

Chapter 4:

Kubernetes API Overview and resources



Table of Content

- Pod
- Controllers – ReplicaSet, ReplicationController, Deployment
- Services
- Storage and Volumes
- StorageClass
- Labels
- Namespaces
- Secret
- ConfigMap
- DaemonSet
- Job and CronJob
- Pod Priority and Pre-emption
- Node Role and Taints
- Resource Quotas
- Authorization and Authentication



Pods



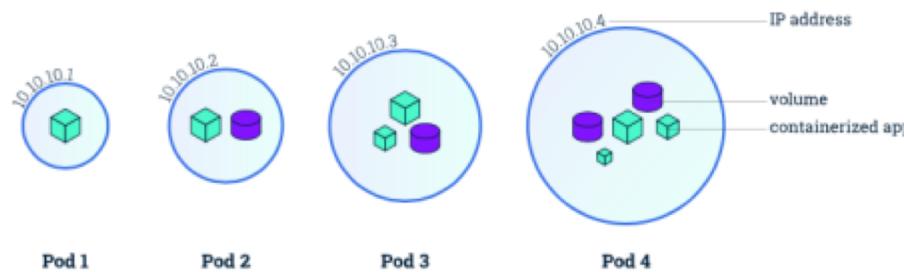
Agile Brains Consulting

©Agile Brains Consulting Inc. All rights reserved. Not to be reproduced without prior written consent.



Kubernetes: Pods

- The smallest unit that can be scheduled to be deployed through K8s is called a *pod*.
- Homogeneous group of containers.
- This group of containers would share storage, Linux namespaces, cgroups, IP addresses. These are co-located, hence share resources and are always scheduled together.
- Pods are not intended to live long. They are created, destroyed and re-created on demand, based on the state of the server and the service itself.



- Containers in a pod share
 - IP address and port space
 - Filesystem
 - Storage (Volumes)
 - Labels
 - Secrets



Kubernetes: Pods

Pods Are...

- Ephemeral, disposable
- Never self-heal, and not restarted by the scheduler by itself
- Never create pods just by themselves
- Always use higher-level constructs



Kubernetes: Pod States

- **Pending** (request accepted, one or more containers are still being created)
- **Running** (Pod has been bound to a node, all of the Containers have been created. At least one Container is still running, or is in the process of starting or restarting.)
- **Succeeded** (All Containers in the Pod have terminated in success, and will not be restarted)
- **Failed** (All Containers in the Pod have terminated, and at least one container has terminated with some failure)
- **CrashLoopBackOff** (pod starting, crashing, starting again, and then crashing again.)



pod.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

pod.json

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {"name": "nginx", "labels": {"name": "nginx"}},
  "spec": {"containers": [{"name": "nginx", "image": "nginx", "ports": [{"containerPort": 80}]}]}
```

*** Delete this pod, it **won't** get re-created automatically*

```
[root@ip-172-31-32-172:~/code# kubectl get pods -o wide
No resources found.
[root@ip-172-31-32-172:~/code# kubectl create -f pod.json
pod/nginx created
[root@ip-172-31-32-172:~/code# kubectl get pods -o wide
NAME      READY     STATUS    RESTARTS   AGE      IP           NODE      NOMINATED NODE   READINESS GATES
nginx    1/1      Running   0          7s      10.32.0.4   ip-172-31-32-172   <none>        <none>
[root@ip-172-31-32-172:~/code# kubectl delete -f pod.json
pod "nginx" deleted
[root@ip-172-31-32-172:~/code# kubectl get pods -o wide
No resources found.
root@ip-172-31-32-172:~/code# ]
```



Controllers



Agile Brains Consulting

©Agile Brains Consulting Inc. All rights reserved. Not to be reproduced without prior written consent.



Controllers

- Kubernetes contains a number of higher-level abstractions called **Controllers**. Controllers build upon the basic objects, and provide additional functionality and convenience features.

They are

- **ReplicationController**
- **ReplicaSet**
- **Deployment**
- DaemonSet – used for specific purpose
- Job
- CronJob



Replication Controller

- The original form of replication in Kubernetes. Its now preferred to use Replicate Sets via a Deployment
- Ensures that a specified number of pod replicas are running at any one time. If a pod does crash, the Replication Controller replaces it.
- Allows you to scale the number of pods, and to update or delete multiple pods with a single command.



Replication Controller

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: testrc
spec:
  replicas: 3
  selector:
    app: webserver
  template:
    metadata:
      name: testrc
      labels:
        app: webserver
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

- This file says:
 - We need to have 3 replicas of this Pod
 - The Pod has 1 container (image: nginx)
- Create the Replication controller:
 - `kubectl create -f rc.yaml`



Replication Controller

```
$ kubectl create -f rc.yml
replicationcontroller/testrc created
$ kubectl describe rc testrc
Name:           testrc
Namespace:      default
Selector:       app=webserver
Labels:         app=webserver
Annotations:    <none>
Replicas:       3 current / 3 desired
Pods Status:   3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=webserver
  Containers:
    nginx:
      Image:      nginx:latest
      Port:       80/TCP
      Host Port:  0/TCP
      Environment: <none>
      Mounts:     <none>
      Volumes:    <none>
  Events:
    Type  Reason          Age   From            Message
    ----  -----          ----  ----            -----
    Normal SuccessfulCreate 24s   replication-controller  Created pod: testrc-nwh7b
    Normal SuccessfulCreate 24s   replication-controller  Created pod: testrc-mq599
    Normal SuccessfulCreate 24s   replication-controller  Created pod: testrc-z2js6
$ 
```



Replica-set



Agile Brains Consulting

©Agile Brains Consulting Inc. All rights reserved. Not to be reproduced without prior written consent.



Replica Set

- It is essentially the same as Replication Controller except that it has more options for selector:
 - matchLabels
 - matchExpressions
- Replica set doesn't support rolling update, instead use deployment



Replica Set

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: testrs
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webserver
  template:
    metadata:
      name: testrs
      labels:
        app: webserver
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

- This Replica Set has the same rules as the Replication Controller in the previous slide.
- This file says:
 - We need to have 3 replicas of this Pod
 - The Pod has 1 container (image: nginx)
- Create the Replication controller:
 - kubectl create -f rs.yaml



Replica Set

```
$ kubectl create -f rs.yml
replicaset.apps/testrs created
$ kubectl describe rs testrs
Name:           testrs
Namespace:      default
Selector:       app=webserver
Labels:         <none>
Annotations:   <none>
Replicas:      3 current / 3 desired
Pods Status:   3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=webserver
  Containers:
    nginx:
      Image:      nginx:latest
      Port:       80/TCP
      Host Port:  0/TCP
      Environment: <none>
      Mounts:     <none>
      Volumes:    <none>
  Events:
    Type  Reason          Age   From            Message
    ----  ----          ----  ----            -----
    Normal SuccessfulCreate 35s   replicaset-controller  Created pod: testrs-5bgpg
    Normal SuccessfulCreate 35s   replicaset-controller  Created pod: testrs-c775f
    Normal SuccessfulCreate 35s   replicaset-controller  Created pod: testrs-z5rxh
$
```



Replica Set – match labels and expressions

```
spec:  
  replicas: 3  
  selector:  
    matchLabels:  
      app: soaktestrs
```

```
spec:  
  replicas: 3  
  selector:  
    matchExpressions:  
      - {key: app, operator: In, values: [soaktestrs, soaktestrs,  
      - {key: teir, operator: NotIn, values: [production]}}
```



Deployments



Agile Brains Consulting

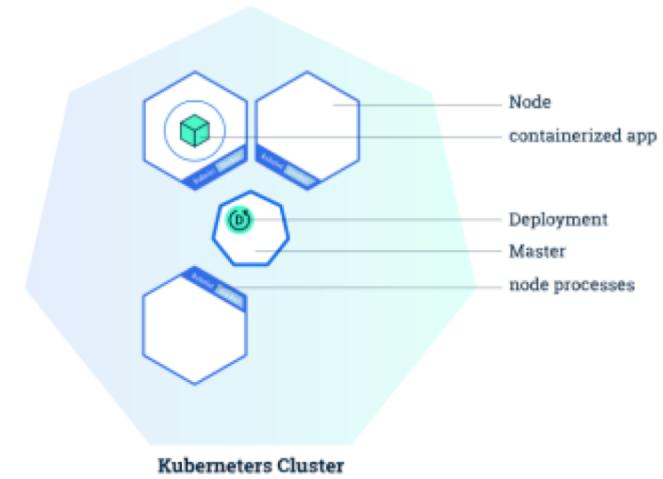
©Agile Brains Consulting Inc. All rights reserved. Not to be reproduced without prior written consent.



Kubernetes: Deployments

Allows you to deploy a (self-healing) instance of an application.

- Self Healing: Continuously monitors and replaces instances if necessary
- Provides declarative updates for Pods and Replica Sets
 - Updates actual state to desired state at a controlled rate
 - For example: Current state is 3 instances of Tomcat. Desired state is 5 instances of Tomcat -> Create 2 more instances of Tomcat



dep.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

```
[root@ip-172-31-68-96:~/code# kubectl create -f dep.yml
deployment.apps/nginx-deployment created
[root@ip-172-31-68-96:~/code# kubectl get pods,deployments
NAME                                READY   STATUS    RESTARTS   AGE
pod/nginx-apparmor                   1/1    Running   0          52s
pod/nginx-deployment-67594d6bf6-7jc78 1/1    Running   0          11s
pod/nginx-deployment-67594d6bf6-tf5df 1/1    Running   0          11s
pod/nginx-deployment-67594d6bf6-xzwww 1/1    Running   0          11s

NAME                           DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
deployment.extensions/nginx-deployment  3        3        3           3          11s
root@ip-172-31-68-96:~/code# ]
```



Kubectl create/apply

- `kubectl create -f <yaml_file>`
- `kubectl create -f <yaml_file_1> -f <yaml_file_2>`
- `kubectl create -f <json_file>`
- `kubectl create -f <url_name>`
- `kubectl create -f <directory_name>`
- **kubectl create** tell the Kubernetes API what you want to create, replace or delete. It's a imperative approach and good if single person is working in the team.
- **kubectl apply** – apply changes to a live object. It's a declarative approach and good when you want to maintain versions



Lab (rc, rs, deploy)

- Exercise to create Replication Controllers, Replica Sets and Deployments and describe their functions
- Check the results and describe the objects

sudo su -

git clone https://github.com/shekhar2010us/kubernetes_teach_git.git

cd kubernetes_teach_git/ex1/

https://github.com/shekhar2010us/kubernetes_teach_git/blob/master/ex1/rc_rs_deployment.md



Services



Agile Brains Consulting

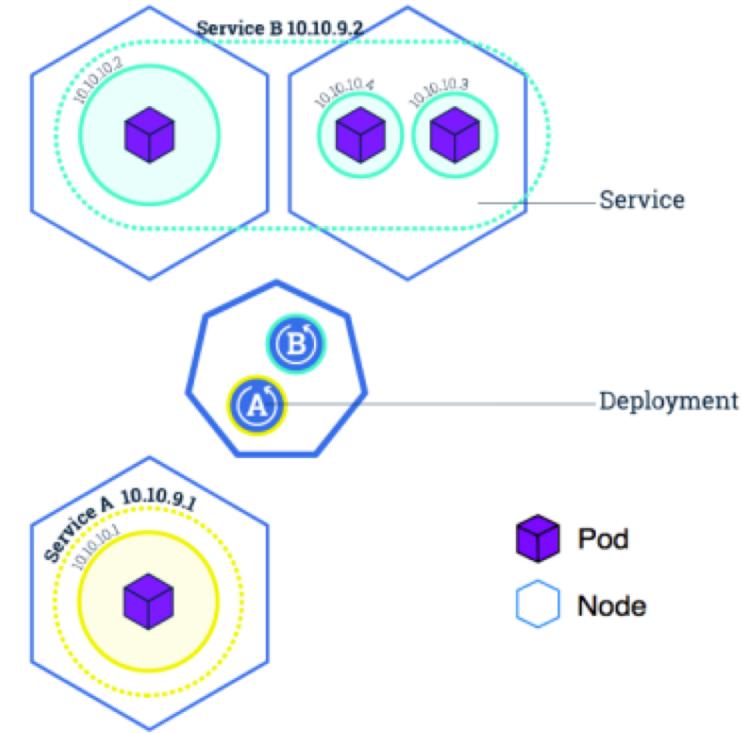
©Agile Brains Consulting Inc. All rights reserved. Not to be reproduced without prior written consent.



Kubernetes: Service

Abstraction regarding a set of pods which enables load balancing, traffic exposure, load balancing and service discovery.

- pods to die and replicate in Kubernetes without affecting your application.
- Enable loose coupling between different Pods.
- Provides a **stable** virtual IP and port
- Services allow Pods to receive traffic.
 - Each Pod has a unique IP but those IP addresses are not exposed outside the Pod without a service.



Kubernetes: Types of Services

- **Cluster IP:** Expose the service on a cluster-internal IP. Using this makes the service only reachable from within the cluster (no external access).
- **NodePort:** Expose the service on each Node's IP at a static port. One service per port, only use ports 30000–32767, have to deal with port changes
- **Load Balancer:** Expose the service externally using load balancer. The Kubernetes service controller automates the creation of the external load balancer, health checks (if needed), firewall rules (if needed) and retrieves the external IP allocated by the cloud provider and populates it in the service object



```

root@ip-172-31-68-96:~# kubectl get svc
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
kubernetes  ClusterIP  10.96.0.1    <none>        443/TCP      37m
root@ip-172-31-68-96:~#
root@ip-172-31-68-96:~# kubectl expose deployment nginx-deployment --type=LoadBalancer --name=nginx-service
service/nginx-service exposed
root@ip-172-31-68-96:~#
root@ip-172-31-68-96:~# kubectl get svc
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
kubernetes  ClusterIP  10.96.0.1    <none>        443/TCP      37m
nginx-service  LoadBalancer  10.102.89.83  <pending>    80:31095/TCP  3s
root@ip-172-31-68-96:~#

```



Kubectl Create Services – using CLI

-- Create directly a deployment from CLI without the config file (**Object management using Imperative Commands** - e.g. run , expose, autoscale)

```
$ kubectl run nginx-deployment --image nginx --port 80
```

```
$ kubectl get nodes
```

```
$ kubectl get po, deploy, rs
```

-- **exposing the deployment as a service** LoadBalancer type and access the Nginx

```
$ kubectl expose deploy/nginx-deployment --type=LoadBalancer --name nginx-service
```

```
$ kubectl get all (command shows all)
```

```
$ kubectl get deploy/nginx-deployment -o yaml (returns the yaml that creates the deployment)
```

-- then you can describe the service and do a curl , on port 80 with internal IP or external IP and the exposed port outside

```
$ kubectl expose pod pod_name --type NodePort --name pod_name-service --port 30006
```



Kubectl Create Services – using Config

-- Create a deployment with the config file (Object management using **Imperative configuration files**)

```
$ kubectl create -f red.yaml  
$ kubectl get po, deploy, rs
```

-- Create Kubernetes Object using **Declarative management** of objects (The **kubectl apply** command calculates the patch request using the configuration file, the live configuration, and the **last-applied-configuration** annotation stored in the live configuration)

```
$ kubectl apply -f <file_name>  
$ kubectl get po, deploy, rs  
edit the file , and try the apply again  
$ kubectl apply -f <file_name>  
$ kubectl get po, deploy, rs
```



Lab (Pods, Deploy, Services)

- For all the concepts discussed until now
- Create pods, Services, Deployments
- Various ways to use the create command

We already have the git repo cloned in our machines.

\$\$ cd to the ex2

https://github.com/shekhar2010us/kubernetes_teach_git/blob/master/ex2/pod_deploy_service.md



Kubernetes Popular commands

Popular commands

- **kubectl get** - list resources about a target.
 - kubectl get services, kubectl get pods
- **kubectl describe** - show basic information about a resource
 - Kubectl describe pods my-pod
- **kubectl logs** - print the logs from a container in a pod. Extremely useful
 - Kubectl logs my-pod
- Also useful:
 - **kubectl exec** - execute a command on a container in a pod
 - **Kubectl top** – Show metrics for a node (needs Heapster)



Storage



Kubernetes: Volumes

On-disk files in a container are the simplest place for an application to write data, but this approach has drawbacks. The files are lost when the container crashes or stops for any other reason. Furthermore, files within a container are inaccessible to other containers running in the same [Pod](#). The [Kubernetes Volume](#) abstraction addresses both of these issues.

- A directory accessible to all containers in a pod
- Volumes out live containers that run within a pod
- May be passed between pods
- Data is preserved across container restarts



Kubernetes: Persistent Volume

Persistent storage in a cluster

- Separate from **PersistentVolumeClaim**, which is a request for storage.
- May be created ahead of time in a static manner for use by a cluster
- May be created dynamically from available, unclaimed resources
- May be freed and passed from user to user so care should be taken to delete contents when done with use



Kubernetes: Persistent Volumes

PV: Storage in the cluster that has been provisioned by an administrator (static or dynamic) and is independent of any individual pod that uses the PV.

Following volumes are supported by Kubernetes:

GCEPersistentDisk ; AWSElasticBlockStore

AzureFile ; AzureDisk

FC (Fibre Channel) ; FlexVolume

Flocker ; NFS

iSCSI ; RBD (Ceph Block Device)

CephFS ; Cinder (OpenStack block storage)

Glusterfs ; VsphereVolume

Portworx Volumes ; ScaleIO Volumes

StorageOS



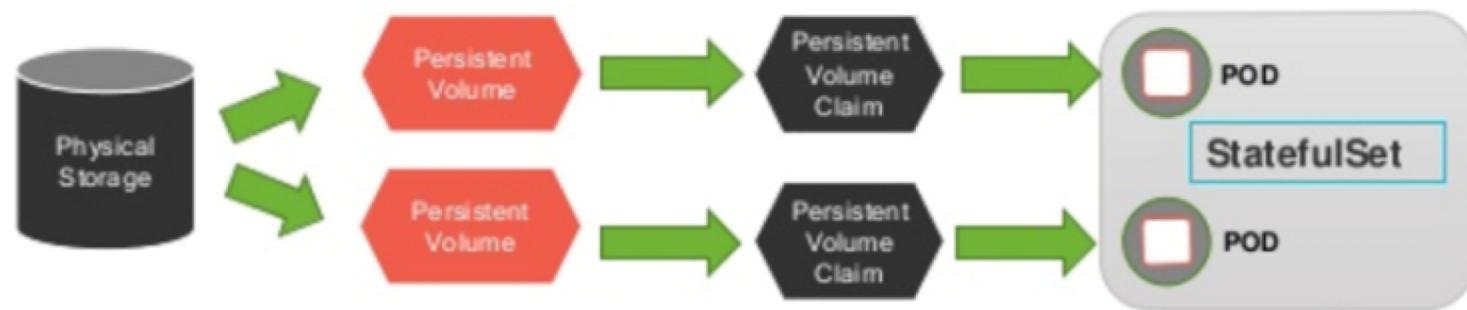
Example: Persistent Volumes

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /tmp
    server: 172.17.0.2
```



Kubernetes: Persistent Volume Claims

- PVC: Request for a storage by a user, it is similar to pod and it can request specific size and access modes (e.g., can be mounted once read/write or many times read-only).



Example: Persistent Volume Claim

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: user-service-war-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```



Example: Deployment using PVC

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: userservice
spec:
  replicas: 1
  template:
    .....
    spec:
      containers:
        .....
        volumeMounts:
          - mountPath: /usr/local/tomcat/webapps/user-service-1.0.war
            name: user-service-war
      volumes:
        - name: user-service-war
      persistentVolumeClaim:
        claimName: user-service-war-pv-claim
```



Lab (Volume, PV, PVC)

- create pods,PV and PVC
- Experiment with creating end to end application, mount volume, read from shared pod
- Fix a bug

cd to ex6

6.1 Exercise of Volume

https://github.com/shekhar2010us/kubernetes_teach_git/blob/master/ex6/create_volumes.md

6.2 Exercise on PV, PVC

https://github.com/shekhar2010us/kubernetes_teach_git/blob/master/ex6/create_pv.md



Storage Classes

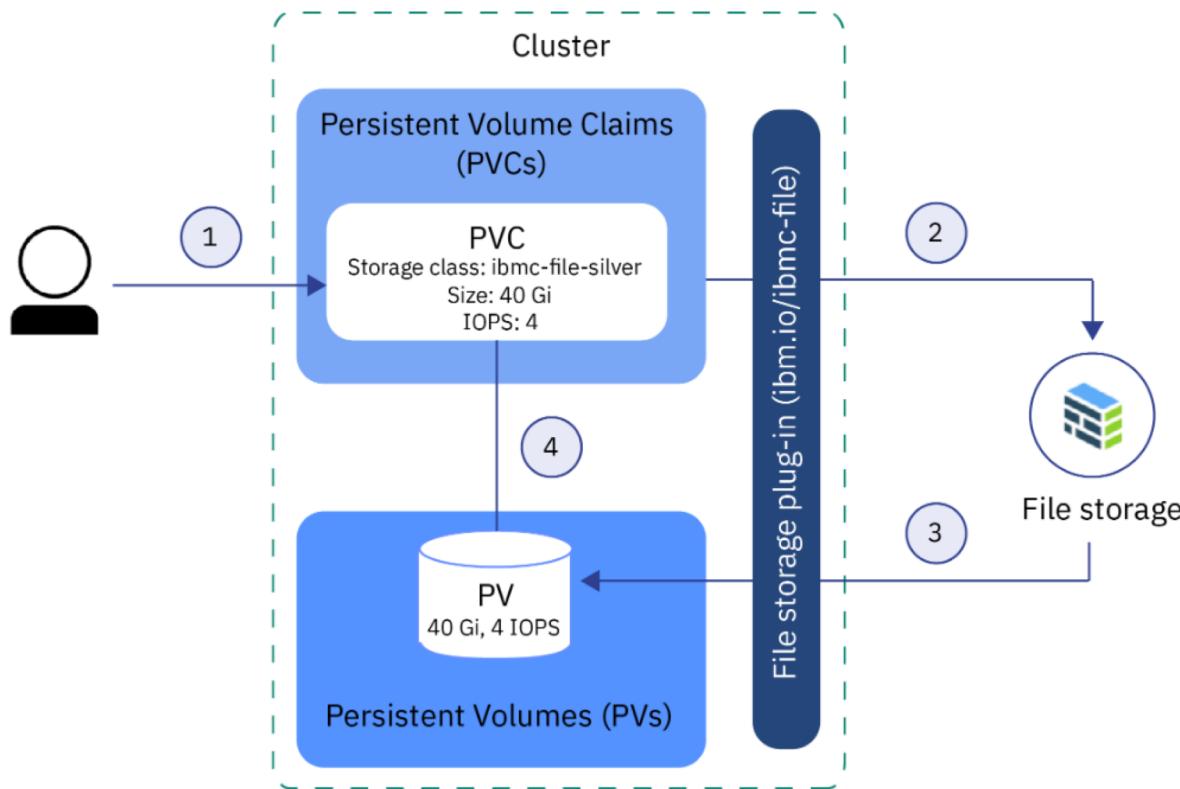


Kubernetes: Storage Classes

- StorageClass provides a way for administrators to describe the “**classes**” of storage they offer. Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators.
- This concept is sometimes called “profiles” in other storage systems.
- Usually required for **dynamic provisioning** of storage (usually done in cloud storage)



Dynamic Storage Provisioning



1. User creates **PVC**, specifying storage type, storage class, size, etc. The **storage class** determines types of storage. Creating PVC triggers the storage plug-in to provision storage
2. The storage device is ordered and provisioned into the Cloud account.
3. The storage plug-in creates a **PV** in the cluster, a virtual storage device that points to the actual storage device in your Cloud infrastructure
4. The PVC and PV are automatically connected to each other. The status of the PVC and the PV changes to **Bound**. If PVC is deleted, PV and related storage instance are also deleted.

Dynamic Storage Provisioner Options

Volume Plugin	Internal Provisioner	Config Example
AWSElasticBlockStore	✓	AWS EBS
AzureFile	✓	Azure File
AzureDisk	✓	Azure Disk
CephFS	-	-
Cinder	✓	OpenStack Cinder
FC	-	-
Flexvolume	-	-
Flocker	✓	-
GCEPersistentDisk	✓	GCE PD
Glusterfs	✓	Glusterfs
iSCSI	-	-
Quobyte	✓	Quobyte
NFS	-	-
RBD	✓	Ceph RBD
VsphereVolume	✓	vSphere
PortworxVolume	✓	Portworx Volume
ScaleIO	✓	ScaleIO
StorageOS	✓	StorageOS
Local	-	Local



Storage Class Examples

Local

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: local-storage
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

Provisioner – provider

Parameters – details of storage

VolumeBindingMode – When volume binding and dynamic provisioning should occur

AWS EBS

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/aws-ebs
parameters:
  type: io1
  iopsPerGB: "10"
  fsType: ext4
```

AWS EBS

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Retain
volumeBindingMode: Immediate
```



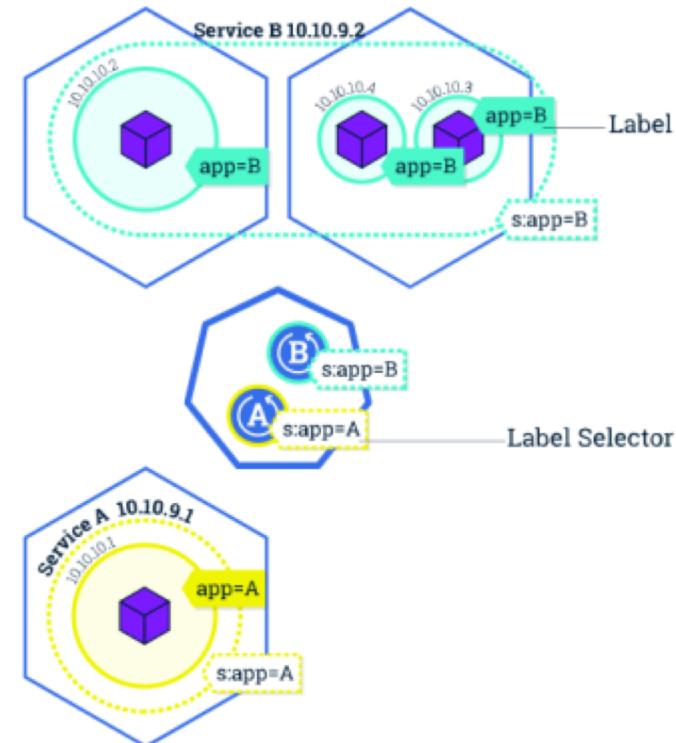
Labels and Namespaces



Kubernetes: Label

Key/Value pairs that can be attached to objects to enable a variety of use cases.

- Designate objects for specific environments such as development, test and production
- Set versions to objects
- Set roles or other arbitrary information to classify objects
- Can be added or modified at any time



\$ kubectl get pod -l name=<label>

```
peter@Azure:~$ kubectl run nginx --image nginx --port 80
deployment "nginx" created
peter@Azure:~$ kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
nginx-158599303-04h8d  1/1     Running   0          4s
peter@Azure:~$ kubectl label pods nginx-158599303-04h8d new-label=awesome
pod "nginx-158599303-04h8d" labeled
peter@Azure:~$ kubectl get pods -l new-label=else
No resources found.
peter@Azure:~$ kubectl get pods -l new-label=awesome
NAME           READY   STATUS    RESTARTS   AGE
nginx-158599303-04h8d  1/1     Running   0          3m
peter@Azure:~$ kubectl get pods --show-labels
NAME           READY   STATUS    RESTARTS   AGE   LABELS
nginx-158599303-04h8d  1/1     Running   0          3m   new-label=awesome,pod-template-hash=158599303,run=nginx
```

Note: Best mechanism to organize Kubernetes objects

- labels can be used to provide logical structure
- divide by teams, or by environments, or versions
- annotations and labels have very subtle difference
- \$ kubectl create -f helloworld-pod-with-labels.yml
- \$ kubectl get po --show-labels



Lab (Kubernetes Labels)

- Create a series of pods with different labels
- Query the pods based on various labels
- Delete pods based on certain selectors

We have already pulled the git repo

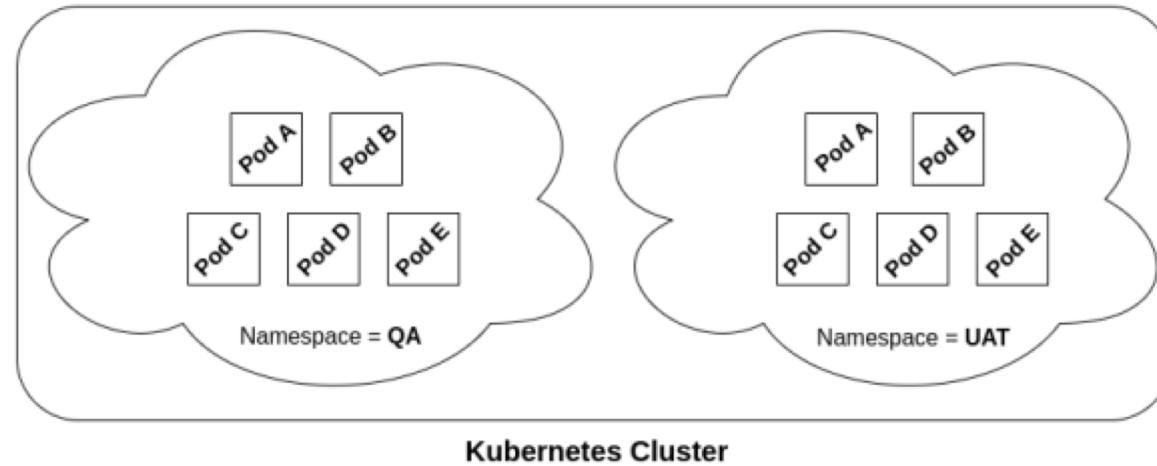
Now cd to ex3

https://github.com/shekhar2010us/kubernetes_teach_git/blob/master/ex3/label_usage.md



Kubernetes: Namespace

- Resources are segmented within namespaces
 - Sandbox per developer for example
 - Pods will route dns to default (own) namespace
 - The same app could run independently in different namespaces
 - Namespaces create an excellent mechanism to subdivide resources and provide isolation. This enables using a cluster for multiple projects or multiple teams



Kubernetes: Namespace

- \$ kubectl get po,deploy,svc,rs --all-namespaces
- \$ kubectl create -f <pod_config> -n <namespace_name>
- \$ kubectl get po -o wide
- \$ kubectl get po -o wide -n <namespace_name>
- \$ kubectl delete po simple_pod
- \$ kubectl delete po simple_pod -n <namespace_name>

```
peter@Azure:~$ kubectl get ns
NAME      STATUS   AGE
default   Active   23d
kube-public   Active   23d
kube-system   Active   23d
peter@Azure:~$ kubectl create -f https://raw.githubusercontent.com/mhausenblas/kbe/master/specs/ns/ns.yaml
W1006 13:59:41.630630    134 factory_object_mapping.go:423] Failed to download OpenAPI (the server could not find the requested resource), falling back to swagger
namespace "test" created
peter@Azure:~$ kubectl get ns
NAME      STATUS   AGE
default   Active   23d
kube-public   Active   23d
kube-system   Active   23d
test      Active   19s
peter@Azure:~$ kubectl create --namespace=test -f https://raw.githubusercontent.com/mhausenblas/kbe/master/specs/ns/pod.yaml
W1006 14:00:18.027941    145 factory_object_mapping.go:423] Failed to download OpenAPI (the server could not find the requested resource), falling back to swagger
pod "podintest" created
peter@Azure:~$ kubectl get pods --namespace=test
NAME      READY   STATUS      RESTARTS   AGE
podintest  0/1     ContainerCreating   0          11s
```



Lab (Namespace)

- Create a namespace
- check various objects in different namespaces
- Can create or delete objects from a specific namespace

cd to ex12

```
# create two namespaces
```

```
kubectl create -f dev-ns.yaml
```

```
kubectl create -f prod-ns.yaml
```

```
# create two pods with 2 namespaces
```

```
kubectl create -f dev-pod.yaml -n dev
```

```
kubectl create -f prod-pod.yaml -n prod
```

```
# Search for pods
```

```
kubectl get pods
```

```
# show only default ns pods
```

```
kubectl get pods -n dev
```

```
# show dev pods
```

```
kubectl get pods -n prod
```

```
# show prod pods
```



Secrets



Agile Brains Consulting

©Agile Brains Consulting Inc. All rights reserved. Not to be reproduced without prior written consent.



Kubernetes: Secrets

- Kubernetes secret objects let you store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys.
- Using secret to store this information is safer and more flexible than putting it verbatim in a container image or Pod Lifecycle definition.



Create Secrets – using “kubectl create secret”

- Create files with the content:

```
echo -n 'admin' > ./username.txt  
echo -n '1f3wr8sdfff' > ./password.txt
```

From file

- Creating secrets from the file:

```
$ kubectl create secret generic db-user-pass --from-file=./username.txt --from-file=./password.txt
```

```
$ kubectl create secret generic test-secret --from-literal=username='my-app' --from-literal=password='39528$vdg7Jb'
```

From literal



Create Secrets – manually Using YAML

- Creating base64 encoded Secrets:

```
$ echo -n 'admin' | base64  
YWRtaW4=  
$ echo -n '1f2d1e2e67df' | base64  
MWYyZDFlMmU2N2Rm
```

secret.yaml

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: mysecret  
type: Opaque  
data:  
  username: YWRtaW4=  
  password: MWYyZDFlMmU2N2Rm
```

Creating secrets from the YAML file:

```
kubectl create -f ./secret.yaml
```

Retrieve values of secrets:

```
kubectl get secret mysecret -o yaml
```



Using Secrets

```
$ kubectl create secret generic mysql --from-literal=password=12345
```

```
spec:  
  containers:  
    - image: mysql:5.6  
      name: mysql  
    env:  
      - name: MYSQL_ROOT_PASSWORD  
        valueFrom:  
          secretKeyRef:  
            name: mysql  
            key: password
```



ConfigMap



Agile Brains Consulting

©Agile Brains Consulting Inc. All rights reserved. Not to be reproduced without prior written consent.



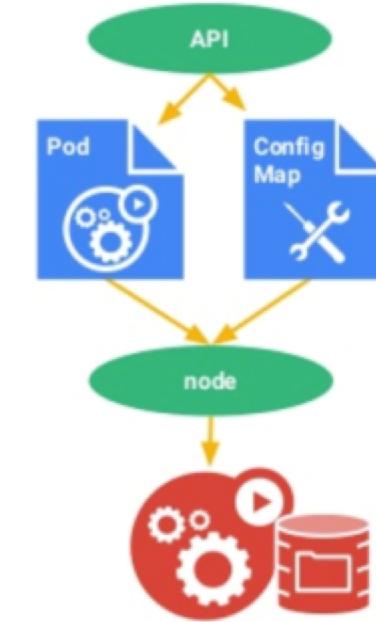
Kubernetes: Configmap

A set of key-value pairs that serve as configuration data for pods. They solve the problem of how to pass config data such as environment variables to pods.

Config maps are commonly composed of:

- Command line arguments
- Environment variables
- Files in a volume

Config maps function similar to secrets, but values are stored as strings and are more readily readable.



```
$ kubectl create -f etcd-config.yml  
  
apiVersion: extensions  
kind: ConfigMap  
metadata:  
  name: etcd-env-config  
data:  
  discovery_url: http://etcd-discovery:2379  
  etcdctl_peers: http://etcd:2379
```



Kubernetes: Config Map

```
[root@ip-172-31-68-96:~# kubectl create configmap test-config --from-literal=key1=config1 --from-literal=key2=config2
configmap/test-config created
[root@ip-172-31-68-96:~# kubectl get configmap
  NAME      DATA      AGE
  my-config   2        1m
  test-config  2        7s
[root@ip-172-31-68-96:~#
[root@ip-172-31-68-96:~# kubectl describe configmap/test-config
  Name:            test-config
  Namespace:       default
  Labels:          <none>
  Annotations:    <none>

  Data
  ====
  key1:
  ----
  config1
  key2:
  ----
  config2
  Events: <none>
root@ip-172-31-68-96:~# ]
```



Kubernetes: Config Map

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: test-cm
  labels:
    name: test-cm
data:
  key1: value1
  key2: value2
```

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-cm
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: test-cm
              key: key1
  restartPolicy: Never
```



Lab (Config Maps)

- Create a deployment which uses hard coded value for log-level
- Create a Config Map
- Use the Config map inside a deployment

cd to ex14

https://github.com/shekhar2010us/kubernetes_teach_git/blob/master/ex14/configmap.md



Daemonset



Agile Brains Consulting

©Agile Brains Consulting Inc. All rights reserved. Not to be reproduced without prior written consent.



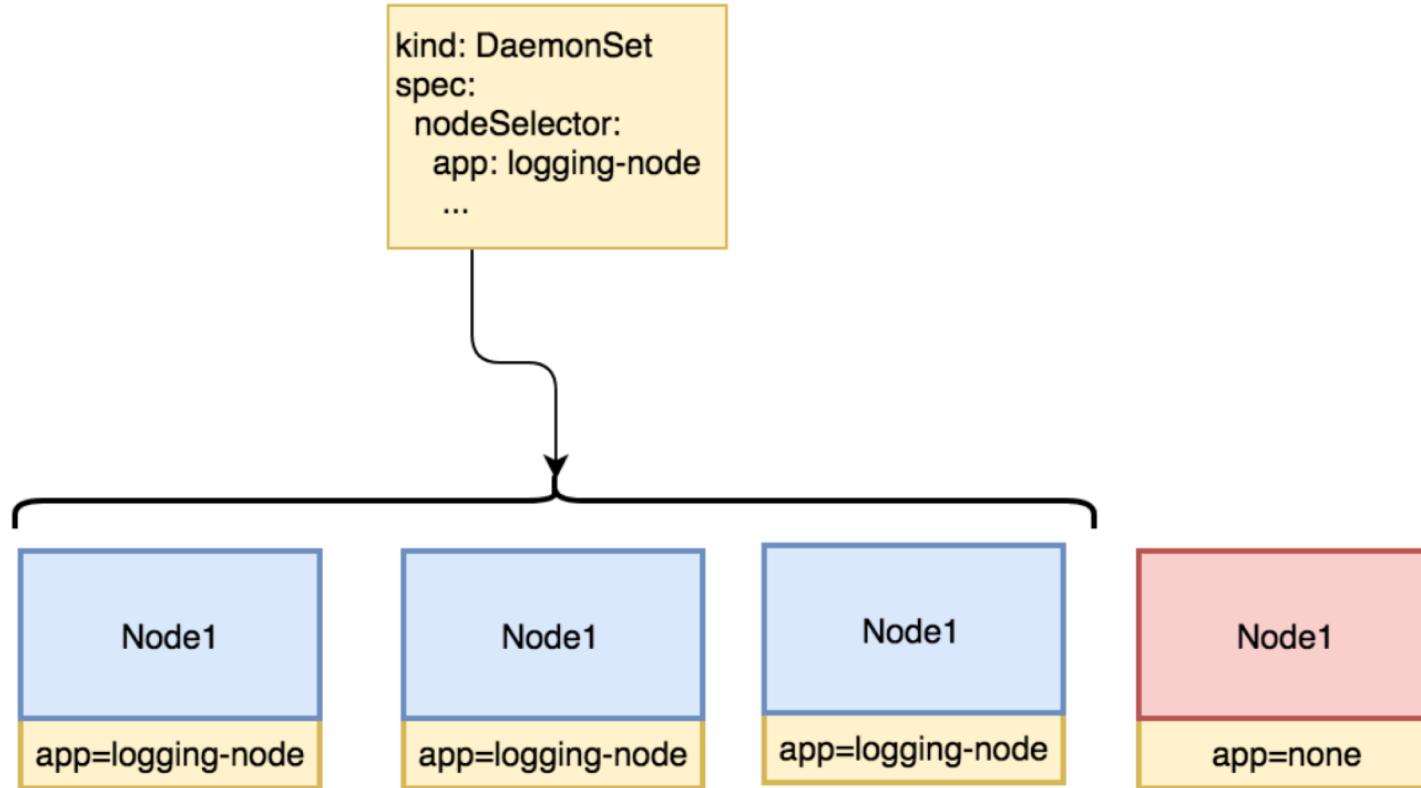
Kubernetes: DaemonSet

A **DaemonSet** ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

- Usually used for pods meant for Monitoring or collecting logs



Kubernetes: DaemonSet



DaemonSet example

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: logging
spec:
  template:
    metadata:
      labels:
        app: logging-app
  spec:
    nodeSelector:
      app: logging-node
    containers:
      - name: webserver
        image: nginx
        ports:
          - containerPort: 80
```

➤ Looks like pod

[How to assign labels to nodes?](#)

If node exists

kubectl label nodes <node-name> <label-key>=<label-value>

While provisioning

--node-labels <label-key>=<label-value>'



Cronjob



Agile Brains Consulting

©Agile Brains Consulting Inc. All rights reserved. Not to be reproduced without prior written consent.



Kubernetes: Jobs and Cron Jobs

Jobs - Creates one or more pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the *job* tracks the successful completions. When a specified number of successful completions is reached, the job itself is complete.

Deleting a Job will cleanup the pods it created, and is run **one-time**

Cron Job - A *Cron Job* creates [Jobs](#) on a time-based schedule. One Cron Job object is like one line of a *crontab* (cron table) file. It runs a job **periodically** on a given schedule



Kubernetes: Jobs and Cron Jobs

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi-with-timeout
spec:
  backoffLimit: 5
  activeDeadlineSeconds: 100
  template:
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
  restartPolicy: Never
```

```
...ash • python | ~ — -bash | ...-1:~ — -bash | ...0: ~ — -bash | ...ata — -bash | ...0: ~ — -bash | ...ntu — -bas
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hellocron
spec:
  schedule: "*/1 * * * *" #Runs every minute (cron syntax) or @hourly.
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hellocron
              image: busybox
              args:
                - /bin/sh
                - -c
                - date; echo Hello from your Kubernetes cluster
  restartPolicy: OnFailure #could also be Always or Never
  suspend: false #Set to true if you want to suspend in the future
Abhiks-MacBook-Pro:Exercise_Files abhikbanerjee$ █
```



Lab (Jobs and Cron Jobs)

- Create Job and check the logs
- Create a Cron Job,
- Edit the Cron Job

cd to ex9

https://github.com/shekhar2010us/kubernetes_teach_git/blob/master/ex9/cronjob.md



How to assign pods to nodes



Kubernetes: Assigning Pods to Nodes

You can constrain a pod to only be able to run on particular nodes or to prefer to run on particular nodes. There are several ways to do this, and the recommended approaches all use label selectors to make the selection.

- *nodeSelector* is the simplest recommended form of node selection constraint.
- *nodeSelector* is a field of PodSpec. It specifies a map of key-value pairs. For the pod to be eligible to run on a node, the node must have each of the indicated key-value pairs as labels (it can have additional labels as well). The most common usage is one key-value pair.



Kubernetes: nodeSelector Example

Step1: Label the nodes

kubectl label nodes <node-name> <label-key>=<label-value>

Eg:- kubectl label nodes kubernetes-foo-node-1.c.a-test.internal disktype=ssd

Step2: Assign the Pod creation on node with label
'disktype' and value 'ssd'

kubectl create -f pod-nginx.yaml

pod-nginx.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```



Pod Priority



Agile Brains Consulting

©Agile Brains Consulting Inc. All rights reserved. Not to be reproduced without prior written consent.



Kubernetes: Pod Priority

- Pods can have priority. Priority indicates the importance of a Pod relative to other Pods. **If a Pod cannot be scheduled, the scheduler tries to pre-empt (evict) lower priority Pods to make scheduling of the pending Pod possible.**



Kubernetes: Pod Priority

Set the priority for pod scheduling but cannot ask never to kill a pod

Example PriorityClass

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
    priorityClassName: high-priority
```

```
apiVersion: scheduling.k8s.io/v1alpha1
kind: PriorityClass
metadata:
  name: high-priority
  value: 1000000
  globalDefault: false
  description: "This priority class should be used for XYZ service pods only."
```



Pod Pre-emption



Agile Brains Consulting

©Agile Brains Consulting Inc. All rights reserved. Not to be reproduced without prior written consent.



Kubernetes: Pod Pre-emption

- When Pods are created, they go to a queue and wait to be scheduled. The scheduler picks a Pod from the queue and tries to schedule it on a Node. If no Node is found that satisfies all the specified requirements of the Pod, pre-emption logic is triggered for the pending Pod.
- The **disablePreemption** flag is set to **false** by default.



Kubernetes: Disabling Preemption

- Option available in component configs only.
- preemption is controlled by a kube-scheduler flag

Configuration file

```
apiVersion: componentconfig/v1alpha1
kind: KubeSchedulerConfiguration
algorithmSource:
  provider: DefaultProvider
...
disablePreemption: true
```

```
algorithmSource:
  provider: DefaultProvider
apiVersion: componentconfig/v1alpha1
clientConnection:
  acceptContentTypes: ""
  burst: 100
  contentType: application/vnd.kubernetes.protobuf
  kubeconfig: ""
  qps: 50
  disablePreemption: false
  enableContentionProfiling: false
  enableProfiling: false
  failureDomains: kubernetes.io/hostname,failure-domain.beta.kube
  hardPodAffinitySymmetricWeight: 1
  healthzBindAddress: 0.0.0.0:10251
  kind: KubeSchedulerConfiguration
  leaderElection:
    leaderElect: true
    leaseDuration: 15s
    lockObjectName: kube-scheduler
    lockObjectNamespace: kube-system
    renewDeadline: 10s
    resourceLock: endpoints
    retryPeriod: 2s
  metricsBindAddress: 0.0.0.0:10251
  schedulerName: default-scheduler
```



Node taints and node roles



Kubernetes: Node taints

- Node affinity is a property of pods that attracts them to a set of nodes (either as a preference or a hard requirement). Taints are the opposite – they allow a node to repel a set of pods.
- Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints.



Adding and removing a taint to node

- Tainting a node named ‘node1’

kubectl taint nodes node1 key=value:NoSchedule

- Taint effect NoSchedule means no Pod will be able to schedule onto node1 until it has matching toleration (key=value)

- Removing taint of the node ‘node1’

kubectl taint nodes node1 key=value:NoSchedule-



Kubernetes: Node role

- A node role is just a label with the format:
 - ***node-role.kubernetes.io/<role>***
- It can be added using ***kubectl label*** command
- Adding a role to node
 - ***kubectl label test.node.test.com node-role.kubernetes.io/worker***
- Removing a role from node
 - ***kubectl label test.node.test.com node-role.kubernetes.io/worker-***

Recall how to add a label:

`kubectl label nodes <node-name> <label-key>=<label-value>`



Resource quotas



Kubernetes: Resource Quotas

- When several users or teams share a cluster with a fixed number of nodes, there is a concern that one team could use more than its fair share of resources.
- Resource quotas are a tool for administrators to address this concern.
- It is defined by “**ResourceQuota**” object. It provides constraints that **limit aggregate resource consumption per namespace**.



Kubernetes: Compute Resource Quotas

Resource Name	Description
cpu	Across all pods in a non-terminal state, the sum of CPU requests cannot exceed this value.
limits.cpu	Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value.
limits.memory	Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value.
memory	Across all pods in a non-terminal state, the sum of memory requests cannot exceed this value.
requests.cpu	Across all pods in a non-terminal state, the sum of CPU requests cannot exceed this value.
requests.memory	Across all pods in a non-terminal state, the sum of memory requests cannot exceed this value.



Kubernetes: Storage Resource Quotas

Resource Name	Description
<code>requests.storage</code>	Across all persistent volume claims, the sum of storage requests cannot exceed this value.
<code>persistentvolumeclaims</code>	The total number of persistent volume claims that can exist in the namespace.
<code><storage-class-name>.storageclass.storage.k8s.io/requests.storage</code>	Across all persistent volume claims associated with the storage-class-name, the sum of storage requests cannot exceed this value.
<code><storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims</code>	Across all persistent volume claims associated with the storage-class-name, the total number of persistent volume claims that can exist in the namespace.



Kubernetes: Object Count Quotas

- `count/persistentvolumeclaims`
- `count/services`
- `count/secrets`
- `count/configmaps`
- `count/replicationcontrollers`
- `count/deployments.apps`
- `count/replicasets.apps`
- `count/statefulsets.apps`
- `count/jobs.batch`
- `count/cronjobs.batch`
- `count/deployments.extensions`



Kubernetes: Resource Quotas Example

cpu_memory_quota.yml

```
apiVersion: v1
kind: List
items:
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-medium
  spec:
    hard:
      cpu: "10"
      memory: 20Gi
      pods: "10"
    scopeSelector:
      matchExpressions:
        - operator : In
          scopeName: PriorityClass
          values: ["medium"]
```

compute_quota.yml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    pods: "4"
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```



Kubernetes: Limit Resource by Pods

pod.yml

```
apiVersion: v1
kind: Pod
spec:
  containers:
    - name: db
      ...
      resources:
        requests:
          memory: "64Mi"
          cpu: "250m"
        limits:
          memory: "128Mi"
          cpu: "500m"
    - name: wp
      ...
      resources:
        requests:
          memory: "64Mi"
          cpu: "250m"
        limits:
          memory: "128Mi"
          cpu: "500m"
```



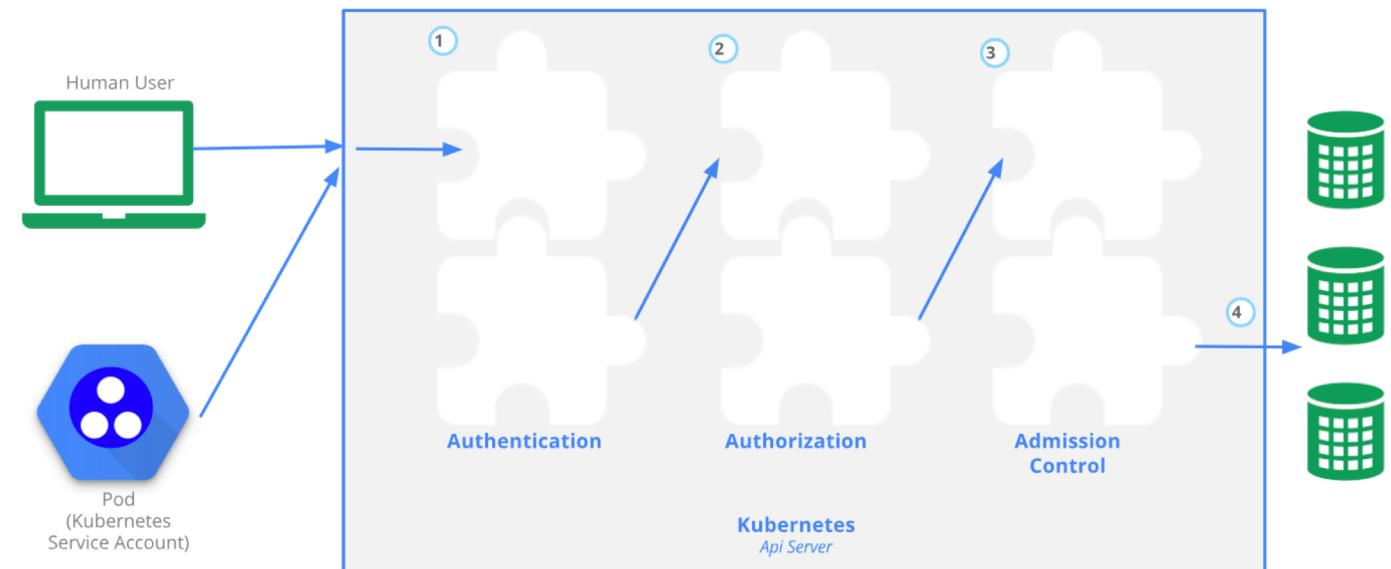
Kubernetes Authentication and Access Control

Control Access to APIs

Human users and K8s service accounts can be authorized for API access.

When a request reaches the API, it goes through 3 stages.

1. Authentication
2. Authorization
3. Admission Control



Step 1: Authentication

HTTP request is passed to the authenticator to authenticate the user.

Type of authenticator (set by cluster admin):

- client certificate (\$USER/.kube/config contains root certificate for api-server certificate)
- password
- plain token
- bootstrap token

* If the admin sets multiple, this step checks all and grants access even if one is passed.

Status:

- **pass**: the user is authenticated as "username", and this "username" is available to subsequent steps.
- **fail**: 401 code



Step 2: Authorization

After successful authentication, the HTTP request is authorized for existing "**policies**"
HTTP request contains {"**username**", "requested action", "objects getting affected"}

Type of authorizers (set by cluster admin):

- **RBAC** – Role based access control
- **ABAC** – Attribute based access control

RBAC and ABAC allows the cluster-admin to dynamically configure policies through the Kubernetes API to grant restricted access.

* If the admin sets multiple, this step checks all and grants access even if one is passed.



RBAC (Role based Access Control)

Role – are additive rules with set of permissions. (no "deny" rules)

API group: rbac.authorization.k8s.io

Enable api-server with: --authorization-mode=RBAC

Type of Roles:

1. **Role** – within a namespace
2. **ClusterRole** – within the cluster



Role

role for accessing pods in default namespace

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

The above role won't be able to access pod logs # {pod

= namespaced resource}, {logs = is not}

```
rules:
- apiGroups: [""]
  resources: ["pods", "pods/log"]
  verbs: ["get", "list"]
```

role using resourceNames

```
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  resourceNames: ["my-configmap"]
  verbs: ["update", "get"]
```

Check all endpoints
\$ kubectl proxy &
\$ curl localhost:8001

role for a non-resource endpoint

```
rules:
- nonResourceURLs: ["/healthz", "/healthz/*"]
  verbs: ["get", "post"]
```



RoleBinding

RoleBinding – refer subjects (users, groups, or service accounts) to a role

```
# A role binding allows "Mary" to read
# pods in the "default" namespace
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: Mary # case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

```
# binding for a group
subjects:
- kind: Group
  name: "frontend-admins"
```

```
# default service account in the kube-system namespace
subjects:
- kind: ServiceAccount
  name: default
  namespace: kube-system
```

```
# binding for all service accounts in the "qa" namespace
subjects:
- kind: Group
  name: system:serviceaccounts:qa
```



Question – why need namespace for a RoleBinding when it is already scoped and defined in a Role?

Answer – A RoleBinding may also refer a ClusterRole to grant the permissions to namespaced resources defined in the ClusterRole within the RoleBinding's namespace.

E.g. for a ClusterRole that reads secrets in all namespace, if you define a RoleBinding (with namespace=n1) referring to the ClusterRole, the user will read secrets in the namespace=n1 only not in all namespace (which was the scope of ClusterRole)



ClusterRole

ClusterRole = Roles defined for the entire cluster

ClusterRole = Role + cluster resources (like nodes) + non-resource (like /healtz) +
not limited to one namespace

```
# ClusterRole to read secrets across all namespace
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
    # no "namespace", ClusterRoles are not namespaced
    name: secret-reader
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```



ClusterRole Binding

ClusterRoleBinding – refer subjects (users, groups, or service accounts) to a ClusterRole

This cluster role binding allows anyone in the "manager" group to read secrets in any namespace.

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-secrets-global
subjects:
- kind: Group
  name: manager # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```



ABAC (Attribute based Access Control)

Defines an access control paradigm whereby access rights are granted to users through the use of policies which combine attributes together

API group: abac.authorization.k8s.io

Enable api-server with: --authorization-mode=ABAC

Provide api-server with access file: --authorization-policy-file=SOME_FILENAME



Each line in the policy file is the policy JSON object



ABAC Policies

```
# John has read-only access to pods in 'developer' namespace
{
  "apiVersion": "abac.authorization.kubernetes.io/v1beta1",
  "kind": "Policy",
  "spec": {
    "user": "john200",
    "namespace": "developer",
    "resource": "pods",
    "readonly": true
  }
}
```

ServiceAccount: is identified by
"system:serviceaccount:<namespace>:<serviceaccountname>"

```
# default service account (in the kube-system namespace) full
privilege to the API
{
  "apiVersion": "abac.authorization.kubernetes.io/v1beta1",
  "kind": "Policy",
  "spec": {
    "user": "system:serviceaccount:kube-system:default",
    "namespace": "*",
    "resource": "*",
    "apiGroup": "*"
  }
}
```



Step 3: Admission Control

Checks if the user requesting for a write operation has the necessary permission to bring the change in the K8s resource. Thus, happens prior to persistence of the object.

Enable: `kube-apiserver --enable-admission-plugins=NamespaceLifecycle,LimitRanger`

Plugins enabled by default: NamespaceLifecycle, LimitRanger, ServiceAccount, PersistentVolumeClaimResize, DefaultStorageClass, DefaultTolerationSeconds, MutatingAdmissionWebhook, ValidatingAdmissionWebhook, ResourceQuota, Priority

* Many advanced features in Kubernetes require an admission controller to be enabled in order to properly support the feature.



Service Accounts

-- Provides an **identity for processes that runs in a Pod**

Human users are authenticated by the api-server as a **particular User account**.

Processes in containers inside pods are authenticated by the api-server as a **particular Service account**.

When a pod is created;

spec.serviceAccountName mentions the ServiceAccount, "default" is auto assigned.

The API permissions of the service account is defined in the **authorization policy** (RBAC or ABAC) as discussed earlier.

Good practice: Grant a role to an application-specific service account



Service Accounts – Tokens

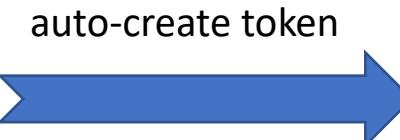
create a service account named acc1

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: acc1
```



create a pod using acc1 service account

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  serviceAccountName: acc1
```



secrets:

```
- name: acc1-token-bvbk5
```



manually created token and use for a service account

```
apiVersion: v1
kind: Secret
metadata:
  name: acc1-secret
  annotations:
    kubernetes.io/service-account.name: acc1
type: kubernetes.io/service-account-token
```



Service Accounts – ImagePullSecret

Case: Provide specific access to serviceaccounts (in turn pods) to pull images from multiple docker registries

- Create a ImagePullSecret

```
$ kubectl create secret docker-registry myregistrykey --docker-server=DOCKER_REGISTRY_SERVER --docker-username=DOCKER_USER --docker-password=DOCKER_PASSWORD --docker-email=DOCKER_EMAIL
```

- Change the service account to use the above ImagePullSecret

```
$ kubectl patch serviceaccount default -p '{"imagePullSecrets": [{"name": "myregistrykey"}]}
```

* Now the pod using the serviceaccount “default” will use the secret to pull images from the specific docker registry



Service Accounts – Pod creation

Case: What happens with ServiceAccount when a Pod is created

- If the pod does not have a ServiceAccount set, it sets the ServiceAccount to **default**.
- It ensures that the ServiceAccount referenced by the pod exists, and otherwise rejects it.
- If the pod does not contain any ImagePullSecrets, then ImagePullSecrets of the ServiceAccount are added to the pod.
- It adds a volume to the pod which contains a token for API access.
- It adds a volumeSource to each container of the pod mounted at `/var/run/secrets/kubernetes.io/serviceaccount`
 - We have seen this in one of the exercises where two containers in the same pod trying to share volumes – was authenticated by serviceaccount secret that was auto-created



Service Accounts – apiserver access from pod

Case: Accessing api-server from a pod (use ServiceAccount token)

- Place the serviceaccount token in the containers of the pod in the path -
/var/run/secrets/kubernetes.io/serviceaccount/token
- Run "kubectl proxy"

kubectl proxy

- Creates a proxy server or application-level gateway between localhost and the Kubernetes API Server.

