    cloud-provider: "vsphere"
    cloud-config: "/etc/kubernetes/vsphere.conf"
---
apiVersion: kubeadm.k8s.io/v1beta1
kind: ClusterConfiguration
kubernetesVersion: v1.13.4
apiServer:
  extraArgs:
    cloud-provider: "vsphere"
    cloud-config: "/etc/kubernetes/vsphere.conf"
  extraVolumes:
  - name: cloud
    hostPath: "/etc/kubernetes/vsphere.conf"
    mountPath: "/etc/kubernetes/vsphere.conf"
controllerManager:
  extraArgs:
    cloud-provider: "vsphere"
    cloud-config: "/etc/kubernetes/vsphere.conf"
  extraVolumes:
  - name: cloud
    hostPath: "/etc/kubernetes/vsphere.conf"
    mountPath: "/etc/kubernetes/vsphere.conf"
networking:
  podSubnet: "10.244.0.0/16"
```

Code language: PHP (php)

```
[root@kube-master kubernetes]# ls
manifests  pki  vsphere.conf  vsphere-init.yml
```

Now, we have the details for our Kubernetes cluster to be re-initialized with the additional connection information for our vSphere environment. Lets re-run the kubeadm init command now with our new configuration file.

```
kubeadm init --config /etc/kubernetes/[filename].yml
```

```
[root@kube-master ~]# kubeadm init --config /etc/kubernetes/vsphere-init.yml
[init] Using Kubernetes version: v1.13.4
[preflight] Running pre-flight checks
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet connection
[preflight] You can also perform this action in beforehand using 'kubeadm config images pull'
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
```

The result will be the setup of your Kubernetes cluster again on the master node. After this is done, we still need to have our other worker nodes join the cluster much like we did in the original kubeadm post. Before we do this though, we want to create another config file on the worker nodes so that they know they are joining a cluster and have the vSphere provider available to them. We will also need some information from the master node that will be passed along during the join. To get this information run:

```
kubectl -n kube-public get configmap cluster-info -o jsonpath='{.data.kubeconfig}' > config.yml
```

Code language: JavaScript (javascript)

Take this config.yml file that was created on the master and place it on the k8s worker nodes. This discovery file provides information needed to join the workers to the cluster properly.

Create another yaml file in the /etc/kubernetes directory called vsphere-join.yml. This file should contain the join token presented at the last step of the kubeadm init provided earlier.

```
apiVersion: kubeadm.k8s.io/v1alpha3
kind: JoinConfiguration
discoveryFile: config.yml
token: cba33r.3f565pcpa8vbxdpu #token used to init the k8s cluster. Yours should be different
nodeRegistration:
  kubeletExtraArgs:
    cloud-provider: vsphere #cloud provider is enabled on worker
```

Code language: PHP (php)

Run the kubeadm joint command like this on the worker nodes.

```
kubeadm join --config /etc/kubernetes/vsphere-join.yml
```

```
[root@kube-node2 ~]# kubeadm join --config /etc/kubernetes/vsphere-join.yml
[preflight] Running pre-flight checks
[discovery] Created cluster-info discovery client, requesting info from "https
[discovery] Synced cluster-info information from the API Server so we have got
[join] Reading configuration from the cluster...
[join] FYI: You can look at this config file with 'kubectl -n kube-system get
[kubelet] Downloading configuration for the kubelet from the "kubelet-config-
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/confi
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/
[kubelet-start] Activating the kubelet service
[tlsbootstrap] Waiting for the kubelet to perform the TLS Bootstrap...
```

When the cluster is setup, there will be a connection to your vSphere environment that is housing the k8s nodes. Don't forget to set your KUBECONFIG path to the admin.conf file and then deploy your flannel networking pods with the:

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/bc79dd1505b0c8681ece4de4c0d86c5cd2643275/Documentation/kube-flannel.yml
```
Code language: JavaScript (javascript)

Let's move on to storage classes for now that the cluster has been initialized and in a ready state.

```
[root@kube-node2 ~]# kubectl get nodes
NAME          STATUS   ROLES    AGE     VERSION
kube-master   Ready    master   22m     v1.13.4
kube-node1    Ready    <none>   4m44s   v1.13.4
kube-node2    Ready    <none>   2m11s   v1.13.4
```

## Storage Classes – In Action

Before we create a Storage Class, login to your vSphere environment and navigate to your datastore where new disks should be created. Create a kubevols folder in that datastore.

Lets create a manifest file for a storage class. Remember that this class acts like a template to create PVs so the format may look familiar to you. I've created a new class named "thin-disk" which will create a thin provisioned disk provided by vSphere. I've also made this StorageClass the default, which you can of course only have 1 at a time. Note the provisioner listed here, as it is specific to the cloud provider being used.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: vsphere-disk #name of the default storage class
  annotations:
    storageclass.kubernetes.io/is-default-class:   "true"   #Make this a default storage class. There can be only 1 or it won't work.
provisioner: kubernetes.io/vsphere-volume
parameters:
  diskformat: thin
```

Deploy the Kubernetes manifest file with the apply command.

```
kubectl apply -f [manifest file].yml
```
Code language: CSS (css)

You can check the status of your StorageClass by running:

```
kubectl get storageclass
```
Code language: JavaScript (javascript)

```
$ kubectl get storageclass
NAME                    PROVISIONER                    AGE
vsphere-disk (default)  kubernetes.io/vsphere-volume   6s
```

Now to test out that storage class and see if it will create new PVs if a claim is issued. Create a new persistent volume claim and apply it. Here is one you can test with:

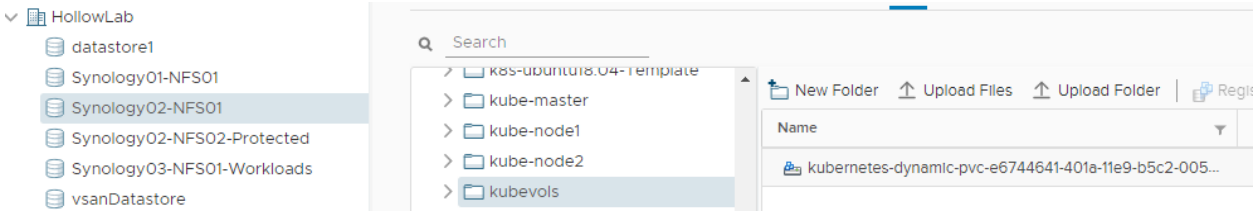
```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: test-claim
  annotations:
    volume.beta.kubernetes.io/storage-class: vsphere-disk
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

Apply it with the kubectl apply -f [manifest].yml command.

When you apply the PVC you can check the status with a kubectl get PVC command to see whats happening. You can see from the screenshot below that good things are happening because the status is "Bound."

```
$ kubectl get pvc
NAME         STATUS    VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS
test-claim   Bound     pvc-e6744641-401a-11e9-b5c2-00505693ee4c   2Gi        RWO            vsphere-disk
```

Now, for a quick look in the vSphere environment, we can look in the kubevols folder in the datastore we specified. That claim that was deployed used the Storage Class to deploy a vmdk for use by the Kubernetes cluster.




**Summary**

Persistent Volumes are necessary, and dynamically deploying them with a storage class may become necessary depending on your application. Adding a cloud provider can really increase the usability of your Kubernetes cluster, especially if its located within a public cloud where you can use a variety of services with your containers.

# Kubernetes – StatefulSets

We love deployments and replica sets because they make sure that our containers are always in our desired state. If a container fails for some reason, a new one is created to replace it. But what do we do when the deployment order of our containers matters? For that, we look for help from Kubernetes StatefulSets.

# StatefulSets – The Theory

StatefulSets work much like a Deployment does. They contain identical container specs but they ensure an order for the deployment. Instead of all the pods being deployed at the same time, StatefulSets deploy the containers in sequential order where the first pod is deployed and ready before the next pod starts. (NOTE: it is possible to deploy pods in parallel if you need them to, but this might confuse your understanding of StatefulSets for now, so ignore that.) Each of these pods has its own identity and is named with a unique ID so that it can be referenced.

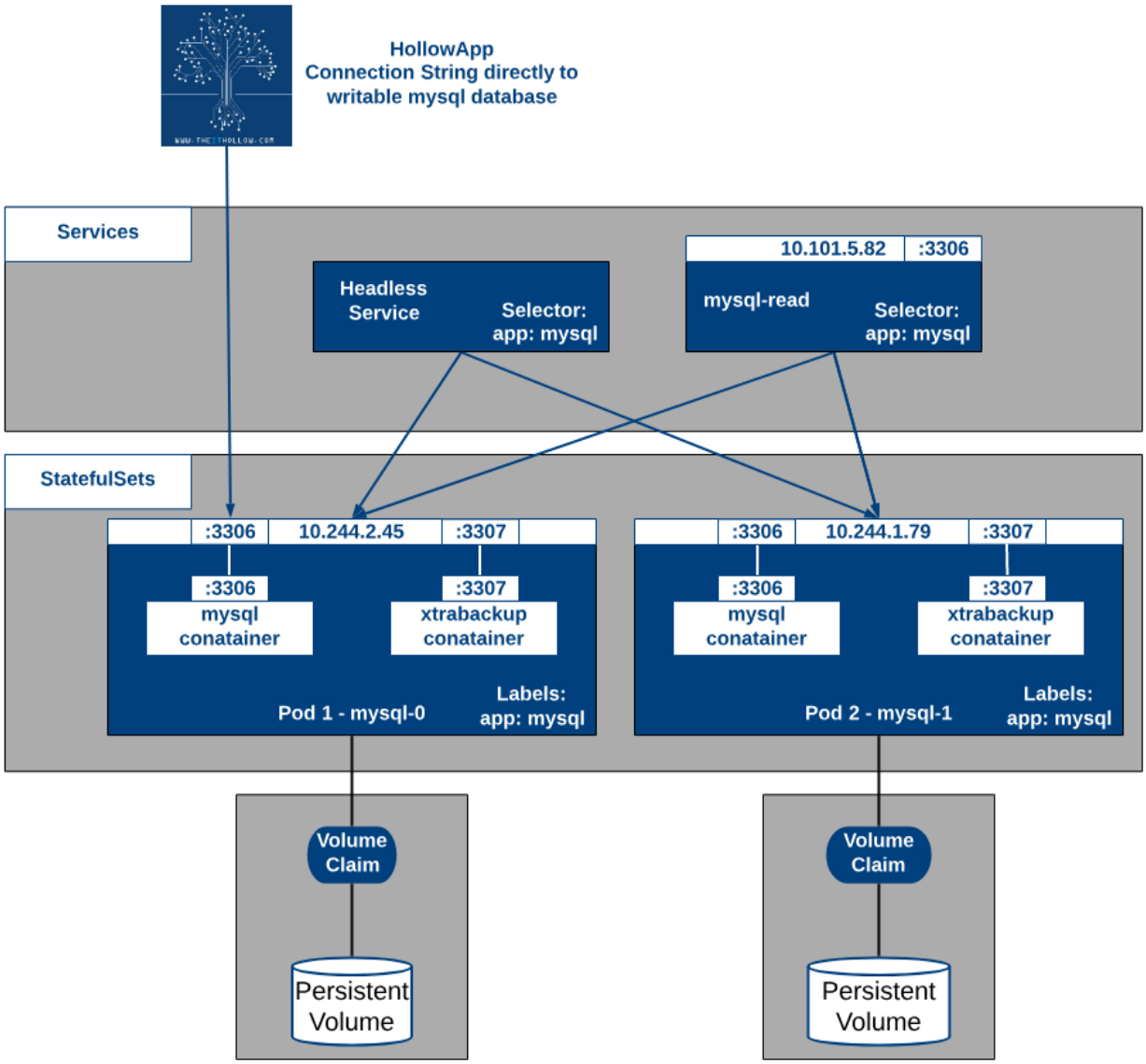
OK, now that we know how the deployment of a StatefulSet works, what about the failure of a pod that is part of a StatefulSet? Well, the order is preserved. If we lose pod-2 due to a host failure, the StatefulSet won't just deploy another pod, it will deploy a new "pod-2" because the identity matters in a StatefulSet.

StatefulSets currently require a "headless" service to manage the identities. This is a service that has the same selectors that you're used to, but won't receive a clusterIP address, meaning that you can't route traffic to the containers through this service object.

# StatefulSets – In Action

The example that this blog post will use for a StatefulSet come right from the Kubernetes website for managing a mysql cluster. In this example, we'll deploy a pair of pods with some mysql containers in them. This example also uses a sidecar container called xtrabackup which is used to aid in mysql replicaiton between mysql instances. So why a StatefulSet for mysql? Well, the order matters of course. Our first pod that gets deployed will contain our mysql master database where reads and writes are completed. The additional containers will contain the mysql replicated data but can only be used for read operations. The diagram below shows the setup of the application we'll be creating.

The diagram below shows our application (which is also made up of a set of containers and services, but doesn't matter in this case) and it connects directly to one of the containers. There is a headless service used to maintain the network identity of the pods, and another service that provides read access to the pods. The Pods have a pair of containers (mysql as the main container, and xtrabackup as a sidecar for replication). And we are also creating persistent storage based on the storage class we created in the Cloud Providers and Storage Classes post.




Before we get to deploying anything, we will create a new configmap. This config map has configuration information used by the containers to get an identity at boot time. The mysql configuration data below ensures that the master mysql container becomes master and the other containers are read-only.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql
  labels:
    app: mysql
data:
  master.cnf: |
    # Apply this config only on the master.
    [mysqld]
```

```
    log-bin
  slave.cnf: |
    # Apply this config only on slaves.
    [mysqld]
    super-read-only
```
Code language: PHP (php)

Deploy the manifest above by running:

```
kubectl apply -f [manifest file].yml
```
Code language: CSS (css)

Next, we'll create our mysql services. Here we'll create the headless service for use with our StatefulSet to manage the identities, as well as a service that handles traffic for mysql reads.

```
# Headless service for stable DNS entries of StatefulSet members.
apiVersion: v1
kind: Service
metadata:
  name: mysql
  labels:
    app: mysql
spec:
  ports:
  - name: mysql
    port: 3306
  clusterIP: None
  selector:
    app: mysql
---
# Client service for connecting to any MySQL instance for reads.
# For writes, you must instead connect to the master: mysql-0.mysql.
apiVersion: v1
kind: Service
metadata:
  name: mysql-read
  labels:
    app: mysql
spec:
  ports:
  - name: mysql
    port: 3306
  selector:
    app: mysql
```
Code language: PHP (php)

Deploy the above manifest file by running:

```
kubectl apply -f [manifest file].yml
```
Code language: CSS (css)

Once the services are deployed, we can check them by looking at the command:

```
kubectl get svc
```

Code language: JavaScript (javascript)

```
Erics-MacBook-Pro-2:hollowapp eshanks$ kubectl get svc
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
hollowapp    ClusterIP   10.104.114.61   <none>        5000/TCP   1d
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP    11d
mysql        ClusterIP   None            <none>        3306/TCP   2d
mysql-read   ClusterIP   10.101.5.82     <none>        3306/TCP   2d
```

Notice that the CLUSTER-IP for the mysql service is “None”. This is our headless service for our StatefulSet.

Lastly, we deploy our StatefulSet. This manifest includes several configs that we haven’t talked about including initcontainers. The init containers spin up prior to the normal pods and take an action. In this case, they reference the pod ID and create a mysql config based on that value to ensure the pods know if they are the master pod, or the secondary pods that are read-only.

You’ll also see that there are scripts written into this manifest file that setup the replication by using the xtrabackup containers.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  serviceName: mysql
  replicas: 2
  template:
    metadata:
      labels:
        app: mysql
    spec:
      initContainers:
      - name: init-mysql
        image: mysql:5.7
        command:
        - bash
        - "-c"
        - |
          set -ex
          # Generate mysql server-id from pod ordinal index.
          [[ `hostname` =~ -([0-9]+)$ ]] || exit 1
          ordinal=${BASH_REMATCH[1]}
          echo [mysqld] > /mnt/conf.d/server-id.cnf
```

```
          # Add an offset to avoid reserved server-id=0 value.
          echo server-id=$((100 + $ordinal)) >> /mnt/conf.d/server-
id.cnf
          # Copy appropriate conf.d files from config-map to emptyDir.
          if [[ $ordinal -eq 0 ]]; then
            cp /mnt/config-map/master.cnf /mnt/conf.d/
          else
            cp /mnt/config-map/slave.cnf /mnt/conf.d/
          fi
        volumeMounts:
        - name: conf
          mountPath: /mnt/conf.d
        - name: config-map
          mountPath: /mnt/config-map
      - name: clone-mysql
        image: gcr.io/google-samples/xtrabackup:1.0
        command:
        - bash
        - "-c"
        - |
          set -ex
          # Skip the clone if data already exists.
          [[ -d /var/lib/mysql/mysql ]] && exit 0
          # Skip the clone on master (ordinal index 0).
          [[ `hostname` =~ -([0-9]+)$ ]] || exit 1
          ordinal=${BASH_REMATCH[1]}
          [[ $ordinal -eq 0 ]] && exit 0
          # Clone data from previous peer.
          ncat --recv-only mysql-$(($ordinal-1)).mysql 3307 | xbstream
-x -C /var/lib/mysql
          # Prepare the backup.
          xtrabackup --prepare --target-dir=/var/lib/mysql
        volumeMounts:
        - name: data
          mountPath: /var/lib/mysql
          subPath: mysql
        - name: conf
          mountPath: /etc/mysql/conf.d
      containers:
      - name: mysql
        image: mysql:5.7
        env:
        - name: MYSQL_ALLOW_EMPTY_PASSWORD
          value: "1"
        ports:
        - name: mysql
```

```
          containerPort: 3306
        volumeMounts:
        - name: data
          mountPath: /var/lib/mysql
          subPath: mysql
        - name: conf
          mountPath: /etc/mysql/conf.d
        resources:
          requests:
            cpu: 500m
            memory: 1Gi
        livenessProbe:
          exec:
            command: ["mysqladmin", "ping"]
          initialDelaySeconds: 30
          periodSeconds: 10
          timeoutSeconds: 5
        readinessProbe:
          exec:
            # Check we can execute queries over TCP (skip-networking is
off).
            command: ["mysql", "-h", "127.0.0.1", "-e", "SELECT 1"]
          initialDelaySeconds: 5
          periodSeconds: 2
          timeoutSeconds: 1
      - name: xtrabackup
        image: gcr.io/google-samples/xtrabackup:1.0
        ports:
        - name: xtrabackup
          containerPort: 3307
        command:
        - bash
        - "-c"
        - |
          set -ex
          cd /var/lib/mysql
```

```
          # Determine binlog position of cloned data, if any.
          if [[ -f xtrabackup_slave_info ]]; then
            # XtraBackup already generated a partial "CHANGE MASTER TO"
query
            # because we're cloning from an existing slave.
            mv xtrabackup_slave_info change_master_to.sql.in
            # Ignore xtrabackup_binlog_info in this case (it's useless).
            rm -f xtrabackup_binlog_info
          elif [[ -f xtrabackup_binlog_info ]]; then
```

```
            # We're cloning directly from master. Parse binlog position.
            [[ `cat xtrabackup_binlog_info` =~ ^(.*?)[[:space:]]+(.*?)$
]] || exit 1
            rm xtrabackup_binlog_info
            echo              "CHANGE              MASTER              TO
MASTER_LOG_FILE='${BASH_REMATCH[1]}',\
                  MASTER_LOG_POS=${BASH_REMATCH[2]}"                    >
change_master_to.sql.in
          fi

          # Check if we need to complete a clone by starting replication.
          if [[ -f change_master_to.sql.in ]]; then
            echo   "Waiting   for   mysqld   to   be   ready   (accepting
connections)"
            until mysql -h 127.0.0.1 -e "SELECT 1"; do sleep 1; done

            echo "Initializing replication from clone position"
            # In case of container restart, attempt this at-most-once.
            mv change_master_to.sql.in change_master_to.sql.orig
            mysql -h 127.0.0.1 <<EOF
          $(<change_master_to.sql.orig),
            MASTER_HOST='mysql-0.mysql',
            MASTER_USER='root',
            MASTER_PASSWORD='',
            MASTER_CONNECT_RETRY=10;
          START SLAVE;
          EOF
          fi

          # Start a server to send backups when requested by peers.
          exec ncat --listen --keep-open --send-only --max-conns=1 3307
-c \
            "xtrabackup   --backup   --slave-info   --stream=xbstream   --
host=127.0.0.1 --user=root"
        volumeMounts:
        - name: data
          mountPath: /var/lib/mysql
          subPath: mysql
        - name: conf
          mountPath: /etc/mysql/conf.d
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
      volumes:
      - name: conf
```

```
        emptyDir: {}
      - name: config-map
        configMap:
          name: mysql
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 10Gi
```
Code language: PHP (php)

Deploy the manifest file above by running:

```
kubectl apply -f [manifest file].yml
```
Code language: CSS (css)

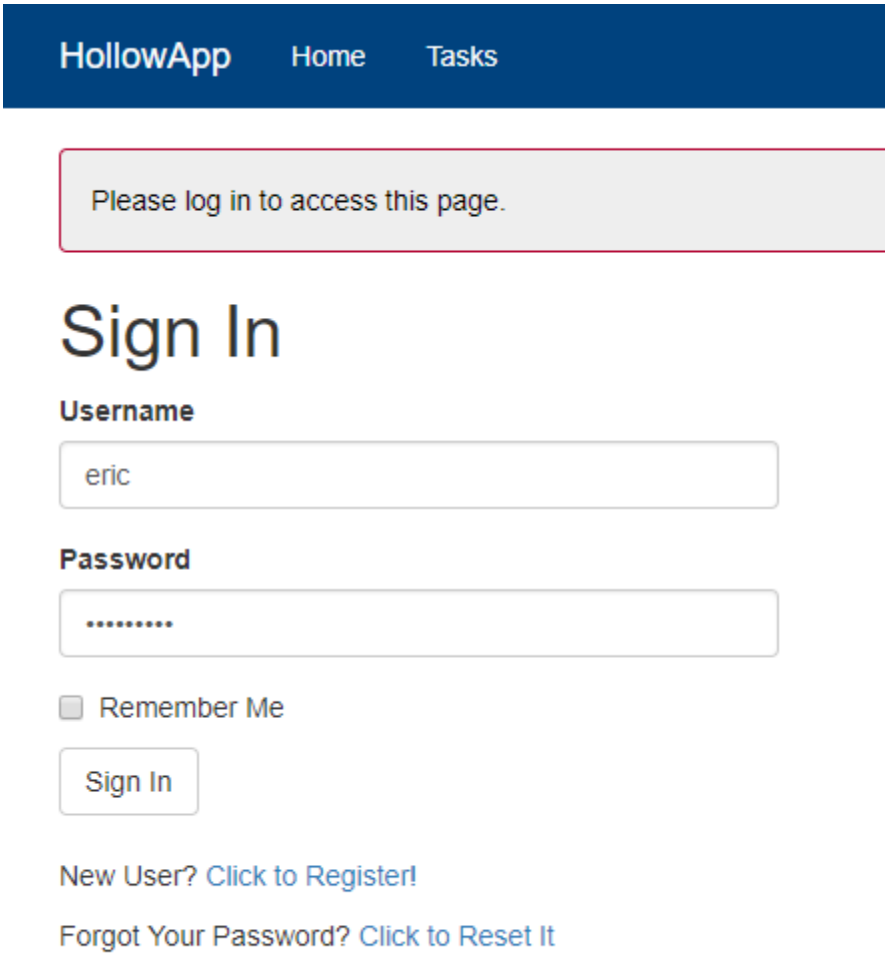
Once the containers have been deployed, we can check on them with the kubectl get pods command.

```
kubectl get pods
```
Code language: JavaScript (javascript)

```
Erics-MacBook-Pro-2:hollowapp eshanks$ kubectl get pod
NAME                          READY     STATUS    RESTARTS   AGE
hollowapp-5cdd56c77d-xnwb4    1/1       Running   0          1d
mysql-0                       2/2       Running   0          1d
mysql-1                       2/2       Running   0          1d
```

Here, you can see that we have two mysql pods and they contain 2 containers. Each are running and they are numbered with the [podname]-X naming structure where we have a mysql-0 and a mysql-1. Our application needs to be able to write data, so it will be configured to write to mysql-0. However, if we have other reports that are being reviewed, those read only reports could be build based on the data from the other containers to reduce load on the writable container.

We can deploy our application where it is writing directly to the mysql-0 pod and the application comes up as usual.




**Summary**

Sometimes, you need to treat your containers more like pets than cattle. In some cases not all containers are the same, like the example in this post where our mysql-0 server is our master mysql instance. When the order of deployment matters, and not all containers are identical, a StatefulSet might be your best bet.

# Kubernetes – Role Based Access

As with all systems, we need to be able to secure a Kubernetes cluster so that everyone doesn't have administrator privileges on it. I know this is a serious drag because no one wants to deal with a permission denied error when we try to get some work done, but permissions are important to ensure the safety of the system. Especially when you have multiple groups accessing the same resources. We might need a way to keep those groups from stepping on each other's work, and we can do that through role based access controls.

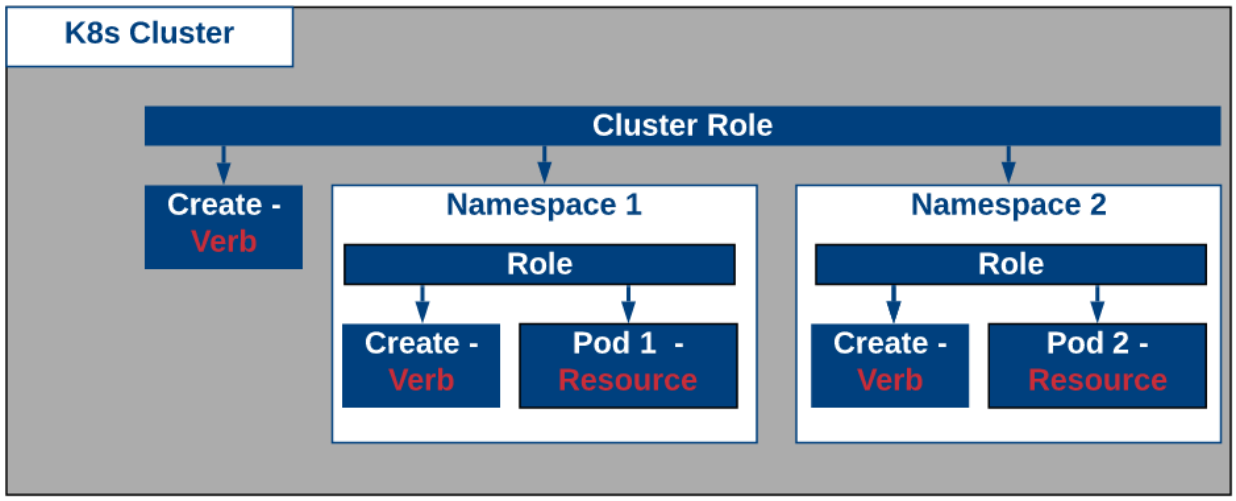
# Role Based Access – The Theory

Before we dive too deep, lets first understand the three pieces in a Kubernetes cluster that are needed to make role based access work. These are Subjects, Resources, and Verbs.

- Subjects – Users or processes that need access to the Kubernetes API.
- Resources – The k8s API objects that you'd grant access to
- Verbs – List of actions that can be taken on a resource

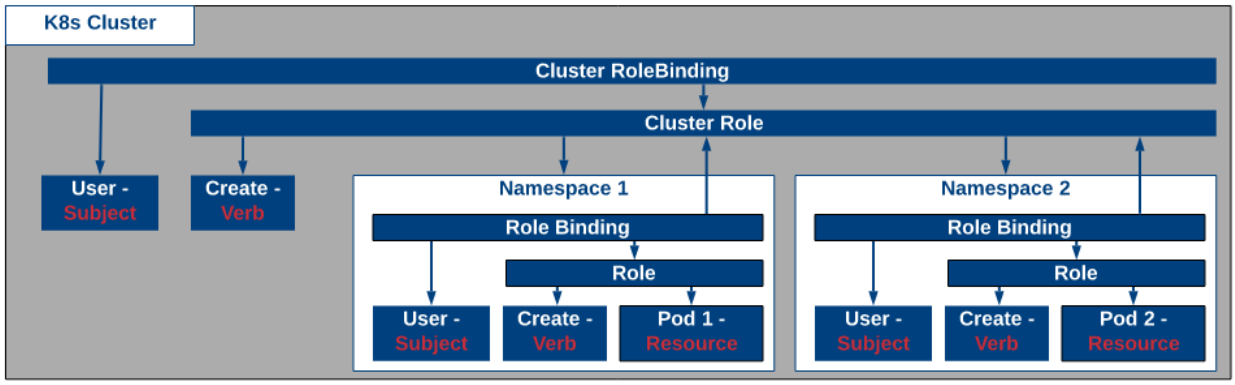
These three items listed above are used in concert to grant permissions such that a user (**Subject**) is allowed access to take an action (**verb**) on an object (**Resource**).

Now we need to look at how we tie these three items together in Kubernetes. The first step will be to create a Role or a ClusterRole. Now both of these roles will be used to tie the **Resources** together with a **Verb**, the difference between them is that a Role is used at a namespace level whereas a ClusterRole is for the entire cluster.




Once you've created your Role or your Cluster Role, you've tied the **Resource** and **Verb** together and are only missing the **Subject** now. To tie the Subject to the Role, a RoleBinding or ClusterRoleBinding is needed. As you can guess the difference between a RoleBinding or a ClusterRoleBinding is whether or not its done at the namespace or for the entire Cluster, much like the Role/ClusterRole described above.

It should be noted that you can tie a ClusterRole with a RoleBinding that lives within a namespace. This enables administrators to use a common set of roles for the entire cluster and then bind them to a specific namespace for use.




# Role Based Access – In Action

Let's see some of this in action. In this section we'll create a service account (**Subject**) that will have full permissions (**Verbs**) on all objects (**Resource**) in a single namespace.

We can assume that multiple teams are using our k8s cluster and we'll separate them by namespaces so they can't access each others pods in a namespace that is not their own. I've already created a namespace named "hollowteam" to set our permissions on.

```
Erics-MacBook-Pro-2:vsphere-tf eshanks$ kubectl get namespace
NAME              STATUS    AGE
default           Active    14d
heptio-sonobuoy   Active    13d
hollowteam        Active    1d
ingress           Active    14d
kube-public       Active    14d
kube-system       Active    14d
```

Lets first start by creating our service account or Subject. Here is a manifest file that can be deployed for our cluster to create the user. Theres not too much to this manifest, but a ServiceAccount with name "hollowteam-user" and the namespace it belongs to.

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: hollowteam-user
  namespace: hollowteam
```

Code language: PHP (php)

Next item is the Role which will tie our user "hollowteam-user" to the verbs. In the below manifest, we need to give the role a name and attach it to a namespace. Below this, we need to add the rules. The rules are going to specify the resource and the verb to tie together. You can see we're tying resources of "*" [wildcard for all] with a verb.

```
---
kind: Role
```

```
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: hollowteam-full-access
  namespace: hollowteam
rules:
- apiGroups: ["", "extensions", "apps"]
  resources: ["*"]
  verbs: ["*"]
```
Code language: JavaScript (javascript)

Then finally we tie the Role to the Service Account through a RoleBinding. We give the RoleBinding a name and assign the namespace. After this we list the Subjects. You could have more than one subject, but in this case we have one, and it is the Service Account we created earlier (hollowteam-user) and the namespace its in. Then the last piece is the roleRef which is the reference to the Role we created earlier which was named "hollowteam-full-access".

```
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: hollowteam-role-binding
  namespace: hollowteam
subjects:
- kind: ServiceAccount
  name: hollowteam-user
  namespace: hollowteam
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: hollowteam-full-access
```

At this point I assume that you have deployed all these manifest files with the command below:

```
kubectl apply -f [manifestfile.yml]
```
Code language: CSS (css)

At this point things should be ready to go, but you're likely still logging in with administrator privileges through the admin.conf KUBECONFIG file. We need to start logging in with different credentials; the hollowteam-user credentials. To do this we'll create a new KUBECONFIG file and use it as part of our connection information.

To do this for our hollowteam-user, we need to run some commands to generate the details for our KUBECONFIG file. To start, we need to get the ServiceAccount token for our hollowteam-user. To do this run:

```
kubectl describe sa [user] -n [namespace]
```
Code language: CSS (css)

Now we can see the details from the example user I created earlier named "hollowteam-user". Note down the service account token which in my case is **hollowteam-user-token-b25n4**.

```
Erics-MacBook-Pro-2:~ eshanks$ kubectl describe sa hollowteam-user -n h
ollowteam
Name:         hollowteam-user
Namespace:    hollowteam
Labels:       <none>
Annotations:  kubectl.kubernetes.io/last-applied-configuration={"apiVer
sion":"v1","kind":"ServiceAccount","metadata":{"annotations":{},"name":
"hollowteam-user","namespace":"hollowteam"}}

Image pull secrets:  <none>
Mountable secrets:   hollowteam-user-token-b25n4
Tokens:              hollowteam-user-token-b25n4
Events:              <none>
```

Next up, we need to grab the client token. To do this we need to get the secret used for our hollowteam-user in base64. To do this run:

```
kubectl get secret [user token] -n [namespace] -o "jsonpath={.data.token}" | base64 -D
```

Code language: JavaScript (javascript)

The output of running this command on my user token is shown below. Note down this client token for use later.

```
Erics-MacBook-Pro-2:~ eshanks$ kubectl get secret hollowteam-user-token
-b25n4 -n hollowteam -o "jsonpath={.data.token}" | base64 -D
eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2
NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJob2xsb
3d0ZWFtIiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZWNyZXQubmFtZSI6Imhv
bGxvd3RlYW0tdXNlci10b2tlbi1iMjVuNCIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY29
1bnQvc2VydmljZS1hY2NvdW50Lm5hbWUiOiJob2xsb3d0ZWFtLXVzZXIiLCJrdWJlcm5ldG
VzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC51aWQiOiJhYjUyYzc4Yi03N
WU0LTExZTktYTQ1Ny0wMDUwNTY5MzQyNTAiLCJzdWIiOiJzeXN0ZW06c2VydmljZWFjY291
bnQ6aG9sbG93dGVhbTpob2xsb3d0ZWFtLXVzZXIifQ.vpqfaqRq1DI55lRcB6YwSn7sqdMs
pHHPEI7wWQ2XmVB8SqXiF8OW0e1lXc169Z1RcjTQUhSZfWuORm_pGXZBRuh6r1vS7tCxZR-
MiCAM144A9_a6I1Q8F2WfAE5bT1q0YvfKMUiWaHLtewWSZG6pCK_USCWAvFP4tgCa5h83WU
_Br-fYKt4n7JT2CglC5qnIk8RPxrY7Kj13NthUkKHyVdsCt43zbh82zg8tJqX6yCqcglLKX
NxSRkYVKxREF-PLmA0S2lc4UE_GWLlDM5r69_ZyRTN3-qUIqV4k3EJTqdNMPt13SzZ1kguX
-hT6NkadZ2VSXG3aKeasC6TVE1T53QErics-MacBook-Pro-2:~ eshanks$
```

The last thing we need to gather is the Client Certificate info. To do this, we'll use our user token again and run the following command:

```
kubectl get secret [user token] -n [namespace] -o "jsonpath={.data['ca\.crt']}"
```

Code language: JavaScript (javascript)

The details of my certificate are shown below. Again, copy down this output for use in our KUBECONFIG file.

```
Erics-MacBook-Pro-2:~ eshanks$ kubectl get secret hollowteam-user-token
-b25n4 -n hollowteam -o "jsonpath={.data['ca\.crt']}"
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUN5RENDQWJDZ0F3SUJBZ0lCQURBTkJ
na3Foa2lHOXcwQkFRc0ZBREFWTVJNd0VRWURWUVFERXdwcmRXSmwKY201bGRHVnpNQjRYRF
RFNU1EVXdNVEF5TURJeE5Wb1hEVEk1TURReU9EQXlNREl4TlZvd0ZURVRNQkVHQTFVRQpBe
E1LYTNWaVpYSnVaWFJsY3pDQ0FTSXdEUVlKS29aSWh2Y05BUUVCQlFBRGdnRVBBRENDQVFv
Q2dnRUJBTXZzCnN0alYyQjJ2YW5pa1BGZDErMVNKaEpUR1lJUkRXR0xjcGhValg4cnZ5cnd
McE1zdGlRZFFCK2prTjdZWmk4QmYKVHRVeWp4RXZYUnlmbndGWUhDK3huTUFPaTREbzR5MH
poemdibkxyY2NlVUJoV1pwcnBPeXdUOE1aaktaUI5LwpYL1M4bUFHdU8xRVphOS9kNjhUS
XVHelkxZzBlQWUvOG93Rkx4MDBPNkY3dUd1RmwrNitpdEgxdzlUNnczbjFZCnpZbzdoMzlD
eElDZGd1YjFMNTV4cWlTbVAyYnpJT2UybmsyclRxOGlKR3VveEM3Q01qaUxzQWFqNjRwemF
HNWEKVHNXeXdqY3d0aWgxZGxTTzhjR3oveUNrdHRwTHJqZHhLbGdjRjRCL1pOb0NQQ3FRbU
9iYmlid2dvUEZpZnpQdgpVY2lGWWNZUm1PT0FuRmYxRUNFQ0F3RUFBYU1qTUNFd0RnWURWU
jBQQVFIL0JBUURBZ0trTUE4R0ExVWRFd0VCCi93UUZNQU1CQWY4d0RRWUpLb1pJaHZjTkFR
RUxCUUFEZ2dFQkFJcjEvUDljRFFjNDRPaHdkUjQzeG51VnlFM0cKL2FwbTdyTzgrSW54WGc
1ZkkzSHdDTjNXSGowYXZYS0VkeHhnWFd5bjNkMVNIR0Y2cnZpYWVvMFNEMjNoUTZkcQp4SW
9JaDVudi9WSUZ4OWdXQ1hLMnVGV3RqclUvUTI0aHZVcnRzSTVWV1Uwc0EzWjdoNDVVUUhDa
25RVkN4N3NMCjZQYmJiY3F0dm1aQzk0TEhnS0VCUEtPMWpGWjRHcEF6d0ZxSStmWUQ2aHF3
dk5kWC9PQVpDRUtLejJSOFZHT2MKS2N4b2k4cVRYV0NYL0x4OU1JSEFEcG1wUjFqT3p1Q3F
EQ0RLc0VJdEhidUtWRHdHZzFlMFZPY3Q2Y0Fsc2dsawp5NkNpYW45aE9oYnVBTm00eHZGbT
FpK2tBeFNRTkczb0x1d3VrU0NMMHkyNlYyT3FZcFMzdGFYVGkwdz0KLS0tLS1FTkQgQ0VSV
ElGSUNBVEUtLS0tLQo=Erics-MacBook-Pro-2:~ eshanks$
```

Next, we can take the data we've gathered and throw it in a new config.conf file. Use the following as a template and place your details in the file.

```
apiVersion: v1
kind: Config
preferences: {}
users:
- name: [user here]
  user:
    token: [client token here]
clusters:
- cluster:
    certificate-authority-data: [client certificate here]
    server: https://[ip or dns entry here for cluster:6443
  name: [cluster name.local]
contexts:
- context:
    cluster: [cluster name.local]
    user: [user here]
    namespace: [namespace]
  name: [context name]
current-context: [set context]
```

Code language: JavaScript (javascript)

The final product from my config is shown below for reference. (Don't worry, you won't be able to use this to access my cluster.)

```
apiVersion: v1
```

```
kind: Config
preferences: {}
users:
- name: hollowteam-user
  user:
    token:
eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY
2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJob2x
sb3d0ZWFtIiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZWNyZXQubmFtZSI6I
mhvbGxvd3RlYW0tdXNlci10b2tlbi1iMjVuNCIsImt1YmVybmV0ZXMuaW8vc2VydmljZWF
jY291bnQvc2VydmljZS1hY2NvdW50Lm5hbWUiOiJob2xsb3d0ZWFtLXVzZXIiLCJrdWJlc
m5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC51aWQiOiJhYjUyYzc
4Yi03NWU0LTExZTktYTQ1Ny0wMDUwNTY5MzQyNTAiLCJzdWIiOiJzeXN0ZW06c2VydmljZ
WFjY291bnQ6aG9sbG93dGVhbTpob2xsb3d0ZWFtLXVzZXIifQ.vpqfaqRq1DI55lRcB6Yw
Sn7sqdMspHHPEI7wWQ2XmVB8SqXiF8OW0e1lXc169Z1RcjTQUhSZfWuORm_pGXZBRuh6r1
vS7tCxZR-
MiCAM144A9_a6I1Q8F2WfAE5bT1q0YvfKMUiWaHLtewWSZG6pCK_USCWAvFP4tgCa5h83W
U_Br-
fYKt4n7JT2CglC5qnIk8RPxrY7Kj13NthUkKHyVdsCt43zbh82zg8tJqX6yCqcglLKXNxS
RkYVKxREF-PLmA0S2lc4UE_GWLlDM5r69_ZyRTN3-qUIqV4k3EJTqdNMPt13SzZ1kguX-
hT6NkadZ2VSXG3aKeasC6TVE1T53Q
clusters:
- cluster:
    certificate-authority-data:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUN5RENDQWJDZ0F3SUJBZ0lCQURBTk
Jna3Foa2lHOXcwQkFRc0ZBREFWTVJNd0VRWURWUVFERXdwcmRXSmwKY201bGRHVnpNQjRY
RFRFNU1EVXdNVEF5TURJeE5Wb1hEVEk1TURReU9EQXlNREl4TlZvd0ZURVRNQkVHQTFVRQ
pBeE1LYTNWaVpYSnVaWFJsY3pDQ0FTSXdEUVlKS29aSWh2Y05BUUVCQlFBRGdnRVBBREND
QVFvQ2dnRUJBTXZzCnN0alYyQjJ2YW5pa1BGZDErMVNKaEpUR1lJUkRXR0xjcGhValg4cn
Z5cndMcE1zdGlRZFFCK2prTjdZWmk4QmYKVHRVeWp4RXZYUnlmbndGWUhDK3huTUFPaTRE
bzR5MHpoemdibkxyY2NlVUJoV1pwcnBPeXdUOE1aakltaUI5LwpYL1M4bUFHdU8xRVphOS
9kNjhUSXVHelkxZzBlQWUvOG93Rkx4MDBPNkY3dUd1RmwrNitpdEgxdzlUNnczbjFZCnpZ
bzdoMzlDeElDZGd1YjFMNTV4cWlTbVAyYnpJT2UybmsyclRxOGlKR3VveEM3Q01qaUxzQW
FqNjRwemFHNWEKVHNXeXdqY3d0aWgxZGxTTzhjR3oveUNrdHRwTHJqZHhLbGdjRjRCL1pO
b0NQQ3FRbU9iYmlid2dvUEZpZnpQdgpVY2lGWWNZUm1PT0FuRmYxRUNFQ0F3RUFBYU1qTU
NFd0RnWURWUjBQQVFIL0JBUURBZ0trTUE4R0ExVWRFd0VCCi93UUZNQU1CQWY4d0RRWUpL
b1pJaHZjTkFRRUxCUUFEZ2dFQkFJcjEvUDljRFFjNDRPaHdkUjQzeG51VnlFM0cKL2FwbT
dyTzgrSW54WGc1ZkkzSHdDTjNXSGowYXZYS0VkeHhnWFd5bjNkMVNIR0Y2cnZpYWVvMFNE
MjNoUTZkcQp4SW9JaDVudi9WSUZ4OWdXQ1hLMnVGV3RqclUvUTI0aHZVcnRzSTVWV1Uwc0
EzWjdoNDVVUUhDa25RVkN4N3NMCjZQYmJiY3F0dm1aQzk0TEhnS0VCUEtPMWpGWjRHcEF6
d0ZxSStmWUQ2aHF3dk5kWC9PQVpDRUtLejJSOFZHT2MKS2N4b2k4cVRYV0NYL0x4OU1JSE
FEcG1wUjFqT3p1Q3FEQ0RLc0VJdEhidUtWRHdHZzFlMFZPY3Q2Y0Fsc2dsawp5NkNpYW45
aE9oYnVBTm00eHZGbTFpK2tBeFNRTkczb0x1d3VrU0NMMHkyNlYyT3FZcFMzdGFYVGkwdz
0KLS0tLS1FTkQgQ0VSVElGSUNBVEUtLS0tLQo=
    server: https://10.10.50.160:6443
  name: hollowcluster.local
```

```
contexts:
- context:
    cluster: hollowcluster.local
    user: hollowteam-user
    namespace: hollowteam
  name: hollowteam
current-context: hollowteam
```
Code language: PHP (php)

Save this file and then update your KUBECONFIG info which is usually done by updating your KUBECONFIG environment variable. On my Mac laptop I used:

```
export KUBECONFIG=~/.kube/newfile.conf
```
Code language: JavaScript (javascript)

Now its time to test. If we did everything right, the logged in entity should only have access to the hollowteam namespace. Lets first test to make sure that we can access our hollowteam namespaces resources by doing a get pods command.

```
Erics-MacBook-Pro-2:~ eshanks$ export KUBECONFIG=~/.kube/hollowteam.conf
Erics-MacBook-Pro-2:~ eshanks$ kubectl get pods
NAME         READY     STATUS    RESTARTS   AGE
shell-demo   1/1       Running   0          1d
```

The first command works and shows that we can access the pod (that I pre-created for this demo) and everything looks fine there. What about if we try a command that affects things outside of the namespace?

```
Erics-MacBook-Pro-2:~ eshanks$ kubectl get pods --all-namespaces
Error from server (Forbidden): pods is forbidden: User "system:serviceaccount:hollow
team:hollowteam-user" cannot list resource "pods" in API group "" at the cluster sco
pe
```

Here we see that we get an error message when we try to access pods that are outside of our namespace. Again, this was the desired result.

**Summary**

Role Based Access controls are part of the basics for most platforms and Kubernetes is no different. Its useful to carefully plan out your roles so that they can be reused for multiple use cases but I'll leave this part up to you. I hope the details in this post will help you to segment your existing cluster by permissions, but don't forget that you can always create more clusters for separation purposes as well. Good luck in your container endeavors.

# Kubernetes – Pod Backups

The focus of this post is on pod based backups, but this could also go for Deployments, replica sets, etc. This is not a post about how to backup your Kubernetes cluster including things like

etcd, but rather the resources that have been deployed on the cluster. Pods have been used as an example to walk through how we can take backups of our applications once deployed in a Kubernetes cluster.

# Pod Backups – The Theory

Typically, when I'm talking about backups, the "theory" part is almost self-explanatory. If you deploy a production workload, you must ensure that it's also backed up in case there is a problem with the application. However, with Kubernetes, you can certainly make the argument that a backup is not necessary.

No, this doesn't mean that you can finally stop worrying about backups altogether, but Kubernetes resources are usually designed to handle failure. If a pod dies, it can be restarted using replica-sets etc. Also, it's possible that a human makes an error and a deployment gets removed or something. Again, this is probably easily fixed by re-deploying the pods through a Kubernetes manifest file stored in version control for safe keeping.

What about state though? Sometimes our pods contain state data for our application. This is really where a backup seems most appropriate to me in a Kubernetes cluster because that state data can't be restored through redeploying your pods from code. Users may have logged into your app, created a profile, etc and this is important data that needs to be restored if the app is to be considered production. My personal views on this (yours may vary) is that if you have stateful data, perhaps that data should be stored outside of your Kubernetes cluster. For example if your app writes to a database, maybe a services such as Amazon RDS might be a better place to store that data than a pod with persistent volumes. I'll let you make this decision.

It is also possible to use backups for reasons other than disaster recovery. Maybe we've got a bunch of pods deployed but are migrating them to a new cluster. Backup/restore processes might be an excellent way for migrating those apps.

# Pod Backups – In Action

For the demo section of this post, we're going to use a tool called "Velero" (formerly named Ark) which is open-sourced software provided by Heptio. The tool will backup our pod(s) to object storage, in this case an S3 bucket in AWS, and then restore this pod to our Kubernetes cluster once disaster strikes.

### Setup Velero

Before we can do anything, we need to install the velero client on our workstation. Much like we installed the kubectl client, we need a client for velero as well. Download the binaries and place them in your system path.

Once the client has been installed, we need to run the setup, which does things like deploy a new namespace named velero and a pod. We'll also configure our authentication to AWS so we can use a pre-created S3 bucket for storing our backups.

The code below shows the command run from my workstation to setup Velero in my lab. You'll notice I've set a provider to AWS, entered the name of an S3 bucket created, specified my aws credentials file with access to the s3 bucket and then specified the AWS region to be used.

```
velero install \
    --provider aws \
    --bucket theithollowvelero \
    --secret-file ~/.aws/credentials \
    --backup-location-config region=us-east-1
```

Code language: JavaScript (javascript)

After running this command, we can checkout whats happened in our kubernetes cluster by checking out the velero namespace. You can see there is a deployment created in that namespace now thanks to the velero install command.

```
Erics-MacBook-Pro-2:velero eshanks$ kubectl get pods -n=velero
NAME                      READY     STATUS    RESTARTS   AGE
velero-db6459bb-zxc9r     1/1       Running   0          1m
```

OK, velero should be ready to go now. Before we can demonstrate it we'll need a test pod to restore. I've deployed a simple nginx pod in a new namespace called "velerotesting" that we'll use to backup and restore. You can see the details below.

```
Erics-MacBook-Pro-2:velero eshanks$ kubectl get pods -n=velerotesting
NAME                                READY     STATUS    RESTARTS   AGE
nginx-deployment-68775fc68c-69hgh   1/1       Running   0          2m
```

**Backup Pod**

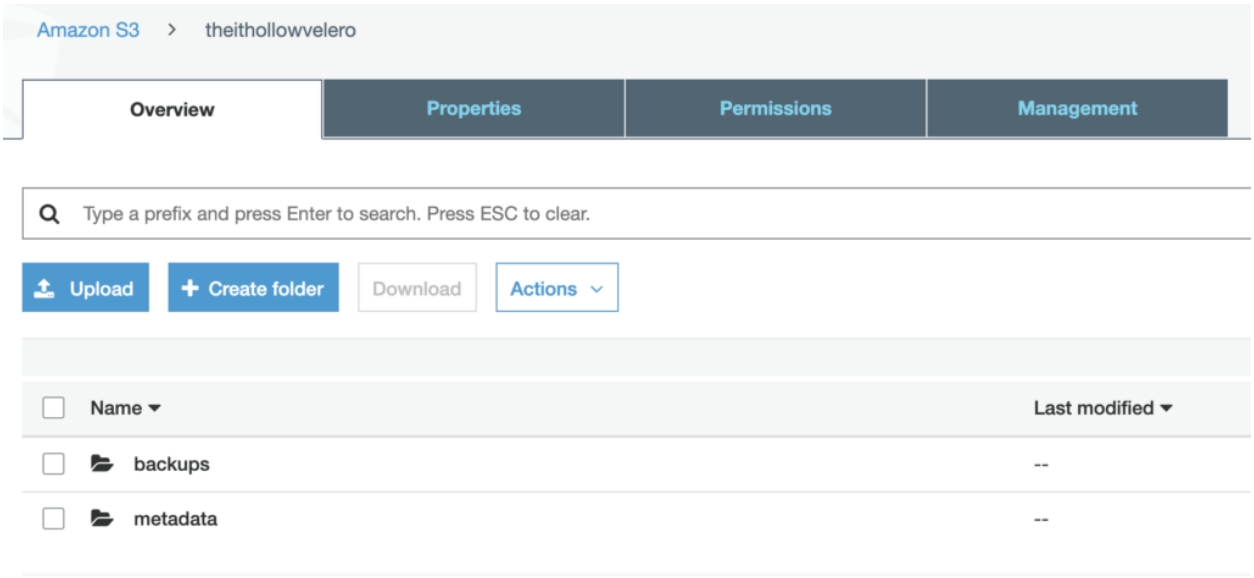
Time to run a backup! This is really simple to do with Velero. To run a one-time backup of our pod, we can run:

```
velero backup create [backup job name] --include-namespaces [namespace to backup]
```

Code language: CSS (css)

```
Erics-MacBook-Pro-2:velero eshanks$ velero backup create nginx-backup --include-namespaces velerotesting
Backup request "nginx-backup" submitted successfully.
Run `velero backup describe nginx-backup` or `velero backup logs nginx-backup` for more details.
```

That's it! Our backup is being created and if I take a look at the S3 bucket where our backups are stored, I now see some data in this bucket.



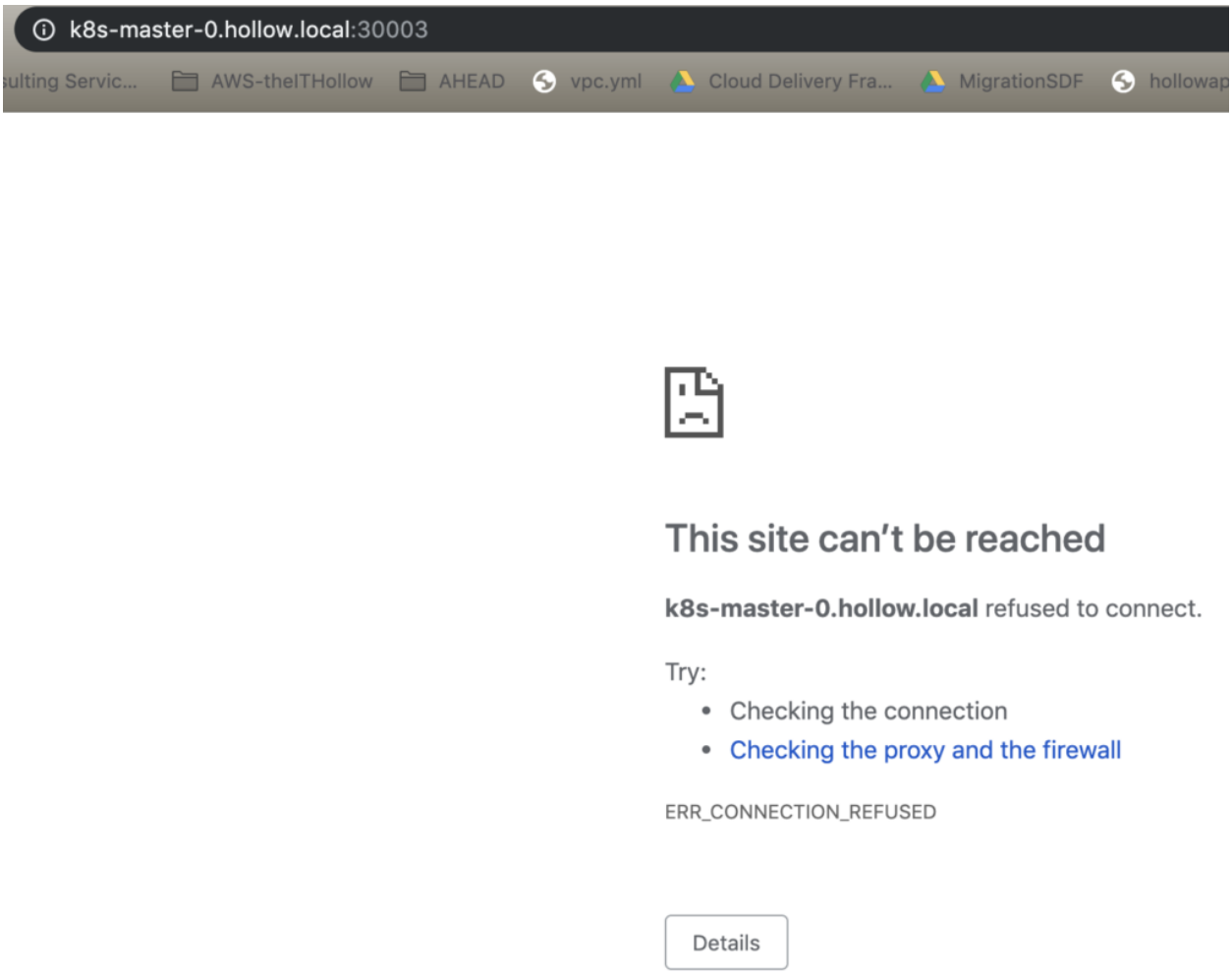

I wouldn't be doing Velero enough justice to leave it at this though. Velero can also run backups on a schedule and instead of backing up an entire namespace, backups can be created based on labels of your pods as well. To find more information about all the different capabilities of Velero, please see their official documentation. This post is focusing on a 1 time backup of an entire namespace.

**Restore Pod**

Everything seems fine, so let's change that for a second. I'll delete the namespace (and the included pods) from the cluster.

```
Erics-MacBook-Pro-2:velero eshanks$ kubectl delete namespace velerotesting
namespace "velerotesting" deleted
```

Here is our nginx container that is no longer working in our cluster.




Oh NO! We need to quickly restore our nginx container ASAP. Declare an emergency and prep the bridge call!

To do a restore of our pods and namespace, we'll run:

```
velero restore create --from-backup [backup name]
```

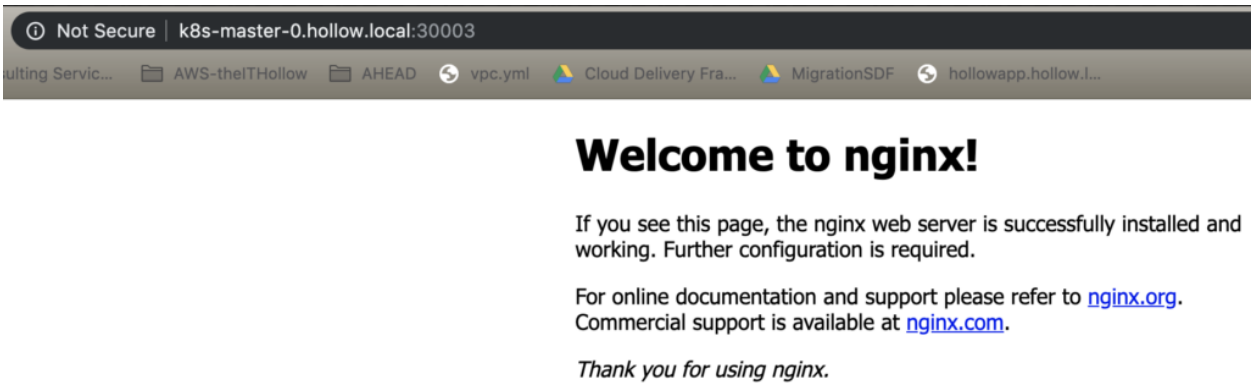
Code language: JavaScript (javascript)

```
Erics-MacBook-Pro-2:velero eshanks$ velero restore create --from-backup nginx-backup
Restore request "nginx-backup-20190602093749" submitted successfully.
Run `velero restore describe nginx-backup-20190602093749` or `velero restore logs ng
inx-backup-20190602093749` for more details.
```

Just in case you can't remember the name of your backup, you can also use velero to show your backups via the get command if you need to review your backups.

```
Erics-MacBook-Pro-2:velero eshanks$ velero backup get
NAME           STATUS      CREATED                         EXPIRES
N   SELECTOR
nginx-backup   Completed   2019-06-02 08:37:24 -0500 CDT   29d
```

Let's check on the restore now. Here's our nginx site.




**Summary**

I know this seems like a silly example since our nginx pod is really basic, but it should give you an idea of what can be done with Velero to backup your workloads. A quick note, that if you are using a snapshot provider to check and see if its supported. https://velero.io/docs/v1.0.0/support-matrix/

If you plan to use your Kubernetes cluster in production, you should plan to have a way to protect your pods whether through version control of stateless applications or with backups and tools like Velero.

# Kubernetes – Helm

*By* ERIC SHANKS

The Kubernetes series has now ventured into some non-native k8s discussions. Helm is a relatively common tool used in the industry and it makes sense to talk about why that is. This post covers the basics of Helm so we can make our own evaluations about its use in our Kubernetes environment.

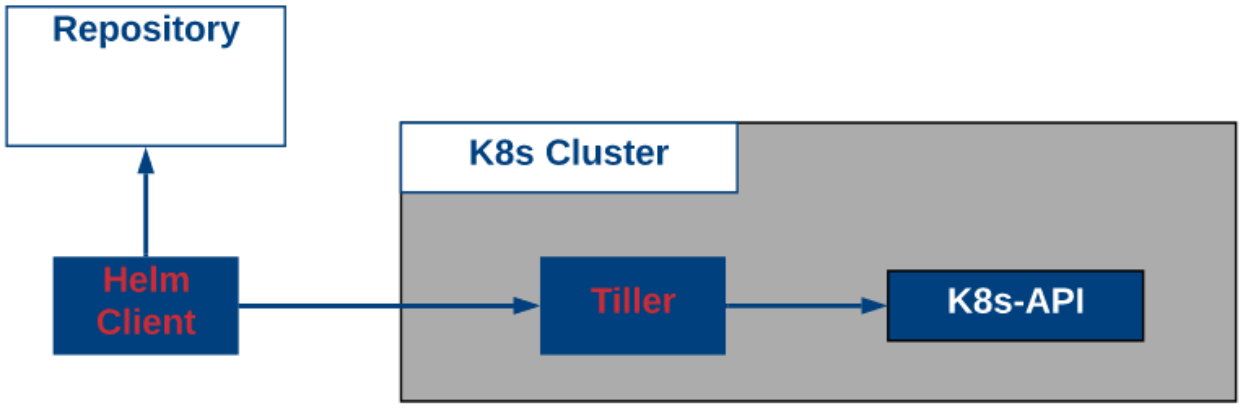
## Helm – The Theory

So what is Helm? In the most simplest terms its a package manager for Kubernetes. Think of Helm this way, Helm is to Kubernetes as yum/apt is to Linux. Yeah, sounds pretty neat now doesn't it?

Just like yum and apt, Helm is configured with a public repository where shared software packages are maintained for users to quickly access and install. For Helm, this is called the "stable" repo and it's managed by Google. You can of course add additional repositories if you're managing your own Helm charts.

Helm (as of version 2) uses a Kubernetes pod named "Tiller" as a server that interacts with the K8s api. Helm also has its own client which as you might guess interacts with Tiller to

take actions on the Kubernetes cluster. The example below shows the basic setup and interactions with Kubernetes.




Helm uses charts to define the details about software packages such as default values or variables, dependencies, templates, readmes and licenses, some of which are optional. We won't go into much detail about this in an introductory post, but the file structure for creating Helm charts is shown below.

```
/chartname
   Chart.yaml #file containing information about the chart
   requirements.yaml #optional file used for listing dependencies
   values.yaml #default parameter values for the chart
   charts/ #any dependent charts would be stored here
   templates/ #templates here are combined with values to generate k8s
manifests
   LICENSE #optional
   README.md #optional
```

Code language: PHP (php)

The big piece to understanding what Helm is doing is in the templates. When you combine the Helm templates with Helm values it creates the valid k8s manifests for the Kubernetes API to deploy. This gives us quite a bit of flexibility to do things like calculate values dynamically before deploying a static manifest to the Kubernetes cluster.

OK, Helm sounds pretty neat, why isn't it always used? Well, there are a couple of drawbacks at the moment. The first of which is around Tiller, which needs to be able to act as a Kubernetes administrator via RBAC. For Tiller to deploy our Deployments, Secrets, Services, etc., it will need access to all of those components. When a user uses the helm client to interact with Tiller, we've basically give our clients Admin access to the cluster which is a problem. Helm v3 (when released) aims to fix this RBAC by removing tiller from the k8s cluster. There are also some things to think about such as whether or not you really want to be able to use dynamic variables in your YAML or not.

## Helm – In Action

For the lab portion of this post, we'll install the Helm components and deploy a package from a repo. If you like the idea of Helm, you could continue your learning by writing your own Helm charts, but its outside the scope of this post.

To get started, we need to deploy Tiller to our Kubernetes cluster. To do this, we'll start by deploying a manifest like we've done many times before. This manifest sets up a service account and role for Tiller to perform actions against the cluster with appropriate (well, maybe too much) permissions.

Copy the manifest code below to a yml file and apply it against your k8s cluster as we did in the rest of this series.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tiller
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: tiller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
    name: tiller
    namespace: kube-system
```

```
Erics-MacBook-Pro-2:git eshanks$ kubectl apply -f tiller-config.yml
serviceaccount "tiller" created
clusterrolebinding.rbac.authorization.k8s.io "tiller" created
```

Next up, we'll need to install the Helm client on our workstation. I am using a Mac so I used homebrew to do this, but it can be done for other operating systems as well. Please see the official documentation if you aren't using homebrew on Mac. https://helm.sh/docs/using_helm/#installing-helm

I'm running the command:

```
brew install kubernetes-helm
```

```
Erics-MacBook-Pro-2:velero eshanks$ brew install kubernetes-helm
==> Downloading https://homebrew.bintray.com/bottles/kubernetes-helm-2.14.0.mojave.b
==> Downloading from https://akamai.bintray.com/4c/4cfcdb6c35a4ecc67b5ecb3b15a03b1c4
######################################################################## 100.0%
==> Pouring kubernetes-helm-2.14.0.mojave.bottle.tar.gz
==> Caveats
Bash completion has been installed to:
  /usr/local/etc/bash_completion.d

zsh completions have been installed to:
  /usr/local/share/zsh/site-functions
==> Summary
🍺  /usr/local/Cellar/kubernetes-helm/2.14.0: 51 files, 91.6MB
```

Now that the Helm client is installed, we can use it do initialize Helm which deploys Tiller for us, using the service account we created through the previously mentioned manifest file. To do this, run:

```
helm init --service-account [service account name]
```

Code language: CSS (css)

```
Erics-MacBook-Pro-2:git eshanks$ helm init --service-account tiller
Creating /Users/eshanks/.helm
Creating /Users/eshanks/.helm/repository
Creating /Users/eshanks/.helm/repository/cache
Creating /Users/eshanks/.helm/repository/local
Creating /Users/eshanks/.helm/plugins
Creating /Users/eshanks/.helm/starters
Creating /Users/eshanks/.helm/cache/archive
Creating /Users/eshanks/.helm/repository/repositories.yaml
Adding stable repo with URL: https://kubernetes-charts.storage.googleapis.com
Adding local repo with URL: http://127.0.0.1:8879/charts
$HELM_HOME has been configured at /Users/eshanks/.helm.

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

After the init process finishes, you're pretty much ready to use Helm to deploy packages. Lets take a look at some other Helm stuff first.

If we run "helm repo list" we can see a list of the available repositories where our packages could be stored. You'll notice taht out of the box, the "stable" repo is already configured for us.

```
Erics-MacBook-Pro-2:git eshanks$ helm repo list
NAME    URL
stable  https://kubernetes-charts.storage.googleapis.com
local   http://127.0.0.1:8879/charts
```

Its usually a good idea to update your repos, just like you might do with yum or apt so lets run a helm repo update before we try to apply anything.

```
Erics-MacBook-Pro-2:git eshanks$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Skip local chart repository
...Successfully got an update from the "stable" chart repository
Update Complete.
```

OK, now lets go find some software to deploy. We can use the search feature to look for software we might want to deploy. Below I've used search to find both Jenkins as well as mysql. You'll notice there are many versions of mysql but only one version of Jenkins at the moment.

```
Erics-MacBook-Pro-2:git eshanks$ helm search jenkins
NAME            CHART VERSION   APP VERSION     DESCRIPTION
stable/jenkins  1.1.23          lts             Open source continuous integration server. It supports mu...
Erics-MacBook-Pro-2:git eshanks$ helm search mysql
NAME                                    CHART VERSION   APP VERSION     DESCRIPTION

stable/mysql                            1.2.0           5.7.14          Fast, reliable, scalable, and easy to u
se open-source rel...
stable/mysqldump                        2.4.1           2.4.1           A Helm chart to help backup MySQL datab
ases using mysqldump
stable/prometheus-mysql-exporter        0.3.2           v0.11.0         A Helm chart for prometheus mysql expor
ter with cloudsqlp...
stable/percona                          1.1.0           5.7.17          free, fully compatible, enhanced, open
source drop-in rep...
stable/percona-xtradb-cluster           1.0.0           5.7.19          free, fully compatible, enhanced, open
source drop-in rep...
stable/phpmyadmin                       2.2.3           4.8.5           phpMyAdmin is an mysql administration f
rontend
stable/gcloud-sqlproxy                  0.6.1           1.11            DEPRECATED Google Cloud SQL Proxy

stable/mariadb                          6.4.0           10.3.15         Fast, reliable, scalable, and easy to u
se open-source rel...
```

Now as a deployment test, we'll try to deploy Jenkins through Helm. Here we'll run the helm install command and the name of the package + our common name that we'll use for the package. For example:

```
helm install [repo]/[package] --name [common name]
```

```
Erics-MacBook-Pro-2:git eshanks$ helm install stable/jenkins --name jenkins
NAME:   jenkins
LAST DEPLOYED: Tue Jun  4 14:23:56 2019
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/ConfigMap
NAME           DATA  AGE
jenkins        5     <invalid>
jenkins-tests  1     <invalid>

==> v1/Deployment
NAME     READY  UP-TO-DATE  AVAILABLE  AGE
jenkins  0/1    1           0          <invalid>

==> v1/PersistentVolumeClaim
NAME     STATUS   VOLUME         CAPACITY  ACCESS MODES  STORAGECLASS  AGE
jenkins  Pending  vsphere-disk   <invalid>
```

You'll notice from the screenshot above, that jenkins was deployed into the default namespace and there are a list of resources (the full list is truncated) that were deployed in our k8s cluster. You'll also notice that at the bottom of the return data there are some notes.

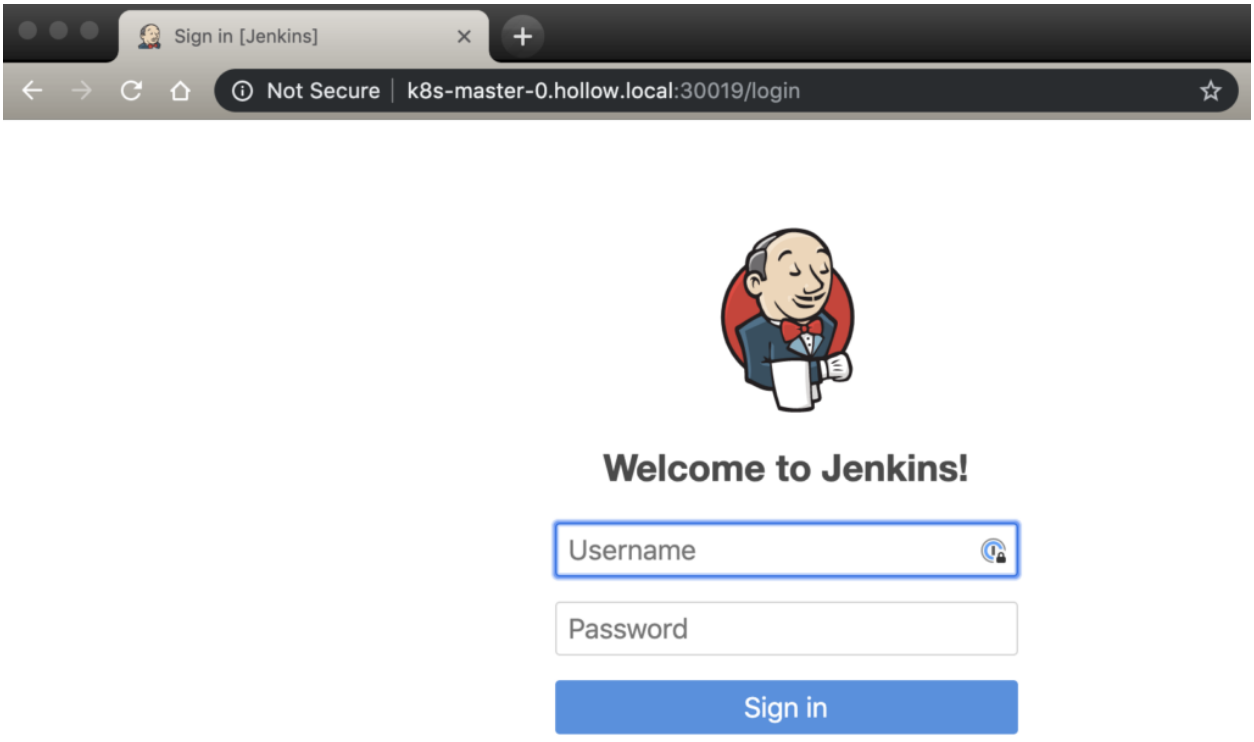
```
NOTES:
1. Get your 'admin' user password by running:
  printf $(kubectl get secret --namespace default jenkins -o jsonpath="{.data.jenkins-admin-password}" | base64
 --decode);echo
2. Get the Jenkins URL to visit by running these commands in the same shell:
  NOTE: It may take a few minutes for the LoadBalancer IP to be available.
        You can watch the status of by running 'kubectl get svc --namespace default -w jenkins'
  export SERVICE_IP=$(kubectl get svc --namespace default jenkins --template "{{ range (index .status.loadBalan
cer.ingress 0) }}{{ . }}{{ end }}")
  echo http://$SERVICE_IP:8080/login

3. Login with the password from step 1 and the username: admin


For more information on running Jenkins on Kubernetes, visit:
https://cloud.google.com/solutions/jenkins-on-container-engine
```

These notes are pretty helpful in getting stated with the package we just deployed. For example in this case it shows how to start using Jenkins and the URL to access.

If we were to use the kubectl client, we can see our new pod and service in our cluster.

```
Erics-MacBook-Pro-2:git eshanks$ kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
hollowapp-799474dbd9-kwk4h    1/1       Running   0          21d
hollowdb-7c4bbd94b7-sj8f9     1/1       Running   0          21d
jenkins-6c9588974-pf75l       1/1       Running   0          3m
```

And even better, we can see that we can access the Jenkins service and the application.




**Summary**

Helm, may be a pretty great way to deploy common packages from vendors just like apt/yum have been for linux. Its also a great tool if you want to dynamically update parameters for your Kubernetes manifests at deployment time without re-writing your static manifest files for each environment. Play around with Helm and see what you think, and watch out for Helm v3 if permissions are a concern of yours.

# Kubernetes – Taints and Tolerations

*By* ERIC SHANKS

One of the best things about Kubernetes, is that I don't have to think about which piece of hardware my container will run on when I deploy it. The Kubernetes scheduler can make that decision for me. This is great until I actually DO care about what node my container runs on. This post will examine one solution to pod placement, through taints and tolerations.

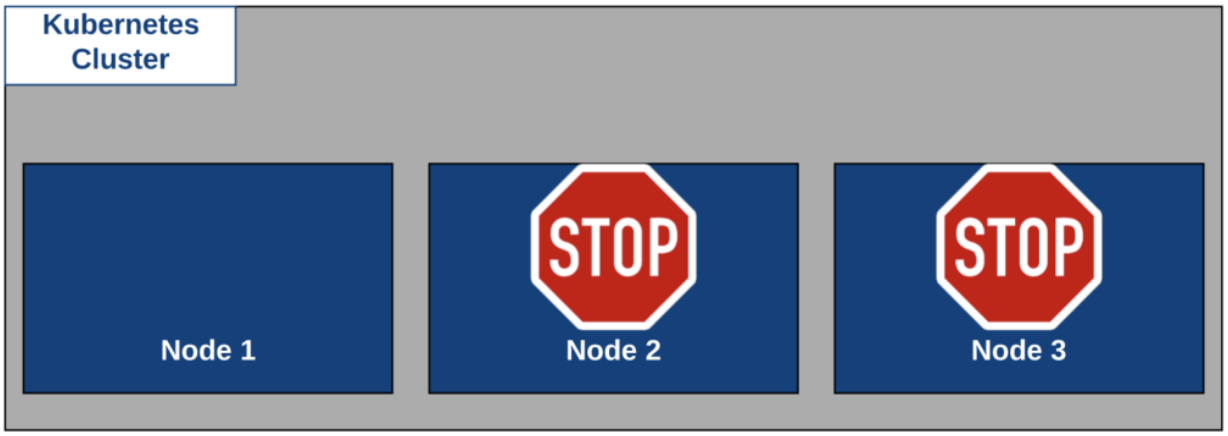
## Taints – The Theory

Suppose we had a Kubernetes cluster where we didn't want any pods to run on a specific node. You might need to do this for a variety of reasons, such as:

- one node in the cluster is reserved for special purposes because it has specialized hardware like a GPU
- one node in the cluster isn't licensed for some software running on it
- one node is in a different network zone for compliance reasons
- one node is in timeout for doing something naughty

Whatever the particular reason, we need a way to ensure our pods are not placed on a certain node. That's where a taint comes in.

Taint's are a way to put up a giant stop sign in front of the K8s scheduler. You can apply a taint to a k8s node to tell the scheduler you're not available for any pods.



# Tolerations – The Theory

How about use case where we had really slow spinning disks in a node. We applied a taint to that node so that our normal pods won't be placed on that piece of hardware, due to it's poor performance, but we have some pods that don't need fast disks. This is where Tolerations could come into play.

A toleration is a way of ignoring a taint during scheduling. Tolerations aren't applied to nodes, but rather the pods. So, in the example above, if we apply a toleration to the PodSpec, we could "tolerate" the slow disks on that node and still use it.

# Taints – In Action

Let's apply a taint to our Kubernetes cluster. But first, you might check to see if you have a taint applied already. Depending upon how you deployed your cluster, your master node(s) might have a taint applied to them to keep pods from running on the master nodes. You can run the:

```
kubectl describe node [k8s master node]
```

Code language: CSS (css)

```
Erics-MacBook-Pro-2:~ eshanks$ kubectl describe nodes k8s-master-0
Name:               k8s-master-0
Roles:              master
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=k8s-master-0
                    kubernetes.io/os=linux
                    node-role.kubernetes.io/master=
Annotations:        kubeadm.alpha.kubernetes.io/cri-socket: /var/run/dockershim.sock
                    node.alpha.kubernetes.io/ttl: 0
                    projectcalico.org/IPv4Address: 10.10.50.171/24
                    volumes.kubernetes.io/controller-managed-attach-detach: true
CreationTimestamp:  Fri, 05 Jul 2019 14:23:45 -0500
Taints:             node-role.kubernetes.io/master:NoSchedule
Unschedulable:      false
```

OK, now lets apply a taint to a couple of nodes in our cluster. I'll create a taint with a key/value pair of "hardware:slow" to identify nodes that should not run my pods any longer because of their slow hardware specifications.

```
kubectl taint nodes [node name] [key=value]:NoSchedule
```

Code language: CSS (css)

In my case I ran this twice because I tainted two nodes. I should mention that this can be done through labels as well to quickly taint multiple nodes. Also, we ran the command with the "NoSchedule" effect which keeps the scheduler from choosing this node, but you could also use other effects like "PreferNoSchedule" or "NoExecute" as well.

```
Erics-MacBook-Pro-2:~ eshanks$ kubectl taint nodes k8s-worker-1 hardware=slow:NoSchedule
node/k8s-worker-1 tainted
Erics-MacBook-Pro-2:~ eshanks$ kubectl taint nodes k8s-worker-2 hardware=slow:NoSchedule
node/k8s-worker-2 tainted
```

At this point, two of my three available worker nodes are tainted with the "hardware" key pair. Lets deploy some pods and see how they're scheduled. I'll deploy nginx pods to my workers and I'll deploy three pods which ordinarily we'd expect to be deployed evenly across my cluster. The manifest file below is what will be deployed.

```
apiVersion: apps/v1 #version of the API to use
kind: Deployment #What kind of object we're deploying
metadata: #information about our object we're deploying
  name: nginx-deployment #Name of the deployment
  labels: #A tag on the deployments created
    app: nginx
spec: #specifications for our object
  strategy:
    type: RollingUpdate
    rollingUpdate: #Update Pods a certain number at a time
      maxUnavailable: 1 #Total number of pods that can be unavailable at
once
      maxSurge: 1 #Maximum number of pods that can be deployed above
desired state
  replicas: 3 #The number of pods that should always be running
  selector: #which pods the replica set should be responsible for
    matchLabels:
      app: nginx #any pods with labels matching this I'm responsible
for.
  template: #The pod template that gets deployed
    metadata:
      labels: #A tag on the replica sets created
        app: nginx
    spec:
      containers:
      - name: nginx-container #the name of the container within the pod
        image: nginx:1.7.9 #which container image should be pulled
        ports:
        - containerPort: 80 #the port of the container within the pod
```

Code language: PHP (php)

After applying the nginx deployment, we'll check our pods and see which nodes they are running on. To do this run:

```
kubectl get pod -o=custom-
columns=NODE:.spec.nodeName,NAME:.metadata.name
```

Code language: JavaScript (javascript)

```
Erics-MacBook-Pro-2:taints eshanks$ kubectl get pod -o=custom-columns=NODE:.spec.nodeName,NAME:.metadata.name
NODE           NAME
k8s-worker-0   nginx-deployment-588dfc5c45-dtg28
k8s-worker-0   nginx-deployment-588dfc5c45-kg992
k8s-worker-0   nginx-deployment-588dfc5c45-ptmtm
```

As you can see, I've got three pods deployed and they're all on k8s-worker-0. This is the only node that wasn't tainted in my cluster, so this confirms that the taints on k8s-worker-1 and k8s-worker-2 are working correctly.

# Tolerations – In Action

Now I'm going to delete that deployment and deploy a new deployment that tolerates our "hardware" taint.

I've created a new manifest file that is the same as we ran before, except this time I added a toleration for the taint we applied to our nodes.

```
apiVersion: apps/v1 #version of the API to use
kind: Deployment #What kind of object we're deploying
metadata: #information about our object we're deploying
  name: nginx-deployment #Name of the deployment
  labels: #A tag on the deployments created
    app: nginx
spec: #specifications for our object
  strategy:
    type: RollingUpdate
    rollingUpdate: #Update Pods a certain number at a time
      maxUnavailable: 1 #Total number of pods that can be unavailable at
once
      maxSurge: 1 #Maximum number of pods that can be deployed above
desired state
  replicas: 3 #The number of pods that should always be running
  selector: #which pods the replica set should be responsible for
    matchLabels:
      app: nginx #any pods with labels matching this I'm responsible
for.
  template: #The pod template that gets deployed
    metadata:
      labels: #A tag on the replica sets created
        app: nginx
    spec:
      tolerations:
      - key: "hardware"
        operator: "Equal"
        value: "slow"
        effect: "NoSchedule"
      containers:
```

```
      - name: nginx-container #the name of the container within the pod
        image: nginx:1.7.9 #which container image should be pulled
        ports:
        - containerPort: 80 #the port of the container within the pod
```
Code language: PHP (php)

Lets apply this new manifest to our cluster and see what happens to the pod placement decisions by the scheduler.

```
Erics-MacBook-Pro-2:taints eshanks$ kubectl get pod -o=custom-columns=NODE:.spec.nodeName,NAME:.metadata.name
NODE           NAME
k8s-worker-2   nginx-deployment-7776c9848c-82x2l
k8s-worker-1   nginx-deployment-7776c9848c-gvrcs
k8s-worker-0   nginx-deployment-7776c9848c-pddkn
```

Well, look there. Now those same three pods were distributed across the three nodes evenly for this deployment that tolerated the node taints.

**Summary**

Its hard to say what needs you might have for scheduling pods on specific nodes in your cluster, but by using taints and tolerations you can adjust where these pods are deployed.

Taints are applied at the node level and prevent nodes from being used. Tolerations are applied at the pod level and can tell the scheduler which taints they are able to withstand.

# Kubernetes – DaemonSets

DaemonSets can be a really useful tool for managing the health and operation of the pods within a Kubernetes cluster. In this post we'll explore a use case for a DaemonSet, why we need them, and an example in the lab.

## DaemonSets – The Theory

DaemonSets are actually pretty easy to explain. A DaemonSet is a Kubernetes construct that ensures a pod is running on every node (where eligible) in a cluster. This means that if we were to create a DaemonSet on our six node cluster (3 master, 3 workers), the DaemonSet would schedule the defined pods on each of the nodes for a total of six pods. Now, this assumes there are either no taints on the nodes, or there are tolerations on the DaemonSets.

In reality, DaemonSets behave very similarly to a Kubernetes deployment, with the exception that they will be automatically distributed to ensure the pods are deployed on each node in the cluster. Also, if a new node is deployed to your cluster after the DaemonSet has been deployed, the new node will also get the DaemonSet deployed by the scheduler, after the join occurs.

So why would we use a DaemonSet? Well, a common use for DaemonSets would be for logging. Perhaps we need to ensure that our log collection service is deployed on each node in our cluster to collect the logs from that particular node. This might be a good use case for a DaemonSet. Think of this another way; we could run and install services on each of our Kubernetes nodes by installing the app on the OS. But now that we've got a container orchestrator available to use, lets take advantage of that and build those tools into a container and automatically schedule them on all nodes within the cluster, in one fell swoop.

# DaemonSets – In Practice

For this example, we'll deploy a simple container as a DaemonSet to show that they are distributed on each node. While this example, is just deploying a basic container, it could be deploying critical services like a log collector. The manifest below contains the simple Daemonset manifest.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: daemonset-example
  labels:
    app: daemonset-example
spec:
  selector:
    matchLabels:
      app: daemonset-example
  template:
    metadata:
      labels:
        app: daemonset-example
    spec:
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: busybox
        image: busybox
        args:
        - sleep
        - "10000"
```

Code language: JavaScript (javascript)

Note that the DaemonSet has a toleration so that we can deploy this container on our master nodes as well, which are tainted.

We apply the manifest with the:

```
kubectl apply -f [file name].yaml
```

Code language: CSS (css)

After we apply it, lets take a look at the results by running:

```
kubectl get pods --selector=app=daemonset-example -o wide
```

Code language: JavaScript (javascript)

```
NAME                          READY   STATUS    RESTARTS   AGE   IP             NODE
daemonset-example-bcnw4       1/1     Running   0          30s   172.16.3.11    k8s-worker-1
daemonset-example-jqfxx       1/1     Running   0          30s   172.16.2.11    k8s-master-2
daemonset-example-k6cs9       1/1     Running   0          30s   172.16.0.10    k8s-master-0
daemonset-example-k6pv7       1/1     Running   0          30s   172.16.1.11    k8s-master-1
daemonset-example-p868c       1/1     Running   0          30s   172.16.4.13    k8s-worker-0
daemonset-example-xfpxn       1/1     Running   0          30s   172.16.5.13    k8s-worker-2
```

You can see from the screenshot above, there are six pods deployed. Three on master nodes and three more on worker nodes. Great, so we didn't have to use any affinity rules to do this, the DaemonSet made sure we had one pod per node as we had hoped.

What happens if we add another worker nodes to the cluster? Well, let's try it.

I've used kubeadm to join another node to my cluster. Shortly after the join completed, I noticed the DaemonSets started deploying another pod without my intervention.

```
NAME                          READY   STATUS              RESTARTS   AGE   IP             NODE
daemonset-example-bcnw4       1/1     Running             0          29m   172.16.3.11    k8s-worker-1
daemonset-example-jqfxx       1/1     Running             0          29m   172.16.2.11    k8s-master-2
daemonset-example-k6cs9       1/1     Running             0          29m   172.16.0.10    k8s-master-0
daemonset-example-k6pv7       1/1     Running             0          29m   172.16.1.11    k8s-master-1
daemonset-example-mf8vj       0/1     ContainerCreating   0          21s   <none>         k8s-worker-3
daemonset-example-p868c       1/1     Running             0          29m   172.16.4.13    k8s-worker-0
daemonset-example-xfpxn       1/1     Running             0          29m   172.16.5.13    k8s-worker-2
```

And, as you'd hoped, a minute later the container was up and running on my new node "k8s-worker-3".

```
NAME                          READY   STATUS    RESTARTS   AGE   IP             NODE
daemonset-example-bcnw4       1/1     Running   0          29m   172.16.3.11    k8s-worker-1
daemonset-example-jqfxx       1/1     Running   0          29m   172.16.2.11    k8s-master-2
daemonset-example-k6cs9       1/1     Running   0          29m   172.16.0.10    k8s-master-0
daemonset-example-k6pv7       1/1     Running   0          29m   172.16.1.11    k8s-master-1
daemonset-example-mf8vj       1/1     Running   0          45s   172.16.6.2     k8s-worker-3
daemonset-example-p868c       1/1     Running   0          29m   172.16.4.13    k8s-worker-0
daemonset-example-xfpxn       1/1     Running   0          29m   172.16.5.13    k8s-worker-2
```

**Summary**

DaemonSets can make pretty short work of deploying resources throughout your k8s nodes. You'll see many applications use DaemonSets to ensure that each node of your cluster is being properly managed.

# Kubernetes – Network Policies

In the traditional server world, we've taken great lengths to ensure that we can micro-segment our servers instead of relying on a few select firewalls at strategically defined chokepoints. What do we do in the container world though? This is where network policies come into play.

# Network Policies – The Theory

In a default deployment of a Kubernetes cluster, all of the pods deployed on the nodes can communicate with each other. Some security folks might not like to hear that, but never fear, we have ways to limit the communications between pods and they're called network policies.

I still see it in many places today where companies have a perimeter firewall protecting their critical IT infrastructure, but if an attacker were to get through that perimeter, they'd have pretty much free reign to connect to whatever system they wanted. We introduced micro-segmentation with technologies like VMware NSX to make a whole bunch of smaller zones even to the point of a zone for each virtual machine NIC. This added security didn't come easily, but it is a significant improvement over a few perimeter firewalls protecting everything.

Now before we go forward with this post, I should say that a Kubernetes Network Policy is not a firewall, but it does restrict the traffic between pods so that they can't all communicate with each other.

We've defined our applications as code already, so why not code in some network security as well by only allowing certain pods to communicate with our pods. For instance if you have a three tiered application, maybe you'd set policies so your web pods could only communicate with the app pods, and the app pods communicate with the database pods, but not the web directly to the database. This is a more sound security design since we're removing attack vectors in our pods.

Now, setting up a network policy is still done in the standard Kubernetes methodology of applying manifests in a yaml format. If you apply the correct network policies to the Kubernetes API, the network plugin will apply the proper rules for you to restrict the traffic. Not all network plugins can do this however so ensure you're using the proper network plugin such as Calico. NOTE: flannel is not capable of applying network policies as of the time of this writing.

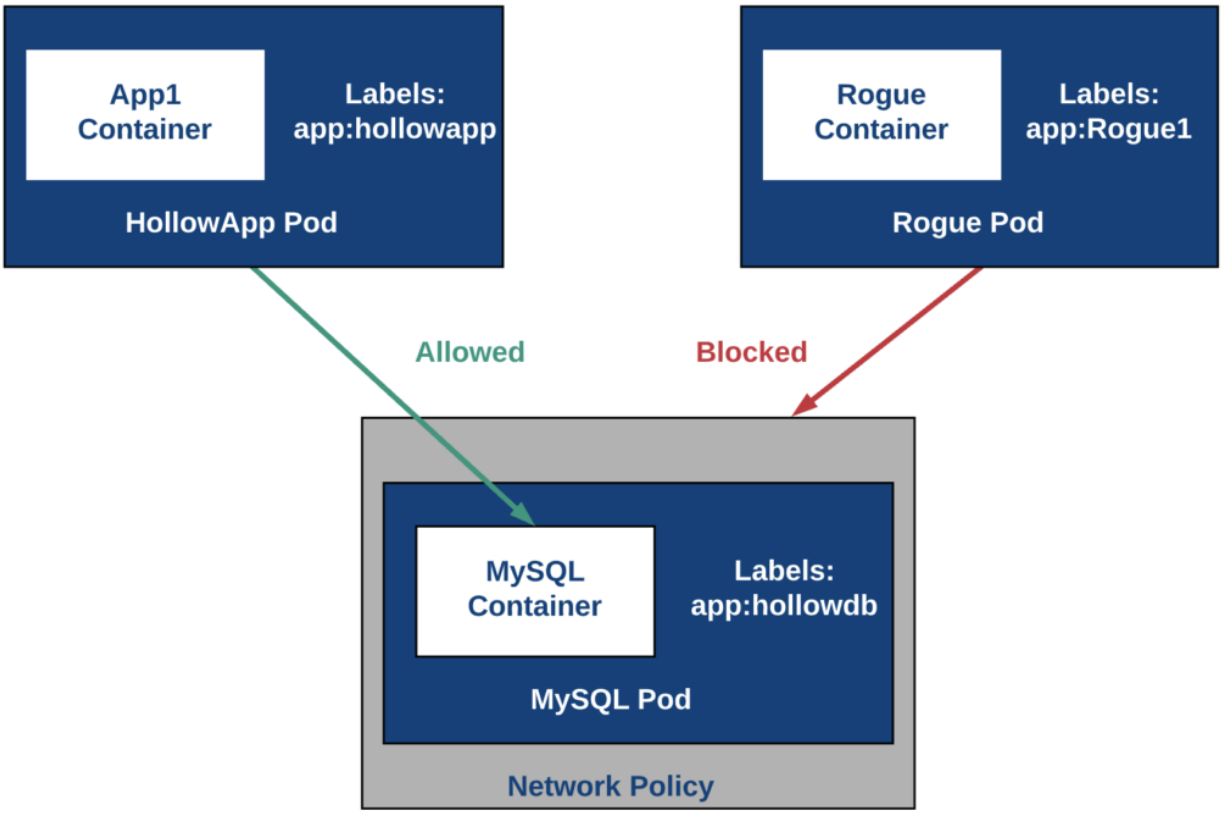
Network Policies can restrict, Ingress or Egress or both if you need. You might want to consider a strategy to keep your rules straight though or you could have a hard time troubleshooting later on. I prefer to only restrict ingress traffic if I can swing it, just to limit the complexity.

# Network Policies – In Action

In this section we'll apply a network policy to limit access to a backend MySQL database pod. In this example, I have an app pod that requires access to a backend MySQL database, but I

don't want some other "rogue" pod to get deployed and have access to that database as well. It COULD have super secret information in it that I need to protect.

Below is a picture of what we'll be testing.




Below is a Network Policy that will allow ingress traffic to a pod with label app:hollowdb from any pods with a label of app:hollowapp over port 3306.

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: hollow-db-policy   #policy name
spec:
  podSelector:
    matchLabels:
      app: hollowdb  #pod applied to
  policyTypes:
  - Ingress #Ingress and/or Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: hollowapp #pod allowed
```

```
    ports:
    - protocol: TCP
      port: 3306 #port allowed
```
Code language: PHP (php)

Once the manifest was applied with:

```
kubectl apply -f [manifest file].yml
```
Code language: CSS (css)

And our database has been deployed of course, then we can test out the policy. To do that we'll deploy a mysql container with an allowed label to ensure we can connect to the backend database over 3306. The interactive command below will deploy the container and get us a mysql prompt.

```
kubectl run hollowapp --image=mysql -it --labels="app=hollowapp" -- bash
```
Code language: JavaScript (javascript)

From there we'll try to connect to the hollowdb container with our super secret root password. As you can see we can login and show the databases in the container. So the network policy is allowing traffic as intended.

```
root@hollowapp-54797cf59d-7kh6p:/# mysql -h hollowdb -uroot -pPassword123
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 65
Server version: 5.7.25 MySQL Community Server (GPL)

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| hollowapp          |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.01 sec)
```
Working MySQL Connection from App

Now, lets deploy our example rouge container the same with but use a label that is not "hollowapp" as seen below.

```
kubectl run rogue1 --image=mysql -it --labels="app=rogue1" -- bash
```

```
Code language: JavaScript (javascript)
```
This time, when we get our mysql prompt we are unable to connect to the MySQL database.

```
root@rogue1-795fcb9974-gbpmn:/# mysql -h hollowdb -uroot -pPassword123
mysql: [Warning] Using a password on the command line interface can be insecure.
ERROR 2003 (HY000): Can't connect to MySQL server on 'hollowdb' (110)
```

Connection Blocked from Rogue App

**Summary**

Kubernetes clusters allow traffic between all pods by default, but if you've got a network plugin capable of using Network Policies, then you can start locking down that traffic. Build your Network Policy manifests and start including them with your application manifests.

# Kubernetes – Pod Security Policies

Securing and hardening our Kubernetes clusters is a must do activity. We need to remember that containers are still just processes running on the host machines. Sometimes these processes can get more privileges on the Kubernetes node than they should, if you don't properly setup some pod security. This post explains how this could be done for your own clusters.

## Pod Security Policies – The Theory

Pod Security policies are designed to limit what can be run on a Kubernetes cluster. Typical things that you might want to limit are: pods that have privileged access, pods with access to the host network, and pods that have access to the host processes just to name a few. Remember that a container isn't as isolated as a VM so we should take care to ensure our containers aren't adversely affecting our nodes's health and security.

Pod Security Policies (PSP) are an optional admission controller added to a cluster. These admission controllers are an added check that determines if a pod should be admitted to the cluster. This is an additional check that comes after both authentication and authorization have been checked for the api call. A pod security policy uses the admission controller to check and see if our pod meets our extra level of scrutiny before being added to our cluster.

## Pod Security Policies – In Action

To demonstrate how Pod Security Policies work, we'll create a policy that blocks pods from having privileged access to the host operating system. Now, the first thing we'll do here is to enable the admission controller in our API server specification.

NOTE: enabling the admission controller does not have to be the first step. In fact, once you enable the admission controller on the API server, no pods will be able to be deployed (or redeployed) because there will not be a policy that matches. By default, EVERY pod deployment will be blocked. It may be a good idea to apply your PSPs first so that it doesn't interrupt operations.

To demonstrate, we'll first create a pod specification that does not have any privileged [escalated] access to our cluster. The container specification of the manifest below has the "allowPrivilegeEscalation: false". If you deploy this before enabling the admission controller. Everything should work fine. Save this file as not-escalated.yaml.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: not-escalated
  labels:
    app: not-escalated
spec:
  replicas: 1
  selector:
    matchLabels:
      app: not-escalated
  template:
    metadata:
      labels:
        app: not-escalated
    spec:
      containers:
      - name: not-escalated
        image: busybox
        command: ["sleep", "3600"]
        securityContext:
          allowPrivilegeEscalation: false
```

Code language: JavaScript (javascript)

Now apply the manifest to your cluster with the command:

```
kubectl create -f not-escalated.yaml
```

Code language: CSS (css)

Check to see if your pods got deployed by running:

```
kubectl get pods
```

Code language: JavaScript (javascript)

```
eshanks-a01:securitypolicy eshanks$ kubectl create -f not-escalated.yaml
deployment.apps/not-escalated created
eshanks-a01:securitypolicy eshanks$ kubectl get pods
NAME                             READY   STATUS    RESTARTS   AGE
not-escalated-9b8c8454f-4txtx    1/1     Running   0          43s
```

At this point, all we've done is prove that a simple pod works on our cluster before we enable our admission controller. Hang on to this pod manifest, because we'll use it again later.

Now we'll enable the admission control plugin for PodSecurityPolicy. You can see the api-server flag that I used below. NOTE: applying this during your cluster bootstrapping may prevent even the cricital kube-system pods from starting up. In this case, I've applied this configuration setting after the initial bootstrap phase were my kube-system pods are ALREADY running.

```
- kube-apiserver
    -                                                --enable-admission-plugins=NodeRestriction,PodSecurityPolicy
```

Once the PodSecurityPolicy admission control plugin is enabled, we can try to apply the not-escalated.yaml manifest again. Once deployed, we can check our replica set and we should notice that the pod was not deployed. Remember that without an appropriate policy set, NO PODS will be able to be deployed.

In the screenshot below you can see that no pods were found after I deployed the same manifest file. To dig deeper, I check the replica set that should have created the pods. There I see a desired count of "1" but no pods deployed.

```
eshanks-a01:securitypolicy eshanks$ kubectl create -f not-escalated.yaml
deployment.apps/not-escalated created
eshanks-a01:securitypolicy eshanks$ kubectl get pods
No resources found in default namespace.
eshanks-a01:securitypolicy eshanks$ kubectl get replicaset
NAME                       DESIRED   CURRENT   READY   AGE
not-escalated-9b8c8454f    1         0         0       18s
```

The reason for this is that there is no PodSecurityPolicy that would allow a pod of ANY kind to be deployed. Let's fix that next, but delete that deployment for now.

OK, so now the next step is to create a psp that will let us deploy pods that don't require privilege escalation (like the one we just did) but not let us deploy a pod that does require escalation.

Right. Now lets apply our first pod security policy that allows pods to be deployed if they don't require any special access. Below is a PSP that allows non-privileged pods to be deployed.

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: default-restricted
spec:
  privileged: false
  hostNetwork: false
  allowPrivilegeEscalation: false #This is the main setting we're looking at in this blog post.
```

```
  defaultAllowPrivilegeEscalation: false
  hostPID: false
  hostIPC: false
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  volumes:
  - 'configMap'
  - 'downwardAPI'
  - 'emptyDir'
  - 'persistentVolumeClaim'
  - 'secret'
  - 'projected'
  allowedCapabilities:
  - '*'
```

Code language: PHP (php)

Deploy that PodSecurity Policy to your cluster with kubectl:

```
kubctl create -f [pspfilename].yaml
```

Code language: CSS (css)

```
eshanks-a01:securitypolicy eshanks$ kubectl get psp
NAME                PRIV    CAPS                        SELINUX    RUNASUSER   FSGROUP    SUPGROUP   READONLYROOTFS
VOLUMES
default-restricted   false   *                           RunAsAny   RunAsAny    RunAsAny   RunAsAny   false
configMap,downwardAPI,emptyDir,persistentVolumeClaim,secret,projected
speaker             true    NET_ADMIN,NET_RAW,SYS_ADMIN   RunAsAny   RunAsAny    RunAsAny   RunAsAny   false
*
```

You might think that's all we need to do, but it isn't. The next step is to give the replica-controller access to this policy through the use of a ClusterRole and ClusterRoleBinding.

First, the cluster role is create to link the "psp" resource to the "use" verb.

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: default-restrictedrole
rules:
- apiGroups:
  - extensions
  resources:
  - podsecuritypolicies
  resourceNames:
  - default-restricted
  verbs:
```

```
  - use
```
Code language: PHP (php)

Apply the ClusterRole by apply the:

```
kubectl create -f [clusterRoleManifest].yaml
```
Code language: CSS (css)

Next deploy the ClusterRoleBinding which links the previously created cluster role to the service accounts.

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: default-psp-rolebinding
subjects:
- kind: Group
  name: system:serviceaccounts
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: default-restrictedrole
  apiGroup: rbac.authorization.k8s.io
```
Code language: JavaScript (javascript)

Deploy the ClusterRoleBinding with the command:

```
kubectl create -f [clusterRoleBindingManifest].yaml
```
Code language: CSS (css)

```
eshanks-a01:securitypolicy eshanks$ kubectl create -f clusterRole.yaml
clusterrole.rbac.authorization.k8s.io/default-restrictedrole created
eshanks-a01:securitypolicy eshanks$ kubectl create -f clusterRoleBinding.yaml
clusterrolebinding.rbac.authorization.k8s.io/default-psp-rolebinding created
```

If everything has gone right, then we should have granted the replica-set controller the permissions needed to use the PodSecurityPolicy that allows our non-priviledged pods to be deployed. Time to test that by deploying that not-escalated.yaml manifest again.

```
eshanks-a01:securitypolicy eshanks$ kubectl create -f not-escalated.yaml
deployment.apps/not-escalated created
eshanks-a01:securitypolicy eshanks$ kubectl get replicaset
NAME                       DESIRED   CURRENT   READY   AGE
not-escalated-9b8c8454f    1         1         1       13s
eshanks-a01:securitypolicy eshanks$ kubectl get pods
NAME                             READY   STATUS    RESTARTS   AGE
not-escalated-9b8c8454f-8lpg7    1/1     Running   0          18s
```

Our pod was deployed correctly! Now we need to test one more thing. We'll change the option on that manifest file to AllowPriviliged access on our container. The manifest below has that flag flipped for you.

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: escalated
  labels:
    app: escalated
spec:
  replicas: 1
  selector:
    matchLabels:
      app: escalated
  template:
    metadata:
      labels:
        app: escalated
    spec:
      containers:
      - name: not-escalated
        image: busybox
        command: ["sleep", "3600"]
        securityContext:
          allowPrivilegeEscalation: true
```

Code language: JavaScript (javascript)

Now save that manifest as "escalated.yaml" and we'll apply it to our cluster and then check the replicasets and pods again. This container is the one we're trying to prevent from being deployed in our cluster.

```
kubectl create -f escalated.yaml
```

Code language: CSS (css)

You can see from the screenshot that the priviledged container could not run in our cluster with our PodSecurityPolicy.

```
eshanks-a01:securitypolicy eshanks$ kubectl create -f escalated.yaml
deployment.apps/escalated created
eshanks-a01:securitypolicy eshanks$ kubectl get replicasets
NAME                       DESIRED   CURRENT   READY   AGE
escalated-5c8fbfd466       1         0         0       13s
not-escalated-9b8c8454f    1         1         1       5m17s
```

Just a reminder, this policy on its own might not be good for a production environment. If you have any pods in your kube-system namespace for instance, that actually need privileged access, this policy will block those too. Keep that in mind even if everything seems to be working fine right now. Leaving this policy in place will also keep them from restarting if they fail, or the node is restarted. So be careful with them.

**Summary**

Pod Security Policies can be an important part to your cluster health since they can reduce unwanted actions being taken on your host. PSPs require an admission controller to be

enabled on the kube-api server. After which all pods will be denied until a PSP that allows them is created, and permissions given to the appropriate service account.

# Kubernetes Resource Requests and Limits

Containerizing applications and running them on Kubernetes doesn't mean we can forget all about resource utilization. Our thought process may have changed because we can much more easily scale-out our application as demand increases, but many times we need to consider how our containers might fight with each other for resources. Resource Requests and Limits can be used to help stop the "noisy neighbor" problem in a Kubernetes Cluster.

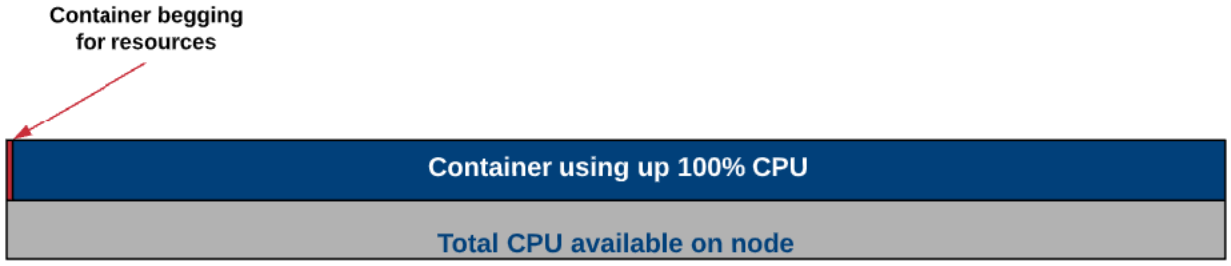
## Resource Requests and Limits – The Theory

Kubernetes uses the concept of a "Resource Request" and a "Resource Limit" when defining how many resources a container within a pod should receive. Lets look at each of these topics on their own, starting with resource requests.

### Resource Requests

To put things simply, a resource request specifies the minimum amount of resources a container needs to successfully run. Thought of in another way, this is a guarantee from Kubernetes that you'll always have this amount of either CPU or Memory allocated to the container.

Why would you worry about the mimimum amount of resources guaranteed to a pod? Well, its to help prevent one container from using up all the node's resources and starving the other containers from CPU or memory. For instance if I had two containers on a node, one container could request 100% of that nodes processor. Meanwhile the other container would likely not be working very well because the processor is being monopolized by its "noisy neighbor".




What a resource request can do, is to ensure that at least a small part of that processor's time is reserved for both containers. This way if there is resource contention, each pod will have a guaranteed, minimum amount of resource in which to still function.

### Resource Limits

As you might guess, a resource limit is the maximum amount of CPU or memory that can be used by a container. The limit represents the upper bounds of how much CPU or memory that a container within a pod can consume in a Kubernetes cluster, regardless of whether or not the cluster is under resource contention.

Limits prevent containers from taking up more resources on the cluster than you're willing to let them.

### Common Practices

As a general rule, all containers should have a request for memory and cpu before deploying to a cluster. This will ensure that if resources are running low, your container can still do the minimum amount of work to stay in a healthy state until those resource free up again (hopefully).

Limits are often used in conjunction with requests to create a "guaranteed pod". This is where the request and limit are set to the same value. In that situation, the container will always have the same amount of CPU available to it, no more or less.

At this point you may be thinking about adding a high "request" value to make sure you have plenty of resource available for your container. This might sound like a good idea, but have dramatic consequences to scheduling on the Kubernetes cluster. If you set a high CPU request, for example 2 CPUs, then your pod will ONLY be able to be scheduled on Kubernetes nodes that have 2 full CPUs available that aren't reserved by other pods' requests. In the example below, the 2 vCPU pod couldn't be scheduled on the cluster. However, if you were to lower the "request" amount to say 1 vCPU, it could.

## Resource Requests and Limits – In Action

### CPU Limit Example

Lets try out using a CPU limit on a pod and see what happens when we try to request more CPU than we're allowed to have. Before we set the limit though, lets look at a pod with a single container under normal conditions. I've deployed a resource consumer container in

my cluster and be default, you can see that I'm use 1m CPU(cores) and 6 Mi(bytes) of memory.

NOTE: CPU is measured in millicores so 1000m = 1 CPU core. Memory is measured in Megabytes.

```
* workload1-admin@workload1 eshanks eshanks-a01 ~/Desktop/temp $ k top pod
NAME                                CPU(cores)   MEMORY(bytes)
resource-consumer-5cd4646db6-sfzwr  1m           6Mi
```

Ok, now that we've seen the "no load" state, lets add some CPU load by making a request to the pod. Here, I've increased the CPU usage on the container to 400 millicores.

```
* workload1-admin@workload1 eshanks eshanks-a01 ~/Desktop/temp $ curl --data "millicores=400&durationSec=600" http://10.10.70.155:30001/ConsumeCPU
ConsumeCPU
400 millicores
600 durationSec
```

After the metrics start coming in, you can see that I've got roughly 400m used on the container as you'd expect to see.

```
master 1✎ * workload1-admin@workload1 eshanks eshanks-a01 ~/git/resource-consumer $ k top pod
NAME                                CPU(cores)   MEMORY(bytes)
resource-consumer-79dd64ccf6-trv8b  401m         12Mi
```

Now I've deleted the container and we'll edit the deployment manifest so that it has a limit on CPU.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: resource-consumer
  name: resource-consumer
  namespace: default
spec:
  progressDeadlineSeconds: 600
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      run: resource-consumer
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      labels:
        run: resource-consumer
    spec:
```

```
      containers:
      - image: theithollow/resource-consumer:v1
        imagePullPolicy: IfNotPresent
        name: resource-consumer
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        resources:
          requests:
            memory: "100Mi"
            cpu: "100m"
          limits:
            memory: "300Mi"
            cpu: "300m"
      dnsPolicy: ClusterFirst
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: {}
      terminationGracePeriodSeconds: 30
```

Code language: JavaScript (javascript)

In the container resources section I've set a limit on CPU to 300m. Lets re-deploy this yaml manifest and then again increase our resource usage to 400m.

After redeploying the container and again increasing my CPU load to 400m, we can see that the container is throttled to 300m instead. I've effectively "limited" the resources the container could consume from the cluster.

```
* workload1-admin@workload1 eshanks eshanks-a01 ~/Desktop/temp $ k top pod
NAME                                CPU(cores)   MEMORY(bytes)
resource-consumer-5cd4646db6-sfzwr   300m         16Mi
```

**CPU Requests Example**

OK, next, I've deployed two pods into my Kubernetes cluster and those pods are on the same worker node for a simple example about contention. I've got a guaranteed pod that has 1000m CPU set as a limit but also as a request. The other pod is unbounded, meaning there is no limit on how much CPU it can utilize.

After the deployment, each pod is really not using any resources as you can see here.

```
master 1✎ 1+ * workload1-admin@workload1 eshanks eshanks-a01 ~/git/resource-consumer $ k get pod -o wide
NAME                                READY   STATUS    RESTARTS   AGE     IP              NODE
guaranteed-pod-77d99f477f-b7z2t     1/1     Running   0          7m19s   100.103.197.9   workload1-md-0-7b866c9947-pvplg
resource-consumer-79dd64ccf6-6zm6m  1/1     Running   0          2m37s   100.103.197.10  workload1-md-0-7b866c9947-pvplg
master 1✎ 1+ * workload1-admin@workload1 eshanks eshanks-a01 ~/git/resource-consumer $ k top pod
NAME                                CPU(cores)   MEMORY(bytes)
guaranteed-pod-77d99f477f-b7z2t     0m           4Mi
resource-consumer-79dd64ccf6-6zm6m  0m           4Mi
```

I make a request to increase the load on my non-guaranteed pod.

```
master 1✎ ?1+ * workload1-admin@workload1 eshanks eshanks-a01 ~/git/resource-consumer $ curl --data "millicores=2000&durationSec=600" http://10.10.70.154:30001/ConsumeCPU
ConsumeCPU
2000 millicores
600 durationSec
```

And if we look at the containers resources you can see that even though my container wants to use 2000m CPU, it's only actually using 1000m CPU. The reason for this is because the guaranteed pod is guaranteed 1000m CPU, whether it is actively using that CPU or not.

```
master 1✎ ?1+ * workload1-admin@workload1 eshanks eshanks-a01 ~/git/resource-consumer $ k top pod
NAME                                CPU(cores)   MEMORY(bytes)
guaranteed-pod-77d99f477f-b7z2t     0m           4Mi
resource-consumer-79dd64ccf6-6zm6m  999m         11Mi
```

**Summary**

Kubernetes uses Resource Requests to set a minimum amount of resources for a given container so that it can be used if it needs it. You can also set a Resource Limit to set the maximum amount of resources a pod can utilize.

Taking these two concepts and using them together can ensure that your critical pods always have the resources that they need to stay healthy. They can also be configured to take advantage of shared resources within the cluster.

Be careful setting resource requests too high so your Kubernetes scheduler can still scheduler these pods. Good luck!

# Kubernetes Pod Auto-scaling

You've built your Kubernetes cluster(s). You've built your apps in containers. You've architected your services so that losing a single instance doesn't cause an outage. And you're ready for cloud scale. You deploy your application and are waiting to sit back and "profit."

When your application spins up and starts taking on load, you are able to change the number of replicas to handle the additional load, but what about the promises of cloud and scaling? Wouldn't it be better to deploy the application and let the platform scale the application automatically?

Luckily Kubernetes can do this for you as long as you've setup the correct guardrails to protect your cluster's health. Kubernetes uses the Horizontal Pod Autoscaler (HPA) to determine if pods need more or less replicas without a user intervening.

## Horizontal Pod Auto-scaling – The Theory

In a conformant Kubernetes cluster you have the option of using the Horizontal Pod Autoscaler to automatically scale your applications out or in based on a Kubernetes metric. Obviously this means that before we can scale an application, the metrics for that application

will have to be available. For this, we need to ensure that the Kubernetes metrics server is deployed and configured.

The Horizontal Pod Autoscaler is implemented as a control loop, checking on the metrics every fifteen seconds by default and then making decisions about whether to scale a deployment or replica-set.

The scaling algorithm determines how many pods should be configured based on the current and desired state values. The actual algorithm is shown below.

```
desiredReplicas = ceil[currentReplicas * ( currentMetricValue / desiredMetricValue )]
```

If you haven't heard of Horizontal Pod Autoscaler until now, you're probably thinking that this feature is awesome and you can't wait to get this in your cluster for better load balancing. I mean, isn't capacity planning for your app much easier when you say, "I'll just start small and scale under load." And you're right in some sense. But also take careful consideration if this makes sense in your production clusters. If you have dozens of apps all autoscaling, the total capacity of the cluster can get chewed up if you haven't put the right restrictions on pods. This is a great time to revisit requests and limits on your pods, as well as setting autoscale limits. Don't make your new autoscaling app, a giant "noisy neighbor" for all the other apps in the cluster.

# Horizontal Pod Auto-scaling – In Action

### Prerequisites

Before you can use the Horizontal Pod Autoscaler, you'll have to have a few things in place.

1. Healthy/Conformant Kubernetes Cluster
2. Kubernetes Metric Server in place and serving metrics
3. A stateless replica-set so that it can be scaled

### Example Application

To show how the HPA works we'll scale out a simple Flask app that I've used in several other posts. This web server will start with a single container and scale up/down based on load. To generate load, we'll use busybox with a wget loop to generate traffic to the web server.

First, to deploy our web server. I'm re-using the hollowapp flask image and setting a 100m CPU limit with the imperative command:

```
kubectl run hollowapp --image=theithollow/hollowapp:allin1-v2 --limits=cpu=100m
```

In this case, we'll assume that we want to add another replica to our replica-set/deployment anytime a pod reaches 20% CPU Utilization. This can be done by using the command:

```
kubectl autoscale deployment [deployment name] --cpu-percentage=20 --min=1 --max-10
```

In my lab, I've deployed the flask container deployment, and you can see that there is virtually no load on the pod when I run the `kubectl top pods` command.

```
* kubernetes-admin@kubernetes eshanks eshanks-a01 ~ $ k get pods -l run=hollowapp
NAME                          READY   STATUS    RESTARTS   AGE
hollowapp-788cfbb55f-489st    1/1     Running   0          16h
* kubernetes-admin@kubernetes eshanks eshanks-a01 ~ $ k top pods -l run=hollowapp
NAME                          CPU(cores)   MEMORY(bytes)
hollowapp-788cfbb55f-489st    1m           53Mi
```

Before we start scaling, we can look at the current HPA object by running:

```
kubectl get hpa
```

Code language: JavaScript (javascript)

```
* kubernetes-admin@kubernetes eshanks eshanks-a01 ~ $ kubectl get hpa
NAME        REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
hollowapp   Deployment/hollowapp     1%/20%    1         10        1          20h
```

In the above screenshot, you can see that there is an HPA object with a target of 20% and the current value is 1%.

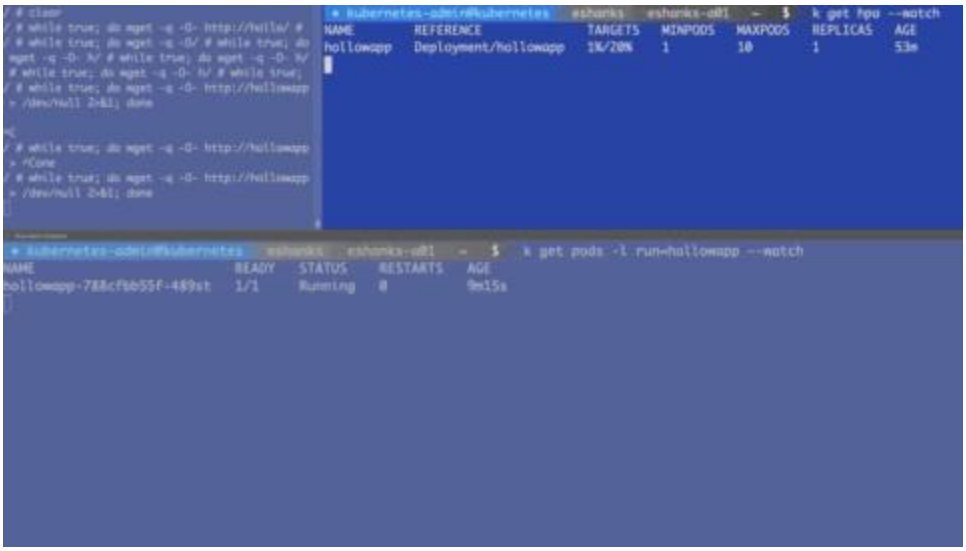
Now we're ready to scale the app. I've used busybox to generate load to this pod. As the load increases the CPU usage should also increase. When it hits 20% CPU utilization, the pods will scale out to try to keep this load under 20%.

In the gif below there are three screens. The top left consists of a script to generate load to my web containers. There isn't much useful info there so ignore it. If you want to generate your own load, deploy a container in interactive mode and run:

```
while true; do wget -q -O- http://[service name] > /dev/null 2>&1; done
```

Code language: JavaScript (javascript)

The upper right shows a `--watch` on the hpa object. Over time, you'll see that the HPA object updates with the current load. If the CPU load is greater than 20%, you'll then see HPA scale out the number of pods. This is displayed in the bottom shell window which is running a `--watch` on the pods.



After the number of pods hits 10, it no longer scales since that the maximum number of pods we allowed in the autoscale command. I should also note that after I stopped generating load, it took several minutes before my pods started to scale back down, but they will scale down again automatically.

**Summary**

The Kubernetes Horizontal Pod Autoscaler is a really nice feature to let you scale your app when under load. Its not always easy to know how many resources your app will need for a production environment, and scaling as you need it can be a nice fix for this. Also, to give back resources when you're not using them. But be careful not to let your scaling get out of control and use up all the resources in your cluster. Good luck!

# Kubernetes Liveness and Readiness Probes

Just because a container is in a running state, does not mean that the process running within that container is functional. We can use Kubernetes Readiness and Liveness probes to determine whether an application is ready to receive traffic or not.

## Liveness and Readiness Probes – The Theory

On each node of a Kubernetes cluster there is a Kubelet running which manages the pods on that particular node. Its responsible for getting images pulled down to the node, reporting the node's health, and restarting failed containers. But how does the Kubelet know if there is a failed container?

Well, it can use the notion of probes to check on the status of a container. Specifically a liveness probe.

Liveness probes indicate if a container is running. Meaning, has the application within the container started running and is it still running? If you've configured liveness probes for your containers, you've probably still seen them in action. When a container gets restarted, it's generally because of a liveness probe failing. This can happen if your container couldn't startup, or if the application within the container crashed. The Kubelet will restart the container because the liveness probe is failing in those circumstances. In some circumstances though, the application within the container is not working, but hasn't crashed. In that case, the container won't restart unless you provide additional information as a liveness probe.

A readiness probe indicates if the application running inside the container is "ready" to serve requests. As an example, assume you have an application that starts but needs to check on other services like a backend database before finishing its configuration. Or an application that needs to download some data before it's ready to handle requests. A readiness probe

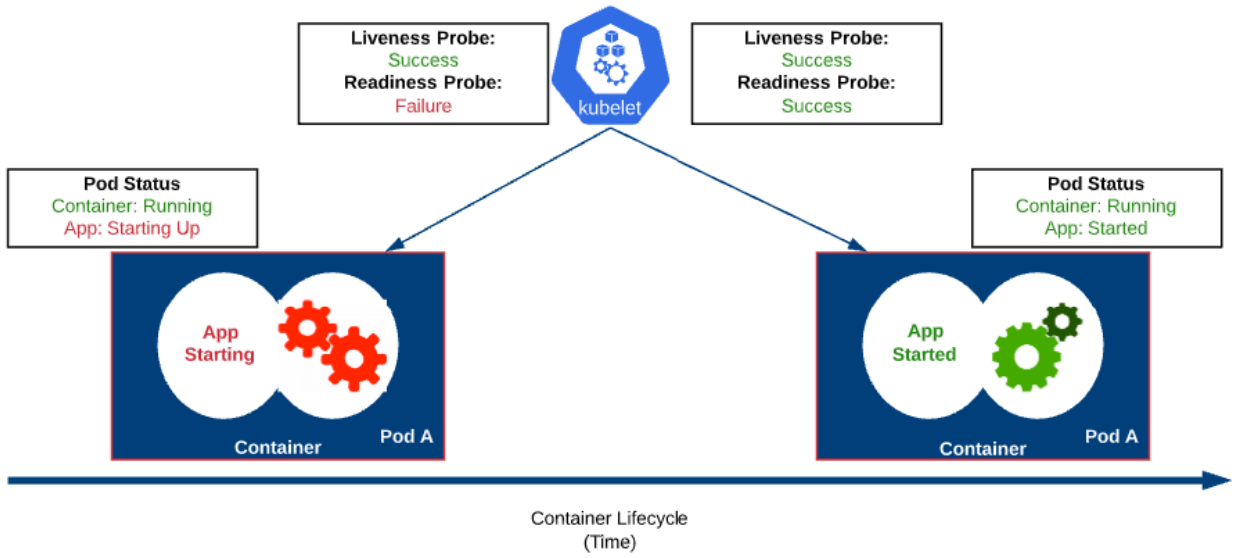
tells the Kubelet that the application can now perform its function and that the Kubelet can start sending it traffic.

There are three different ways these probes can be checked.

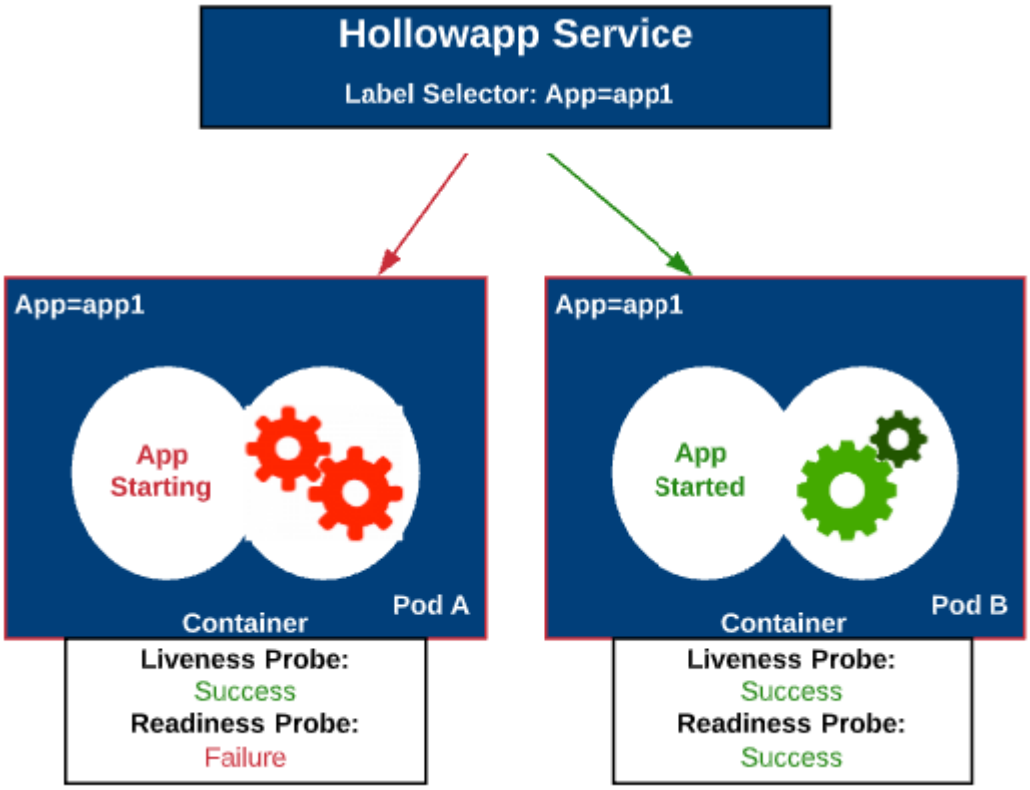
- ExecAction: Execute a command within the container
- TCPSocketAction: TCP check against the container's IP/port
- HTTPGetAction: An HTTP Get request against the container's IP/Port

Let's look at the two probes in the context of a container starting up. The diagram below shows several states of the same container over time. We have a view into the containers to see whats going on with the application with relationship to the probes.

On the left side, the pod has just been deployed. A liveness probe performed at TCPSocketAction and found that the pod is "alive" even though the application is still doing work (loading data, etc) and isn't ready yet. As time moves on, the application finishes its startup routine and is now "ready" to serve incoming traffic.




Let's take a look at this from a different perspective. Assume we have a deployment already in our cluster, and it consists of a single replica which is displayed on the right side, behind our service. Its likely that we'll need to scale the app, or replace it with another version. Now that we know our app isn't ready to handle traffic right away after being started, we can wait to have our service add the new app to the list of endpoints until the application is "ready". This is an important thing to consider if your apps aren't ready as soon as the container starts up. A request could be sent to the container before its able to handle the request.




## Liveness and Readiness Probes – In Action

First, we'll look to see what happens with a readiness check. For this example, I've got a very simple apache container that displays pretty elaborate website. I've created a yaml manifest to deploy the container, service, and ingress rule.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: theithollow/hollowapp-blog:liveness
    livenessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 3
      periodSeconds: 3
    readinessProbe:
      httpGet:
```

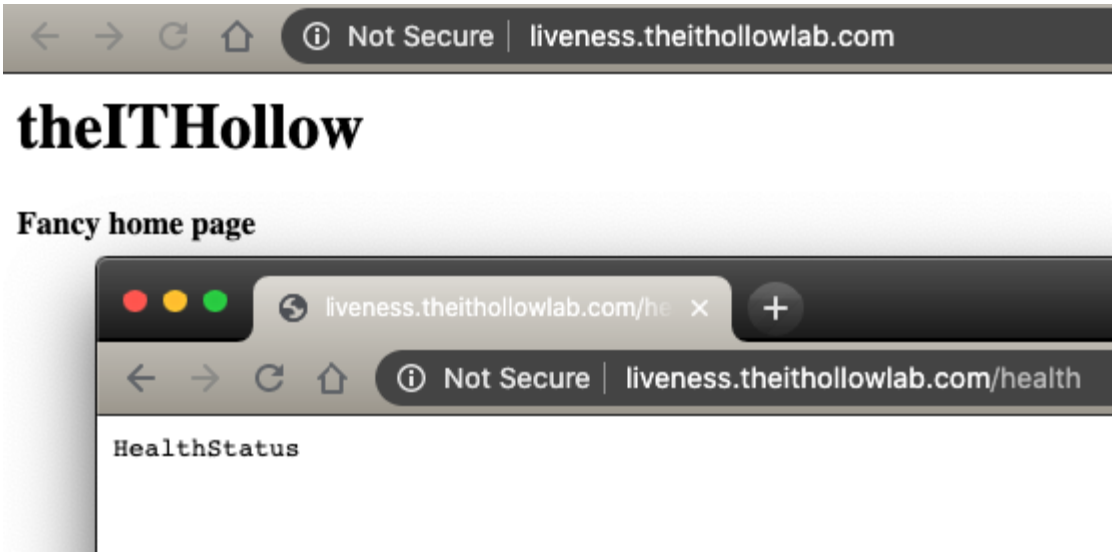
```
        path: /health
        port: 80
      initialDelaySeconds: 3
      periodSeconds: 3
---
apiVersion: v1
kind: Service
metadata:
  name: liveness
spec:
  selector:
    app: liveness
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: liveness-ingress
  namespace: default
spec:
  rules:
  - host: liveness.theithollowlab.com
    http:
      paths:
      - backend:
          serviceName: liveness
          servicePort: 80
```

This manifest includes two probes:

1. Liveness check doing an HTTP request against "/"
2. Readiness check doing an HTTP request agains /health

```
    livenessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 3
      periodSeconds: 3
    readinessProbe:
      httpGet:
        path: /health
        port: 80
      initialDelaySeconds: 3
      periodSeconds: 3
```

My container uses a script to start the HTTP daemon right away, and then waits 60 seconds before creating a /health page. This is to simulate some work being done by the application and the app isn't ready for consumption. This is the entire website for reference.




And here is my container script.

```
/usr/sbin/httpd > /dev/null 2>&1 &. #Start HTTP Daemon
sleep 60. #wait 60 seconds
echo HealthStatus > /var/www/html/health #Create Health status page
sleep 3600
```

Code language: PHP (php)

Deploy the manifest through kubectl apply. Once deployed, I've run a `--watch` command to keep an eye on the deployment. Here's what it looked like.

```
NAME             READY   STATUS    RESTARTS   AGE
liveness-http    0/1     Running   0          12s
liveness-http    1/1     Running   0          64s
```

You'll notice that the ready status showed 0/1 for about 60 seconds. Meaning that my container was not in a ready status for 60 seconds until the /health page became available through the startup script.

As a silly example, what if we modified our liveness probe to look for /health? Perhaps we have an application that sometimes stops working, but doesn't crash. Will the application ever startup? Here's my new probe in the yaml manifest.

```
    livenessProbe:
      httpGet:
        path: /health
        port: 80
      initialDelaySeconds: 3
      periodSeconds: 3
```

After deploying this, let's run another `--watch` on the pods. Here we see that the pod is restarting, and I am unable to ever access the /health page because it restarts before its ready.

```
NAME             READY   STATUS    RESTARTS   AGE
liveness-http    1/1     Running   0          3s
liveness-http    1/1     Running   1          36s
liveness-http    1/1     Running   2          69s
liveness-http    1/1     Running   3          102s
```

We can see that the liveness probe is failing if we run a describe on the pod.

```
Events:
  Type     Reason     Age   From                                               Message
  ----     ------     ----  ----                                               -------
  Normal   Scheduled  20s   default-scheduler                                  Successfully assigned default/liveness-http to ip-10-0-1-20.us
nal
  Normal   Pulled     19s   kubelet, ip-10-0-1-20.us-east-2.compute.internal   Container image "theithollow/hollowapp-blog:liveness" already
  Normal   Created    19s   kubelet, ip-10-0-1-20.us-east-2.compute.internal   Created container liveness
  Normal   Started    19s   kubelet, ip-10-0-1-20.us-east-2.compute.internal   Started container liveness
  Warning  Unhealthy  16s   kubelet, ip-10-0-1-20.us-east-2.compute.internal   Liveness probe failed: HTTP probe failed with statuscode: 404
  Normal   Killing    16s   kubelet, ip-10-0-1-20.us-east-2.compute.internal   Container liveness failed liveness probe, will be restarted
```

# Kubernetes Validating Admission Controllers

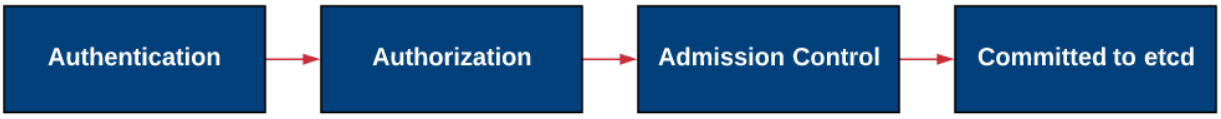
Hey! Who deployed this container in our shared Kubernetes cluster without putting resource limits on it? Why don't we have any labels on these containers so we can report for charge back purposes? Who allowed this image to be used in our production cluster?

If any of the questions above sound familiar, its probably time to learn about Validating Admission Controllers.

## Validating Admission Controllers – The Theory

Admission Controllers are used as a roadblocks before objects are deployed to a Kubernetes cluster. The examples from the section above are common rules that companies might want to enforce before objects get pushed into a production Kubernetes cluster. These admission controllers can be from custom code that you've written yourself, or a third party admission controller. A common open-source project that manages admission control rules is Open Policy Agent (OPA).

A generalized request flow for new objects starts with Authenticating with the API, then being authorized, and then optionally hitting an admission controller, before finally being committed to the etcd store.



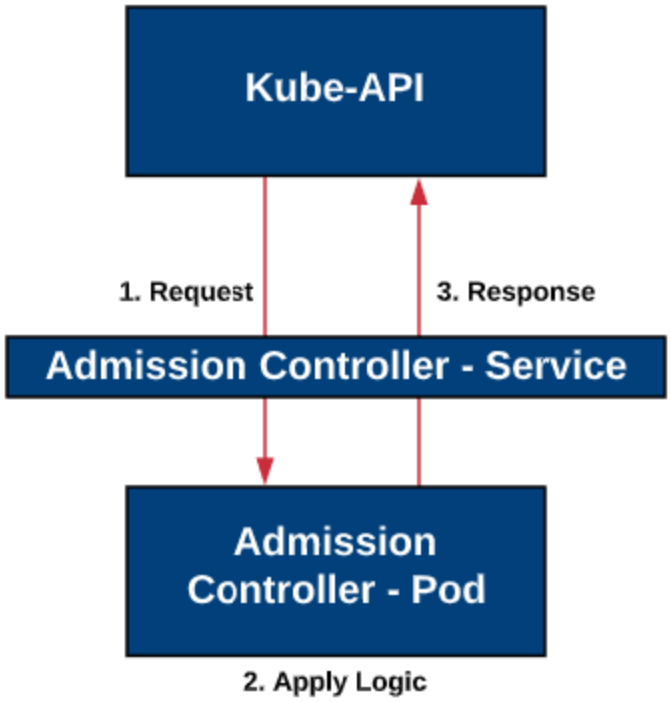

The great part about an admission controller is that you've got an opportunity to put our own custom logic in as a gate for API calls. This can be done in two forms.

**ValidatingAdmissionController –** This type of admission controller returns a binary result. Essentially, this means either **yes** the object is permitted or **no** the object is not permittetd.

**MutatingAdmissionController –** This type of admission controller has the option of modifying the API call and replacing it with a different version.

As an example, our example requirement that a label needs to be added to all pods, could be handled by either of these methods. A validating admission controller would allow or deny a pod from being deployed without the specified tag. Meanwhile, in a mutating admission controller, we could add a tag if one was missing, and then approve it. This post focuses on building a validating admission controller.

Once the API request hits our admission controller webhook, it will make a REST call to the admission controller. Next, the admission controller will apply some logic on the Request, and then make a response call back to the API server with the results.




The api request to the admission controller should look similar to the request below. This was lifted directly from the [v1.18 Kubernetes documentation site](#).

```
{
  "apiVersion": "admission.k8s.io/v1",
```

```
  "kind": "AdmissionReview",
  "request": {
    # Random uid uniquely identifying this admission call
    "uid": "705ab4f5-6393-11e8-b7cc-42010a800002",

    # Fully-qualified group/version/kind of the incoming object
    "kind": {"group":"autoscaling","version":"v1","kind":"Scale"},
    # Fully-qualified group/version/kind of the resource being modified
    "resource":
{"group":"apps","version":"v1","resource":"deployments"},
    # subresource, if the request is to a subresource
    "subResource": "scale",

    # Fully-qualified group/version/kind of the incoming object in the
original request to the API server.
    #  This  only  differs  from  `kind`  if  the  webhook  specified
`matchPolicy: Equivalent` and the
    # original request to the API server was converted to a version the
webhook registered for.
    "requestKind":
{"group":"autoscaling","version":"v1","kind":"Scale"},
    # Fully-qualified group/version/kind of the resource being modified
in the original request to the API server.
    #  This  only  differs  from  `resource`  if  the  webhook  specified
`matchPolicy: Equivalent` and the
    # original request to the API server was converted to a version the
webhook registered for.
    "requestResource":
{"group":"apps","version":"v1","resource":"deployments"},
    # subresource, if the request is to a subresource
    #  This  only  differs  from  `subResource`  if  the  webhook  specified
`matchPolicy: Equivalent` and the
    # original request to the API server was converted to a version the
webhook registered for.
    "requestSubResource": "scale",

    # Name of the resource being modified
    "name": "my-deployment",
    #  Namespace  of  the  resource  being  modified,  if  the  resource  is
namespaced (or is a Namespace object)
    "namespace": "my-namespace",

    # operation can be CREATE, UPDATE, DELETE, or CONNECT
    "operation": "UPDATE",

    "userInfo": {
```

```
      # Username of the authenticated user making the request to the API
server
      "username": "admin",
      # UID of the authenticated user making the request to the API
server
      "uid": "014fbff9a07c",
      # Group memberships of the authenticated user making the request
to the API server
      "groups": ["system:authenticated","my-admin-group"],
      # Arbitrary extra info associated with the user making the request
to the API server.
      # This is populated by the API server authentication layer and
should be included
      # if any SubjectAccessReview checks are performed by the webhook.
      "extra": {
        "some-key":["some-value1", "some-value2"]
      }
    },

    # object is the new object being admitted.
    # It is null for DELETE operations.
    "object": {"apiVersion":"autoscaling/v1","kind":"Scale",...},
    # oldObject is the existing object.
    # It is null for CREATE and CONNECT operations.
    "oldObject": {"apiVersion":"autoscaling/v1","kind":"Scale",...},
    # options contains the options for the operation being admitted,
like meta.k8s.io/v1 CreateOptions, UpdateOptions, or DeleteOptions.
    # It is null for CONNECT operations.
    "options":
{"apiVersion":"meta.k8s.io/v1","kind":"UpdateOptions",...},

    # dryRun indicates the API request is running in dry run mode and
will not be persisted.
    # Webhooks with side effects should avoid actuating those side
effects when dryRun is true.
    # See http://k8s.io/docs/reference/using-api/api-concepts/#make-a-
dry-run-request for more details.
    "dryRun": false
  }
}
```

Code language: PHP (php)

The response REST call back to the API server should look similar to the response below. The uid must match the uid from the request in v1 of the admission.k8s.io api. The `allowed` field is true for permitting the request to go through and false if it should be denied.

```
{
```

```
  "apiVersion": "admission.k8s.io/v1",
  "kind": "AdmissionReview",
  "response": {
    "uid": "<value from request.uid>",
    "allowed": true/false
  }
}
```
Code language: JSON / JSON with Comments (json)

# Validating Admission Controllers – In Action

Now, let's build our own custom validating admission controller in python using a Flask API. The goal of this controller is to ensure that all deployments and pods have a "Billing" label added. If you'd like to get a headstart with the code presented in this post, you may want to pull down the github repo: https://github.com/theITHollow/warden

We'll assume that we're doing some chargeback/showback to our customers and we need this label on everything or we can't identify what customer it belongs to, and we can't bill them.

### Create the Admission Controller API

The first step we'll go through is to build our flask API and put in our custom logic. If you look in the flask API example below, I'm handling an incoming request to the `/validate` URI and checking the metadata of the object for a "billing" label. I'm also capturing the uid of the request so I can pass that back in the response. Also, be sure to provide a message so that the person requesting the object gets feedback about why the operation was not permitted. Then depending on the label being found, a response is created with an allowed value of True or False.

This is a very simple example, but once you've set this up, use your own custom logic.

```
from flask import Flask, request, jsonify

warden = Flask(__name__)

#POST route for Admission Controller
@warden.route('/validate', methods=['POST'])

#Admission Control Logic
def deployment_webhook():
    request_info = request.get_json()
    uid = request_info["request"].get("uid")
    try:
        if
request_info["request"]["object"]["metadata"]["labels"].get("billing")
:
```

```
            #Send response back to controller if validations succeeds
            return k8s_response(True, uid, "Billing label exists")
    except:
        return k8s_response(False, uid, "No labels exist. A Billing label is required")

    #Send response back to controller if failed
    return k8s_response(False, uid, "Not allowed without a billing label")

#Function to respond back to the Admission Controller
def k8s_response(allowed, uid, message):
    return jsonify({"apiVersion": "admission.k8s.io/v1", "kind": "AdmissionReview", "response": {"allowed": allowed, "uid": uid, "status": {"message": message}}})

if __name__ == '__main__':
    warden.run(ssl_context=('certs/wardencrt.pem', 'certs/wardenkey.pem'),debug=True, host='0.0.0.0')
```

Code language: PHP (php)

One of the requirements for an admission controller is that it is protected by certificates. So, let's go create some certificates. I've created a script to generate these certificates for us and it's stored in the git repository. The CN is important here, so it should match the DNS name of your admission controller.

NOTE: As of Kubernetes version 1.19 SAN certificates are required. If this will be deployed on 1.19 or higher, you must create a SAN Certificate. The github repository has been updated with an ext.cnf file and script will deploy a SAN certificate now.

Since I'll be deploying this as a container within k8s, the service name exposing it is `warden` and the namespace it will be stored in will be `validation`. You should modify the script and the ext.cnf file before using it yourself.

```
keydir="certs"
cd "$keydir"

# CA root key
openssl genrsa -out ca.key 4096

#Create and sign the Root CA
openssl req -x509 -new -nodes -key ca.key -sha256 -days 1024 -out ca.crt -subj "/CN=Warden Controller Webhook"

#Create certificate key
openssl genrsa -out warden.key 2048

#Create CSR
```

```
openssl req -new -sha256 \
    -key warden.key \
    -config ../ext.cnf \
    -out warden.csr

#Generate the certificate
openssl x509 -req -in warden.csr -CA ca.crt -CAkey ca.key -CAcreateserial
-out  warden.crt  -days  500  -sha256  -extfile  ../ext.cnf  -extensions
req_ext

# Create .pem versions
cp warden.crt wardencrt.pem \
    | cp warden.key wardenkey.pem
```

Code language: PHP (php)

The associated ext.cnf file is show below. Notice the alt_names section, which must match the name of your admission controller service. Also, feel free to update the country, locality, and organization to match your environment.

```
[ req ]
default_bits        = 2048
distinguished_name = req_distinguished_name
req_extensions     = req_ext
prompt              = no
[ req_distinguished_name ]
countryName                 = US
stateOrProvinceName         = Illinois
localityName                = Chicago
organizationName            = HollowLabs
commonName                  = Warden Controller Webhook
[ req_ext ]
subjectAltName = @alt_names
[alt_names]
DNS.1   = warden.validation.svc
```

You'll notice that the flask code is using these certificates and if you don't change the names or locations in the script, it should just work. If you made modifications you will need to update the last line of the python code.

```
warden.run(ssl_context=('certs/wardencrt.pem',
'certs/wardenkey.pem'),debug=True, host='0.0.0.0')
```

### Build and Deploy the Admission Controller

It's time to build a container for this pod to run in. My `Dockerfile` is listed below as well as in the git repo.

```
FROM ubuntu:16.04

RUN apt-get update -y && \
```

```
    apt-get install -y python-pip python-dev

# We copy just the requirements.txt first to leverage Docker cache
COPY ./requirements.txt /app/requirements.txt

WORKDIR /app

RUN pip install -r requirements.txt

COPY . /app

ENTRYPOINT [ "python" ]

CMD [ "app/warden.py" ]
```
Code language: PHP (php)

Push your image to your image registry in preparation for being deployed in your Kubernetes cluster.

Deploy the admission controller with the following Kubernetes manifest, after changing the name of your image.

```
---
apiVersion: v1
kind: Namespace
metadata:
   name: validation
---
apiVersion: v1
kind: Pod
metadata:
  name: warden
  labels:
    app: warden
  namespace: validation
spec:
  restartPolicy: OnFailure
  containers:
    - name: warden
      image: theithollow/warden:v1 #EXAMPLE- USE YOUR REPO
      imagePullPolicy: Always
---
apiVersion: v1
kind: Service
metadata:
  name: warden
  namespace: validation
spec:
```

```
  selector:
    app: warden
  ports:
  - port: 443
    targetPort: 5000
```
Code language: PHP (php)

**Deploy the Webhook Configuration**

The admission controller has been deployed and is waiting for some requests to come in. Now, we need to deploy the webhook configuration that tells the Kubernetes API to check with the admission controller that we just deployed.

The webhook configuration needs to know some information about what types of objects it's going to make these REST calls for, as well as the URI to send them to. The `clientConfig` section contains info about where to make the API call. Within the `rules` section, you'll define what apiGroups, resources, versions and operations will trigger the requests. This will seem similar to RBAC polices. Also note the `failurePolicy` which defines what happens to object requests if the admission controller is unreachable.

```
---
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name: validating-webhook
  namespace: validation
webhooks:
  - name: warden.validation.svc
    failurePolicy: Fail
    sideEffects: None
    admissionReviewVersions: ["v1","v1beta1"]
    rules:
      - apiGroups: ["apps", ""]
        resources:
          - "deployments"
          - "pods"
        apiVersions:
          - "*"
        operations:
          - CREATE
    clientConfig:
      service:
        name: warden
        namespace: validation
        path: /validate/
      caBundle: #See command below
```

The last piece of the config is the base64 encoded version of the CA certificate. If you used the script in the git repo, the command below will print the `caBundle` information for the manifest.

```
cat certs/ca.crt | base64
```

Deploy the webook. `kubectl apply -f [manifest].yaml`

**Test the Results**

There are three manifests in the `/test-pods` folder that can be used to test with.

**test1.yaml** – Should work with the admission controller because it has a proper `billing` label.

```
pod/test1 created
```

**test2.yaml –** Should fail, because there are no labels assigned.

```
Error from server: error when creating "test-pods/test2.yaml": admission webhook "warden.validation.svc"
denied the request: No labels exist. A Billing label is required
```

**test3.yaml –** Should fail, because while it does have a label, it does not have a `billing` label.

```
Error from server: error when creating "test-pods/test3.yaml": admission webhook "warden.validation.svc"
denied the request: Not allowed without a billing label
```

Notice that each of these tests provided different responses. These responses can be customized so that you can give good feedback on why an operation was not permitted.

# Kubernetes – Jobs and CronJobs

Sometimes we need to run a container to do a specific task, and when its completed, we want it to quit. Many containers are deployed and continuously run, such as a web server. But other times we want to accomplish a single task and then quit. This is where a Job is a good choice.

## Jobs and CronJobs – The Theory

Perhaps, we need to run a batch process on demand. Maybe we built an automation routine for something and want to kick it off through the use of a container. We can do this by submitting a job to the Kubernetes API. Kubernetes will run the job to completion and then quit.

Now, thats kind of handy, but what if we want to run this job we've automated at a specific time every single day. Maybe you need to perform some batch processing at the end of a

workday, or perhaps you want to destroy pods at the end of the day to free up resources. There could be a myriad of things that you'd like to be done at different times during the day or night.

A Kubernetes CronJob lets you schedule a container to be run at a time, defined by you, much the same way that a linux server would schedule a cron job. Think of the CronJob as a Job with a schedule.

## Jobs and CronJobs – In Action

First, we'll deploy a simple job that will run curl against this blog just to see if it will work. I've created a simple alpine container with the curl installed on it. We'll use this container image to run curl against this site. The manifest below will create our simple curl job.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: hollow-curl
spec:
  template:
    spec:
      containers:
          - name: hollow-curl
            image: theithollow/hollowapp-blog:curl
            args:
            - /bin/sh
            - -c
            - curl https://theithollow.com
      restartPolicy: Never
  backoffLimit: 4
```

Code language: JavaScript (javascript)

Apply the job manifest with the following command:

```
kubectl apply -f [job manifest file].yml
```

Code language: CSS (css)

Once the job has been submitted, we can see that a job was run by running a "get" on the job object. We can see that a job was run and how long it took.

```
eshanks-a01:k8s-guide eshanks$ kubectl get job
NAME          COMPLETIONS   DURATION   AGE
hollow-curl   1/1           4s         6s
```

To go a bit deeper, we can run a describe on our job to see additional details about it. Some interesting information is shown below for the job I submitted to my cluster.

```
Pods Statuses:  0 Running / 1 Succeeded / 0 Failed
Pod Template:
  Labels:  controller-uid=dad26948-c41e-469e-8780-45770d3bc4ac
           job-name=hollow-curl
  Containers:
   hollow-curl:
    Image:      theithollow/hollowapp-blog:curl
    Port:       <none>
    Host Port:  <none>
    Args:
      /bin/sh
      -c
      curl https://theithollow.com
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From            Message
  ----    ------            ----  ----            -------
  Normal  SuccessfulCreate  21s   job-controller  Created pod: hollow-curl-chvqf
```

According to the events, we can see that our job completed successfully already. Next, lets just check if there are any pods deployed. Sure enough, a pod was deployed and it has a status of completed.

```
NAME                READY   STATUS      RESTARTS   AGE
hollow-curl-chvqf   0/1     Completed   0          6m43s
```

Now, what if we want to run the same job on a set schedule? For this example, we'll create a simple CronJob that will run curl against this blog at the top of the hour. We can pretend that we're checking uptime on the website if we want it to feel more like a real world example, but your cron job could execute anything you could dream up and drop into a container.

Let's build a manifest file to deploy a CronJob with the appropriate schedule, which in this case will be every hour, at the top of the hour. The manifest to create this CronJob is shown below.

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hollow-curl
spec:
  schedule: "0 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
```

```
          - name: hollow-curl
            image: theithollow/hollowapp-blog:curl
            args:
            - /bin/sh
            - -c
            - curl https://theithollow.com
          restartPolicy: OnFailure
```
Code language: JavaScript (javascript)

Deploy the CronJob by running:

```
kubectl apply -f [manifest file name].yml
```
Code language: CSS (css)

Once deployed, we can check to see if we can list the CronJob by running a "get" on the cronjob object.

```
eshanks-a01:k8s-guide eshanks$ kubectl get cronjob
NAME          SCHEDULE    SUSPEND   ACTIVE   LAST SCHEDULE
hollow-curl   0 * * * *   False     0        <none>
```

Now, at the top of the hour, we should be able to run this again and see that its been run successfully. In my case, I ran a –watch so that I could see what happened at the top of the hour. We can see that it went from not scheduled, to active, to completed.

```
eshanks-a01:k8s-guide eshanks$ k get cronjob --watch
NAME          SCHEDULE    SUSPEND   ACTIVE   LAST SCHEDULE   AGE
hollow-curl   0 * * * *   False     0        <none>          50m
hollow-curl   0 * * * *   False     1        9s              50m
hollow-curl   0 * * * *   False     0        19s             50m
```

**Summary**

Jobs and CronJobs may seem like a really familiar solution to what we've done to run batch jobs for many years. Now we can take some of our jobs that were stored on servers and replace them with a container and drop those in our Kubernetes cluster as well. Good luck with your jobs!

# Deploy Kubernetes Using Kubeadm – CentOS7

I've been wanting to have a playground to mess around with Kubernetes (k8s) deployments for a while and didn't want to spend the money on a cloud solution like AWS Elastic Container Service for Kubernetes or Google Kubernetes Engine. While these hosted

solutions provide additional features such as the ability to spin up a load balancer, they also cost money every hour they're available and I'm planning on leaving my cluster running. Also, from a learning perspective, there is no greater way to learn the underpinnings of a solution than having to deploy and manage it on your own. Therefore, I set out to deploy k8s in my vSphere home lab on some CentOS 7 virtual machines using Kubeadm. I found several articles on how to do this but somehow I got off track a few times and thought another blog post with step by step instructions and screenshots would help others. Hopefully it helps you. Let's begin.

# Prerequisites

This post will walk through the deployment of Kubernetes version 1.13.2 (latest as of this posting) on three CentOS 7 virtual machines. One virtual machine will be the Kubernetes master server where the control plane components will be run and two additional nodes where the containers themselves will be scheduled. To begin the setup, deploy three CentOS 7 servers into your environment and be sure that the master node has 2 CPUs and all three servers have at least 2 GB of RAM on them. Remember the nodes will run your containers so the amount of RAM you need on those is dependent upon your workloads.

Also, I'm logging in as root on each of these nodes to perform the commands. Change this as you need if you need to secure your environment more than I have in my home lab.

### Prepare Each of the Servers for K8s

On all three (or more if you chose to do more nodes) of the servers we'll need to get the OS setup to be ready to handle our kubernetes deployment via kubeadm. Lets start with stopping and disabling firewalld by running the commands on each of the servers:

```
systemctl stop firewalld
systemctl disable firewalld
```

Next, let's disable swap. Kubeadm will check to make sure that swap is disabled when we run it, so lets turn swap off and disable it for future reboots.

```
swapoff -a
sed -i.bak -r 's/(.+ swap .+)/#\1/' /etc/fstab
```

Code language: JavaScript (javascript)

Now we'll need to disable SELinux if we have that enabled on our servers. I'm leaving it on, but setting it to Permissive mode.

```
setenforce 0
sed -i 's/^SELINUX=enforcing$/SELINUX=permissive/' /etc/selinux/config
```

Code language: JavaScript (javascript)

Next, we'll add the kubernetes repository to yum so that we can use our package manager to install the latest version of kubernetes. To do this we'll create a file in the /etc/yum.repos.d directory. The code below will do that for you.

```
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
```

```
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-
x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
exclude=kube*
EOF
```
Code language: JavaScript (javascript)

Now that we've updated our repos, lets install some of the tools we'll need on our servers including kubeadm, kubectl, kubelet, and docker.

```
yum install -y docker kubelet kubeadm kubectl --
disableexcludes=kubernetes
```

After installing docker and our kubernetes tools, we'll need to enable the services so that they persist across reboots, and start the services so we can use them right away.

```
systemctl enable docker && systemctl start docker
systemctl enable kubelet && systemctl start kubelet
```

Before we run our kubeadm setup we'll need to enable iptables filtering so proxying works correctly. To do this run the following to ensure filtering is enabled and persists across reboots.

```
cat <<EOF > /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF

sysctl --system
```
Code language: JavaScript (javascript)

# Setup Kubernetes Cluster on Master Node

We're about ready to initialize our kubernetes cluster but I wanted to take a second to mention that we'll be using Flannel as the network plugin to enable our pods to communicate with one another. You're free to use other network plugins such as Calico or Cillium but this post focuses on the Flannel plugin specifically.

Let's run kubeadm init on our master node with the –pod-network switch needed for Flannel to work properly.

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

When you run the init command several pre-flight checks will take place including making sure swap is off and iptables filtering is enabled.

```
[root@kube-master ~]# kubeadm init --pod-network-cidr=10.244.0.0/16
[init] Using Kubernetes version: v1.13.2
[preflight] Running pre-flight checks
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet connection
[preflight] You can also perform this action in beforehand using 'kubeadm config images pull'
```

After the initialization is complete you should have a working kubernetes master node setup. The last few lines output from the kubeadm init command are important to take note of since it shows the commands to run on the worker nodes to join the cluster.

```
Your Kubernetes master has initialized successfully!

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

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

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

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

  kubeadm join 10.10.50.50:6443 --token zzpnsn.jaf69dl0fb0z8yop --discovery-token-ca-cert-hash
```

In my case, it showed that I should use the following command to join this cluster which we'll use in the following section.

```
kubeadm join 10.10.50.50:6443 --token zzpnsn.jaf69dl0fb0z8yop --discovery-token-ca-cert-hash sha256:8f98d7e7fc701b2d686da84d7708376f4b59d83c0173b68a459a86cbe4922562
```

Code language: CSS (css)

Before we start working on the other nodes, lets make sure we can run the kubectl commands from the master node, in case we want to use it later on.

```
export KUBECONFIG=/etc/kubernetes/admin.conf
```

Code language: JavaScript (javascript)

After setting our KUBECONFIG variable, we can run kubectl get nodes just to verify that the cluster is serving requests to the API. You can see that I have 1 node of type master setup now.

```
[root@kube-master ~]# kubectl get nodes
NAME          STATUS     ROLES    AGE     VERSION
kube-master   NotReady   master   7m10s   v1.13.2
```

# Configure Member Nodes

Now that the master is setup, lets focus on joining our two member nodes to the cluster and setup the networking. To begin, lets join the nodes to the cluster using the command that was output from the kubeadm init command we ran on the master. This is the command that

I used, but your cluster will have a different value. Please use your own and not the one shown below as an example.

```
kubeadm join 10.10.50.50:6443 --token zzpnsn.jaf69dl0fb0z8yop --discovery-token-ca-cert-hash sha256:8f98d7e7fc701b2d686da84d7708376f4b59d83c0173b68a459a86cbe4922562
```

Code language: CSS (css)

Once you run the command on all of your nodes you should get a success message that lets you know that you've joined the cluster successfully.

```
This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the master to see this node join the cluster.
```

At this point I'm going to copy the admin.conf file over to my worker nodes and set KUBECONFIG to use it for authentication so that I can run kubectl commands on the worker nodes as well. This is step is optional if you are going to run kubectl commands elsewhere, like from your desktop computer.

```
scp root@[MASTERNODEIP]:/etc/kubernetes/admin.conf /etc/kubernetes/admin.conf

export KUBECONFIG=/etc/kubernetes/admin.conf
```

Code language: JavaScript (javascript)

Now if you run the kubectl get nodes command again, we should see that there are three nodes in our cluster.

```
[root@kube-master ~]# kubectl get nodes
NAME          STATUS     ROLES    AGE    VERSION
kube-master   NotReady   master   18m    v1.13.2
kube-node1    NotReady   <none>   2m8s   v1.13.2
kube-node2    NotReady   <none>   2m     v1.13.2
```

If you're closely paying attention you'll notice that the nodes show a status of "NotReady" which is a concern. This is because we haven't finished deploying our network components for the Flannel plugin. To do that we need to deploy the flannel containers in our cluster by running:

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/bc79dd1505b0c8681ece4de4c0d86c5cd2643275/Documentation/kube-flannel.yml
```

Code language: JavaScript (javascript)

Give your cluster a second and re-run the kubectl get nodes command. You should see the nodes are all in a Ready status now.

```
NAME          STATUS   ROLES    AGE     VERSION
kube-master   Ready    master   24m     v1.13.2
kube-node1    Ready    <none>   8m7s    v1.13.2
kube-node2    Ready    <none>   7m59s   v1.13.2
```

## Configure a Mac Client

You've got your cluster up and running and are ready to go now. The last step might be to setup your laptop to run the kubectl commands against your k8s cluster. To do this, be sure to install the kubectl binaries on your mac and then copy the admin.conf file to your laptop by running this in a terminal window. Be sure to replace the IP with the master node IP address.

```
scp root@[MasterNodeIP]:/etc/kubernetes/admin.conf ~/.kube/admin.conf
export KUBECONFIG=~/.kube/admin.conf
```

Code language: JavaScript (javascript)

Once done, you should be able to run kubectl get nodes from your laptop and start deploying your configurations.

# Deploy Kubernetes on AWS

The way you deploy Kubernetes (k8s) on AWS will be similar to how it was done in a previous post on vSphere. You still setup nodes, you still deploy kubeadm, and kubectl but there are a few differences when you change your cloud provider. For instance on AWS we can use the LoadBalancer resource against the k8s API and have AWS provision an elastic load balancer for us. These features take a few extra tweaks in AWS.

## AWS Prerequisites

Before we start deploying Kubernetes, we need to ensure a few prerequisites are covered in our AWS environment. You will need the following components setup to use the cloud provider and have the kubeadm installation complete successfully.

- An AWS Account with administrative access.
- EC2 instances for Control plane nodes (This post uses three ubuntu nodes split across AWS Availability Zones, but this is not necessary.)
- EC2 instances hostname must match the private dns name assigned to it. For example, when I deployed my ec2 instance from template and ran `hostname`, I got ip-10-0-4-208. However, when I check the private DNS from the ec2 instance metadata by the following command `curl http://169.254.169.254/latest/meta-data/local-hostname` I received: ip-10-0-4-208.us-east-2.compute.internal. This won't work. The

hostname command must match the private dns name exactly. Use this command to set the hostname.

```
hostnamectl set-hostname <hostname.region.compute.internal>
```

Code language: CSS (css)

```
root@ip-10-0-1-10:/home/ubuntu# hostname
ip-10-0-1-10.us-east-2.compute.internal
root@ip-10-0-1-10:/home/ubuntu# curl http://169.254.169.254/latest/meta-data/local-hostname
ip-10-0-1-10.us-east-2.compute.internalroot@ip-10-0-1-10:/home/ubuntu#
```

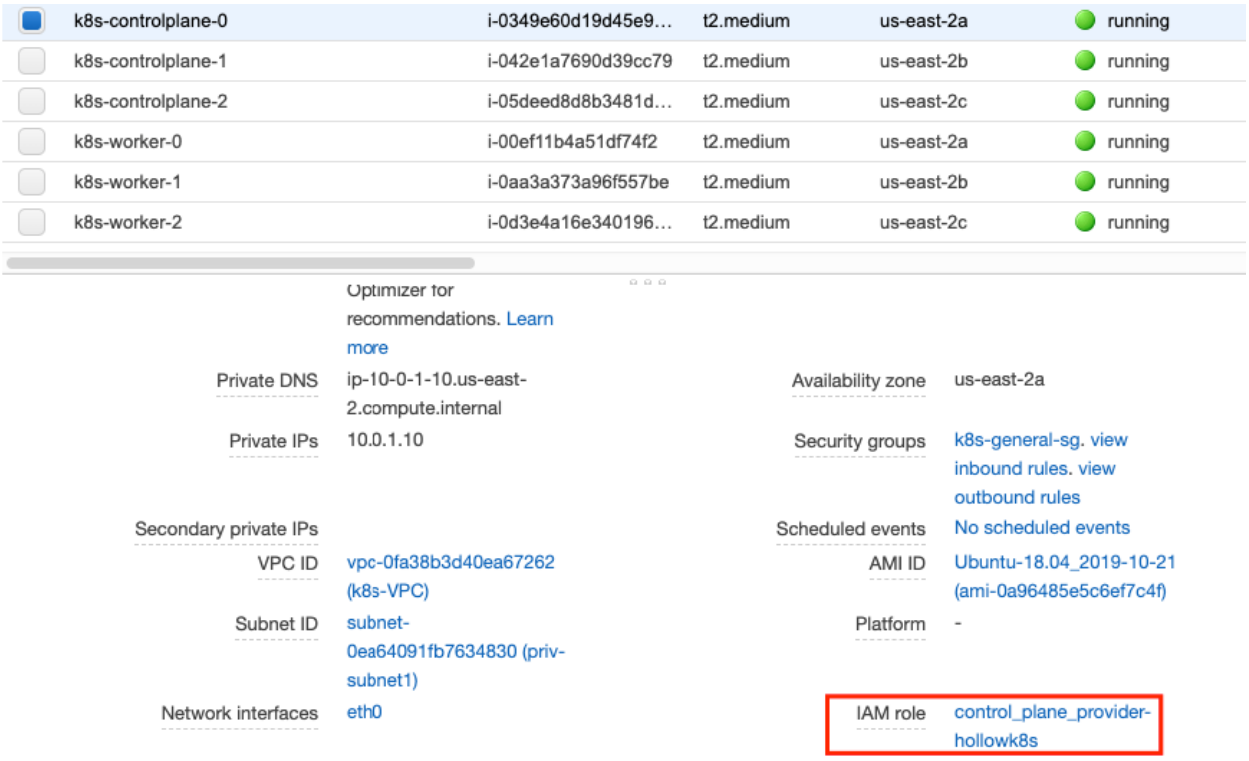
- To ensure the Control plane instances have access to AWS to spin up Load Balancers, ebs volumes, etc. we must apply an Instance Policy to the Control Plane nodes with the following iam policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "autoscaling:DescribeAutoScalingGroups",
        "autoscaling:DescribeLaunchConfigurations",
        "autoscaling:DescribeTags",
        "ec2:DescribeInstances",
        "ec2:DescribeRegions",
        "ec2:DescribeRouteTables",
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets",
        "ec2:DescribeVolumes",
        "ec2:CreateSecurityGroup",
        "ec2:CreateTags",
        "ec2:CreateVolume",
        "ec2:ModifyInstanceAttribute",
        "ec2:ModifyVolume",
        "ec2:AttachVolume",
        "ec2:AuthorizeSecurityGroupIngress",
        "ec2:CreateRoute",
        "ec2:DeleteRoute",
        "ec2:DeleteSecurityGroup",
        "ec2:DeleteVolume",
        "ec2:DetachVolume",
        "ec2:RevokeSecurityGroupIngress",
        "ec2:DescribeVpcs",
        "elasticloadbalancing:AddTags",
        "elasticloadbalancing:AttachLoadBalancerToSubnets",
        "elasticloadbalancing:ApplySecurityGroupsToLoadBalancer",
        "elasticloadbalancing:CreateLoadBalancer",
```

```
        "elasticloadbalancing:CreateLoadBalancerPolicy",
        "elasticloadbalancing:CreateLoadBalancerListeners",
        "elasticloadbalancing:ConfigureHealthCheck",
        "elasticloadbalancing:DeleteLoadBalancer",
        "elasticloadbalancing:DeleteLoadBalancerListeners",
        "elasticloadbalancing:DescribeLoadBalancers",
        "elasticloadbalancing:DescribeLoadBalancerAttributes",
        "elasticloadbalancing:DetachLoadBalancerFromSubnets",
        "elasticloadbalancing:DeregisterInstancesFromLoadBalancer",
        "elasticloadbalancing:ModifyLoadBalancerAttributes",
        "elasticloadbalancing:RegisterInstancesWithLoadBalancer",
```

```
"elasticloadbalancing:SetLoadBalancerPoliciesForBackendServer",
        "elasticloadbalancing:AddTags",
        "elasticloadbalancing:CreateListener",
        "elasticloadbalancing:CreateTargetGroup",
        "elasticloadbalancing:DeleteListener",
        "elasticloadbalancing:DeleteTargetGroup",
        "elasticloadbalancing:DescribeListeners",
        "elasticloadbalancing:DescribeLoadBalancerPolicies",
        "elasticloadbalancing:DescribeTargetGroups",
        "elasticloadbalancing:DescribeTargetHealth",
        "elasticloadbalancing:ModifyListener",
        "elasticloadbalancing:ModifyTargetGroup",
        "elasticloadbalancing:RegisterTargets",
        "elasticloadbalancing:DeregisterTargets",
        "elasticloadbalancing:SetLoadBalancerPoliciesOfListener",
        "iam:CreateServiceLinkedRole",
        "kms:DescribeKey"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

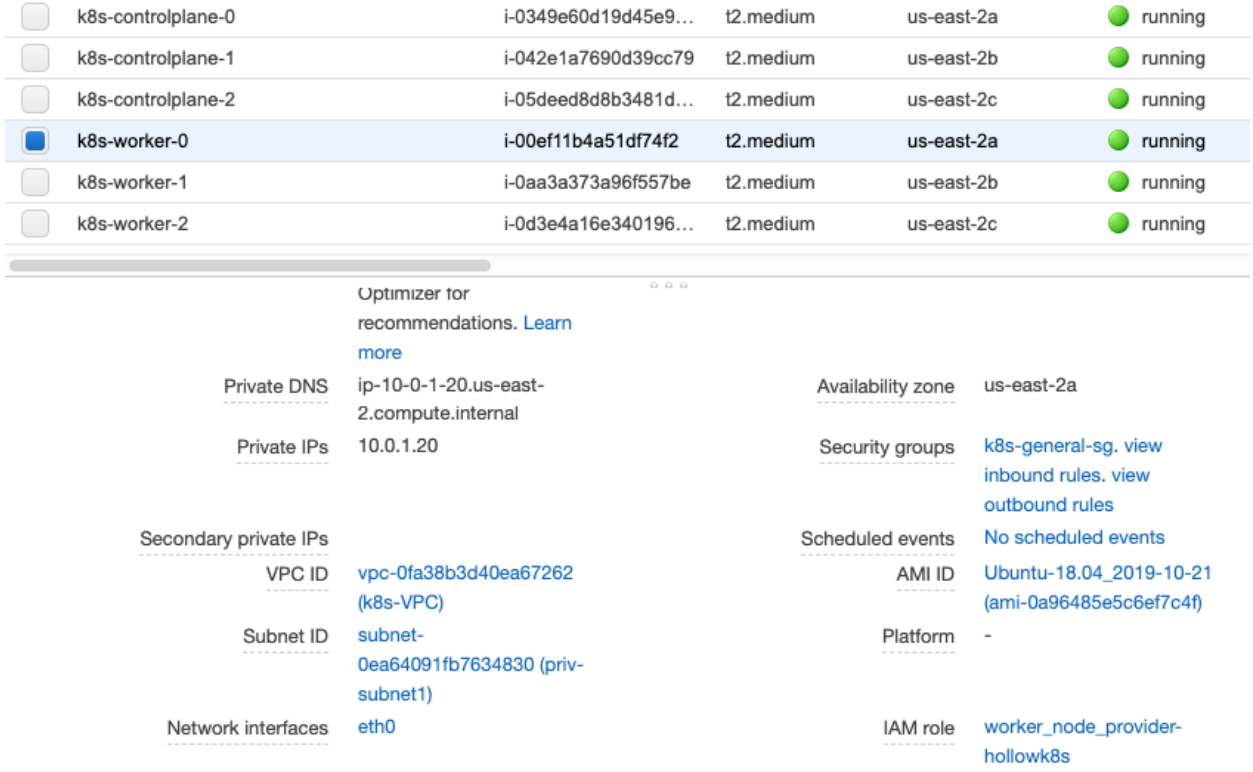
Code language: JSON / JSON with Comments (json)




- EC2 instances for worker nodes (Likewise, this post uses three ubuntu worker nodes across AZs)
- Worker node EC2 instances also need an AWS Instance Profile assigned to them with permissions to the AWS control plane.

```
{
      "Version": "2012-10-17",
      "Statement": [
          {
              "Effect": "Allow",
              "Action": [
                  "ec2:DescribeInstances",
                  "ec2:DescribeRegions",
                  "ecr:GetAuthorizationToken",
                  "ecr:BatchCheckLayerAvailability",
                  "ecr:GetDownloadUrlForLayer",
                  "ecr:GetRepositoryPolicy",
                  "ecr:DescribeRepositories",
                  "ecr:ListImages",
                  "ecr:BatchGetImage"
              ],
              "Resource": "*"
          }
      ]
  }
```

```
Code language: JSON / JSON with Comments (json)
```

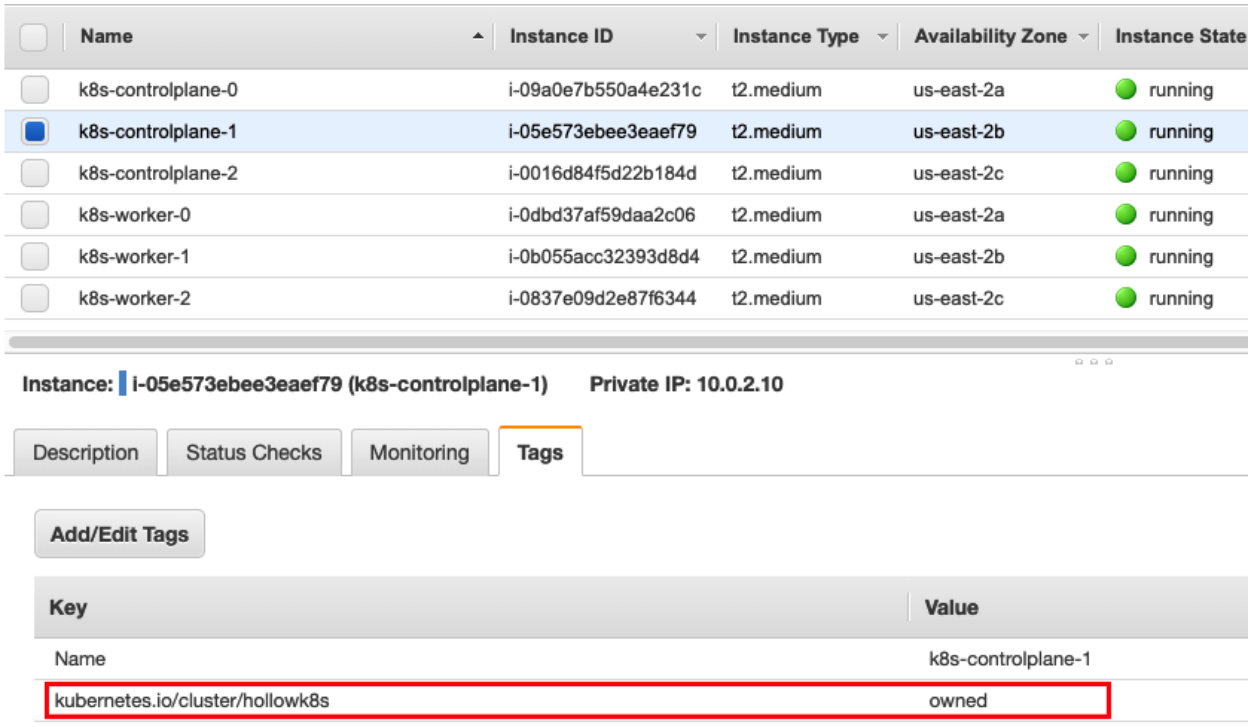


| | | | | |
|---|---|---|---|---|
| k8s-controlplane-0 | i-0349e60d19d45e9... | t2.medium | us-east-2a | running |
| k8s-controlplane-1 | i-042e1a7690d39cc79 | t2.medium | us-east-2b | running |
| k8s-controlplane-2 | i-05deed8d8b3481d... | t2.medium | us-east-2c | running |
| k8s-worker-0 | i-00ef11b4a51df74f2 | t2.medium | us-east-2a | running |
| k8s-worker-1 | i-0aa3a373a96f557be | t2.medium | us-east-2b | running |
| k8s-worker-2 | i-0d3e4a16e340196... | t2.medium | us-east-2c | running |

Optimizer for recommendations. Learn more

| | | | |
|---|---|---|---|
| Private DNS | ip-10-0-1-20.us-east-2.compute.internal | Availability zone | us-east-2a |
| Private IPs | 10.0.1.20 | Security groups | k8s-general-sg. view inbound rules. view outbound rules |
| Secondary private IPs | | Scheduled events | No scheduled events |
| VPC ID | vpc-0fa38b3d40ea67262 (k8s-VPC) | AMI ID | Ubuntu-18.04_2019-10-21 (ami-0a96485e5c6ef7c4f) |
| Subnet ID | subnet-0ea64091fb7634830 (priv-subnet1) | Platform | - |
| Network interfaces | eth0 | IAM role | worker_node_provider-hollowk8s |

- All EC2 instances (Worker and Control Plane nodes) must have a tag named `kubernetes.io/cluster/<CLUSTERNAME>` where CLUSTERNAME is a name you'll give your cluster.



| Name | Instance ID | Instance Type | Availability Zone | Instance State |
|---|---|---|---|---|
| k8s-controlplane-0 | i-09a0e7b550a4e231c | t2.medium | us-east-2a | running |
| k8s-controlplane-1 | i-05e573ebee3eaef79 | t2.medium | us-east-2b | running |
| k8s-controlplane-2 | i-0016d84f5d22b184d | t2.medium | us-east-2c | running |
| k8s-worker-0 | i-0dbd37af59daa2c06 | t2.medium | us-east-2a | running |
| k8s-worker-1 | i-0b055acc32393d8d4 | t2.medium | us-east-2b | running |
| k8s-worker-2 | i-0837e09d2e87f6344 | t2.medium | us-east-2c | running |

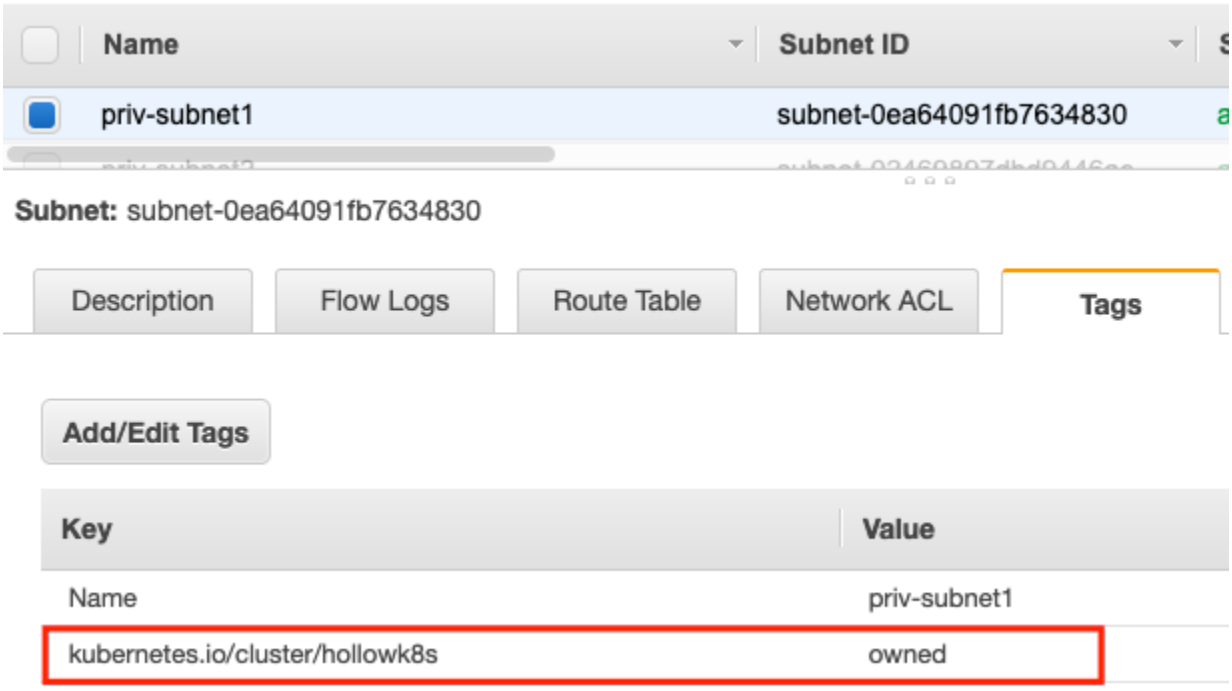
**Instance: i-05e573ebee3eaef79 (k8s-controlplane-1)** **Private IP: 10.0.2.10**

Description | Status Checks | Monitoring | **Tags**

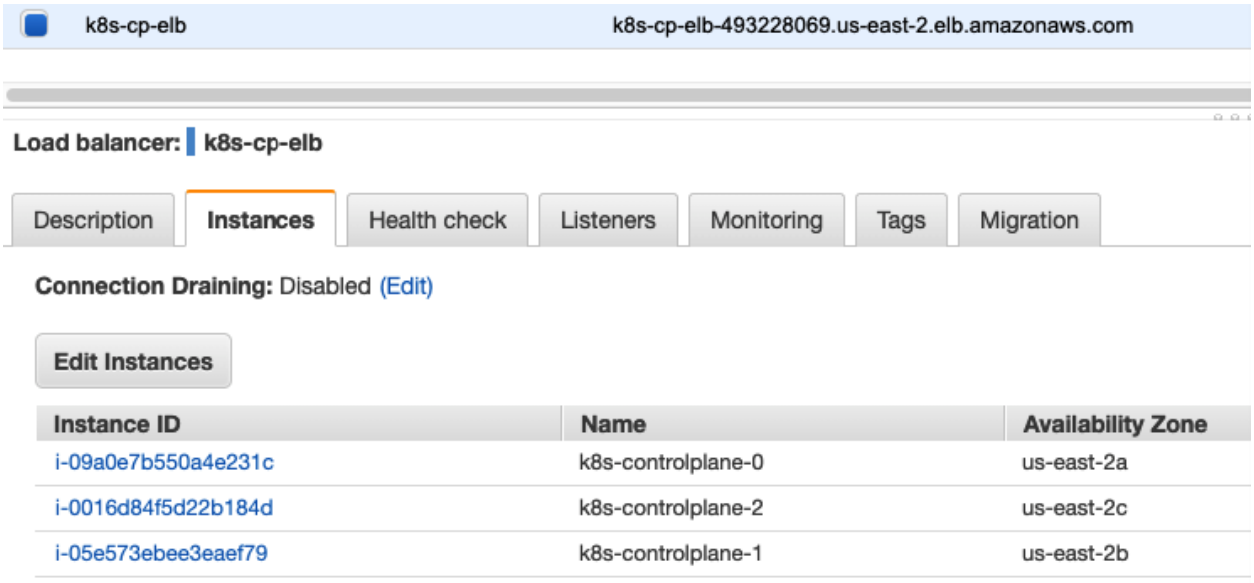
**Add/Edit Tags**

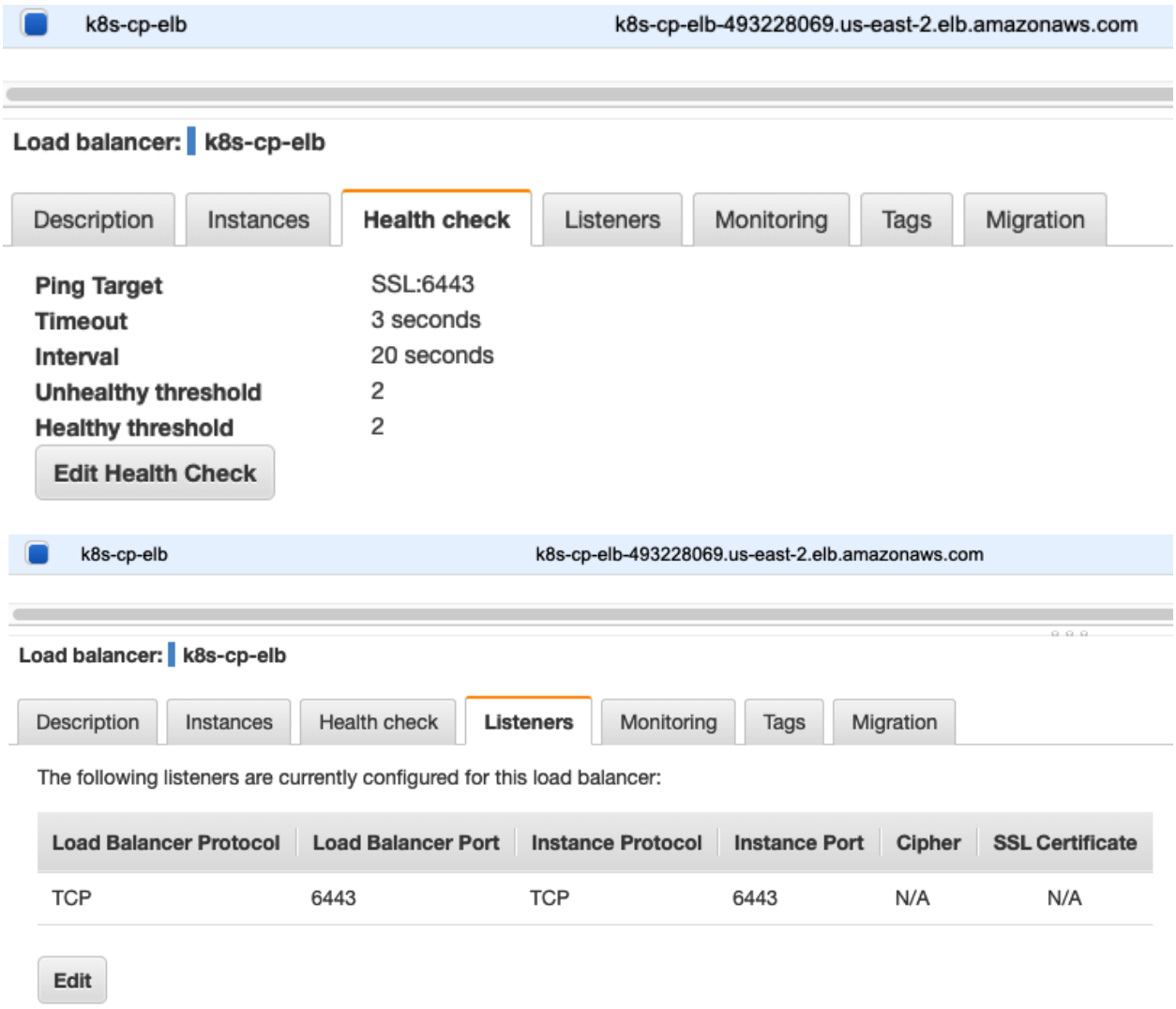
| Key | Value |
|---|---|
| Name | k8s-controlplane-1 |
| kubernetes.io/cluster/hollowk8s | owned |

- Subnets must have a tag named`kubernetes.io/cluster/<CLUSTERNAME>`. where CLUSTERNAME is the name you'll give your cluster. This is used when new Load Balancers are attached to Availability Zones.



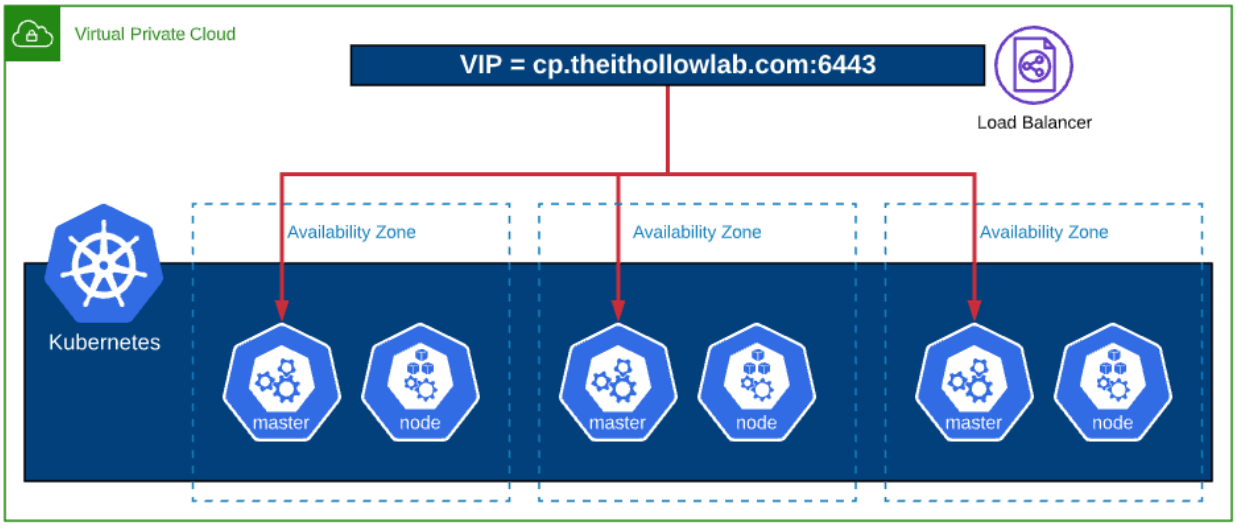

- An elastic load balancer setup and configured with the three master EC2 instances. There should be a health check on the targets of SSL:6443 as well as listeners.






- A DNS Entry configured to work with your load balancer.

The high level view of my instances are shown here:




# Create a Kubeadm Config File

Now that the AWS infrastructure is ready to go, we're ready to stat working on the Kubernetes pieces. The first section is dedicated to setting up a kubeadm.conf file. This file has instructions on how to setup the control plane components when we use kubeadm to bootstrap them. There are a ton of options that can be configured here, but we'll use a simple example that has AWS cloud provider configs included.

Create a kubeadm.conf file based on the example below, using your own environment information.

```
---
apiServer:
  extraArgs:
    cloud-provider: aws
apiServerCertSANs:
- cp.theithollowlab.com
apiServerExtraArgs:
  endpoint-reconciler-type: lease
apiVersion: kubeadm.k8s.io/v1beta1
clusterName: hollowk8s #your cluster name
controlPlaneEndpoint: cp.theithollowlab.com #your VIP DNS name
controllerManager:
  extraArgs:
    cloud-provider: aws
    configure-cloud-routes: 'false'
kind: ClusterConfiguration
kubernetesVersion: 1.17.0 #your desired k8s version
networking:
  dnsDomain: cluster.local
  podSubnet: 172.16.0.0/16 #your pod subnet matching your CNI config
```

```
nodeRegistration:
  kubeletExtraArgs:
    cloud-provider: aws
```
Code language: PHP (php)

# Kubernetes EC2 Instance Setup

Now, we can start installing components on the ubuntu instances before we deploy the cluster. Do this on all virtual machines that will be part of your Kubernetes cluster.

Disable swap

```
swapoff -a
sed -i.bak -r 's/(.+ swap .+)/#\1/' /etc/fstab
```
Code language: JavaScript (javascript)

Install Kubelet, Kubeadm, and Kubectl.

```
sudo apt-get update && sudo apt-get install -y apt-transport-https curl
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo
apt-key add -
cat <<EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
sudo apt-get update
sudo apt-get install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl
```
Code language: JavaScript (javascript)

Install Docker and change the cgroup driver to systemd.

```
sudo apt install docker.io -y
```

```
cat > /etc/docker/daemon.json <<EOF
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m"
  },
  "storage-driver": "overlay2"
}
EOF
```

```
sudo systemctl restart docker
sudo systemctl enable docker
```
Code language: JavaScript (javascript)

Once you've completed the steps above, copy the kubeadm.conf file to /etc/kubernetes/ on the Control Plane VMs.

After the kubeadm.conf file is placed we need to update the configuration of the kubelet service so that it knows about the AWS environment as well. Edit the `/etc/systemd/system/kubelet.service.d/10-kubeadm.conf` and add an additional configuration option.

`--cloud-provider=aws`

```
[Service]
Environment="KUBELET_KUBECONFIG_ARGS=--bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf"
Environment="KUBELET_CONFIG_ARGS=--config=/var/lib/kubelet/config.yaml"
# This is a file that "kubeadm init" and "kubeadm join" generates at runtime, populating the KUBELET_KUBEADM_ARGS variable dynamically
EnvironmentFile=-/var/lib/kubelet/kubeadm-flags.env
# This is a file that the user can use for overrides of the kubelet args as a last resort. Preferably, the user should use
# the NodeRegistration.KubeletExtraArgs object in the configuration files instead. KUBELET_EXTRA_ARGS should be sourced from this file.
EnvironmentFile=-/etc/default/kubelet
ExecStart=
ExecStart=/usr/bin/kubelet $KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS $KUBELET_KUBEADM_ARGS $KUBELET_EXTRA_ARGS
 --cloud-provider=aws
```

Once the configuration has been made, reset the kubelet daemon.

`systemctl daemon-reload`

# Bootstrap the First K8s Control Plane Node

The time has come to setup the cluster. Login to one of your control plane nodes which will become the first master in the cluster. We'll run the kubeadm initialization with the kubeadm.conf file that we created earlier and placed in the /etc/kubernetes directory.

`kubeadm init --config /etc/kubernetes/kubeadm.conf --upload-certs`

It may take a bit for the process to complete. Kubeadm init is ensuring that our api-server, controller-manager, and etcd container images are downloaded as well as creating certificates which you should find in the /etc/kubernetes/pki directory.

When the process is done, you should receive instructions on how to add additional control plane nodes and worker nodes.

```
You can now join any number of the control-plane node running the following command on each as root:

  kubeadm join cp.theithollowlab.com:6443 --token c5u9ax.ua2896eckzxms16p \
    --discovery-token-ca-cert-hash sha256:ceaa3e06e246228bbca02cc9b107070bf42383778a66b91c0bc3fecaac1fd26b \
    --control-plane --certificate-key 82b68a17877ab24c705b427d41373068482c7e6bc07b28ae8e63062019371989

Please note that the certificate-key gives access to cluster sensitive data, keep it secret!
As a safeguard, uploaded-certs will be deleted in two hours; If necessary, you can use
"kubeadm init phase upload-certs --upload-certs" to reload certs afterward.

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join cp.theithollowlab.com:6443 --token c5u9ax.ua2896eckzxms16p \
    --discovery-token-ca-cert-hash sha256:ceaa3e06e246228bbca02cc9b107070bf42383778a66b91c0bc3fecaac1fd26b
```

# Add Additional Control Plane Nodes

We can now take the information provided from our init command and run the kubeadm join provided in our output, on the other two control plane nodes.

Before we add those additional control plan nodes, you'll need to copy the contents of the pki directory to the other control plane nodes. This is needed because they need those certificates for authentication purposes with the existing control plane node.

Your instructions are going to be different. The command below was what was provided to me.

```
kubeadm join cp.theithollowlab.com:6443 --token c5u9ax.ua2896eckzxms16p
\
    --discovery-token-ca-cert-hash
sha256:ceaa3e06e246228bbca02cc9b107070bf42383778a66b91c0bc3fecaac1fd26
b \
    --control-plane                                    --certificate-key
82b68a17877ab24c705b427d41373068482c7e6bc07b28ae8e63062019371989
```

Code language: CSS (css)

When you're done with your additional control plane nodes, you should see a success message with some instrucitons on setting up the KUBECONFIG file, which we'll cover later.

# Join Worker Nodes to the Cluster

At this point we should have three control plane nodes working in our cluster. Let's add the worker nodes now by using the other kubeadm join command presented to use after setting up our first control plane node.

Again, yours will be different, but for example purposes mine was:

```
kubeadm join cp.theithollowlab.com:6443 --token c5u9ax.ua2896eckzxms16p
\
    --discovery-token-ca-cert-hash
sha256:ceaa3e06e246228bbca02cc9b107070bf42383778a66b91c0bc3fecaac1fd26
b
```

Code language: CSS (css)

# Setup KUBECONFIG

Log back into your first Kubernetes control plane node and we'll setup KUBECONFIG so we can issue some commands against our cluster and ensure that it's working properly.

Run the following to configure your KUBECONFIG file for use:

```
export KUBECONFIG=/etc/kubernetes/admin.conf
```

Code language: JavaScript (javascript)

When you're done, you can run:

```
kubectl get nodes
```

Code language: JavaScript (javascript)

We can see here that we have a cluster created, but the status is not ready. This is because we're missing a CNI.

```
root@ip-10-0-1-10:/home/ubuntu# kubectl get nodes
NAME                                         STATUS     ROLES
ip-10-0-1-10.us-east-2.compute.internal   NotReady   master
ip-10-0-1-20.us-east-2.compute.internal   NotReady   <none>
ip-10-0-2-10.us-east-2.compute.internal   NotReady   master
ip-10-0-2-20.us-east-2.compute.internal   NotReady   <none>
ip-10-0-3-10.us-east-2.compute.internal   NotReady   master
ip-10-0-3-20.us-east-2.compute.internal   NotReady   <none>
```

# Deploy a CNI

There are a variety of Networking interfaces that could be deployed. For this simple example I've used calico. Simply apply this manifest from one of your nodes.

```
kubectl apply -f https://docs.projectcalico.org/v3.9/manifests/calico.yaml
```

Code language: JavaScript (javascript)

When you're done, you should have a working cluster.

```
NAME                                         STATUS   ROLES
ip-10-0-1-10.us-east-2.compute.internal   Ready    master
ip-10-0-1-20.us-east-2.compute.internal   Ready    <none>
ip-10-0-2-10.us-east-2.compute.internal   Ready    master
ip-10-0-2-20.us-east-2.compute.internal   Ready    <none>
ip-10-0-3-10.us-east-2.compute.internal   Ready    master
ip-10-0-3-20.us-east-2.compute.internal   Ready    <none>
```

# Deploy an AWS Storage Class

If you wish to use EBS volumes for your Persistent Volumes, you can apply a storage class manifest such as the following:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/kubernetes/master/cluster/addons/storage-class/aws/default.yaml
```

Code language: JavaScript (javascript)

# Common Errors

There are some tricky places along the way, below are a few common issues you may run into when standing up your Kubernetes cluster.

## Missing EC2 Cluster Tags

If you've forgotten to add the kubernetes.io/cluster/<CLUSTERNAME> tag to your EC2 instances, kubeadm will fail stating issues with the kubelet.

```
[kubelet-check] The HTTP call equal to 'curl -sSL http://localhost:10248/healthz' failed with error: Get http://
localhost:10248/healthz: dial tcp 127.0.0.1:10248: connect: connection refused.

Unfortunately, an error has occurred:
        timed out waiting for the condition

This error is likely caused by:
        - The kubelet is not running
        - The kubelet is unhealthy due to a misconfiguration of the node in some way (required cgroups disabled)

If you are on a systemd-powered system, you can try to troubleshoot the error with the following commands:
        - 'systemctl status kubelet'
        - 'journalctl -xeu kubelet'
```

The kubelet logs will include something like the following:

Tag "KubernetesCluster" nor "kubernetes.io/cluster/..." not found; Kubernetes may behave unexpectedly.
... failed to run Kubelet: could not init cloud provider "aws": AWS cloud failed to find ClusterID

## Missing Subnet Cluster Tags

If you've forgotten to add the kubernetes.io/cluster/<clustername> tag to your subnets, then any LoadBalancer resources will be stuck in a "pending" state.

```
NAME      TYPE          CLUSTER-IP      EXTERNAL-IP   PORT(S)                      AGE
contour   ClusterIP     10.98.197.40    <none>        8001/TCP                     4m14s
envoy     LoadBalancer  10.98.86.120    <pending>     80:32467/TCP,443:30868/TCP   4m14s
```

The controller managers will throw errors about a missing tags on the subnets.

failed to ensure load balancer: could not find any suitable subnets for creating the ELB

## Node Names Don't Match Private DNS Names

If your hostname and private dns names don't match, you'll see error messages during the kubeadm init phase.

Error writing Crisocket information for the control-plane node: timed out waiting for the condition.

The kubelet will show error messages about the node not being found.

```
node "ip-10-0-1-10.us-east-2.compute.internal" not found
node "ip-10-0-1-10.us-east-2.compute.internal" not found
```

To fix this, update the nodes hostnames so that they match the private dns names.

**Summary**

Once you have completed the above sections, you should have a functional Kubernetes cluster that can take advantage of things like Load Balancers and EBS volumes through your Kubernetes API calls. Have fun!