# Kubernetes workloads

## Deployments

Deployments represent a set of multiple, identical Pods with no unique identities. A Deployment runs multiple replicas of your application and automatically replaces any instances that fail or become unresponsive. In this way, Deployments help ensure that one or more instances of your application are available to serve user requests. Deployments are managed by the Kubernetes Deployment controller.

Deployments use a Pod template, which contains a specification for its Pods. The Pod specification determines how each Pod should look like: what applications should run inside its containers, which volumes the Pods should mount, its labels, and more.

When a Deployment's Pod template is changed, new Pods are automatically created one at a time.

You can create your own deployment using using the kubectl CLI tool:

$ kubectl create deployment example --image=nginx:1.16.1  
deployment "example" created

You can see the deployment you just created with:

$ kubectl get deployments  
NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE  
example 1 1 1 1 1m

And you can see the replicasets created by the deployments using:

$ kubectl get replicasets  
NAME DESIRED CURRENT READY AGE  
example-1745274054 1 1 1 3m

To see the details of the deployment use:

$ kubectl describe deployment example  
Name: example  
Namespace: platform  
CreationTimestamp: Tue, 13 Jun 2017 23:58:46 +0200  
Labels: run=example  
Selector: run=example  
Replicas: 1 updated | 1 total | 1 available | 0 unavailable  
StrategyType: RollingUpdate  
MinReadySeconds: 0  
RollingUpdateStrategy: 1 max unavailable, 1 max surge  
Conditions:  
 Type Status Reason  
 ---- ------ ------  
 Available True MinimumReplicasAvailable  
OldReplicaSets: <none>  
NewReplicaSet: example-1745274054 (1/1 replicas created)  
Events:  
 FirstSeen LastSeen Count From SubObjectPath Type Reason Message  
 --------- -------- ----- ---- ------------- -------- ------ -------  
 4m 4m 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set example-1745274054 to 1

We can scale the deployment to a larger number of pods using:

$ kubectl scale deployment example --replicas=2  
deployment "example" scaled

As you can see by viewing the pods deployed:

kubectl get pods  
NAME READY STATUS RESTARTS AGE  
example-1745274054-5d8r7 1/1 Running 0 51s  
example-1745274054-qm9h0 1/1 Running 0 7m

Delete a pod and see that it will be recreated by kubernetes:

$ kubectl delete pod example-1745274054-qm9h0  
pod "example-1745274054-qm9h0" deleted  
  
$ kubectl get pods  
NAME READY STATUS RESTARTS AGE  
example-1745274054-5d8r7 1/1 Running 0 1m  
example-1745274054-qm9h0 1/1 Terminating 0 43s  
example-4113047535-wr6sw 1/1 Running 0 4s

Now we want to upgrade our deployment to a new version. First we check the strategy for this deployment. This can be either: RollingUpdate or Recreate. Default this is set to RollingUpdate. The controller will than always keep a minimum amount of pods available. With Recreate all pods are first destroyed before new ones are created.

$ kubectl get deployment example -o yaml  
...  
 strategy:  
 rollingUpdate:  
 maxSurge: 1  
 maxUnavailable: 1  
 type: RollingUpdate  
...

We want to update our webserver container to the new Nginx version 1.17.8. To upgrade our example deployment we can either edit our deployment using:

method 1: edit the deployment and change the version, this will open the running Yaml in a VI editor.

$ kubectl edit deployment example

method 2: specify the new image using the command line

$ kubectl set image deployment example nginx=nginx:1.17.8  
deployment "example" image updated

After this you can view the status of the rollout using:

$ kubectl rollout status deployment example  
deployment "example" successfully rolled out

You can now also see two replicasets (old and new) using:

$ kubectl get replicasets  
NAME DESIRED CURRENT READY AGE  
example-1745274054 0 0 0 35m  
example-824231406 2 2 2 2m

You can undo a deployment using:

$ kubectl rollout undo deployment example  
deployment "example" rolled back

For more information on deployments see: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

## 8 persistent storage

Not all your applicaties will probably be stateless. To enable applications which need a form of state stored on disk you can use persistent volume claims. Kubernetes doen't come out of the box with a storage solution. There are several ways to enalbe the persistent storage and hand out persistent volumes to the applications.

We will create our own persistant volume and persistent volume claim and link a deployment to it. First make a directory on the host:

mkdir /workshop-volume

Create a file called: pv.yml with the following content:

apiVersion: v1  
kind: PersistentVolume  
metadata:  
 name: task-pv-volume  
 labels:  
 type: local  
spec:  
 storageClassName: manual  
 capacity:  
 storage: 5Gi  
 accessModes:  
 - ReadWriteOnce  
 hostPath:  
 path: "/workshop-volume"

Now create the pv using:

$ kubectl apply -f pv.yml  
persistentvolume "task-pv-volume" created

On this PV we will create the PersistentVolumeClaim, this is also done with a yaml file. Create a file called: pvc.yml with the following content:

apiVersion: v1  
kind: PersistentVolumeClaim  
metadata:  
 name: my-pvc  
spec:  
 storageClassName: manual  
 accessModes:  
 - ReadWriteOnce  
 resources:  
 requests:  
 storage: 4Gi

Now create the pvc using:

$ kubectl apply -f pvc.yml  
persistentvolumeclaim "my-pvc" created

You can get the status and see if the persistent volume has been bound to a persistent volume.

$ kubectl get pvc my-pvc  
NAME STATUS VOLUME CAPACITY ACCESSMODES STORAGECLASS AGE  
my-pvc Bound pvc-c1acaf5c-93ce-11e7-a20d-000d3a2a7d16 1Gi RWO glusterfs-storage 1m  
  
$ kubectl describe pvc my-pvc  
Name: my-pvc  
Namespace: namespace11  
StorageClass: glusterfs-storage  
Status: Bound  
Volume: pvc-c1acaf5c-93ce-11e7-a20d-000d3a2a7d16  
Labels: <none>  
Annotations: pv.kubernetes.io/bind-completed=yes  
 pv.kubernetes.io/bound-by-controller=yes  
 volume.beta.kubernetes.io/storage-class=glusterfs-storage  
 volume.beta.kubernetes.io/storage-provisioner=kubernetes.io/glusterfs  
Capacity: 1Gi  
Access Modes: RWO  
Events: <none>

We can bound this volume to a set of containers, e.g. to a deployment:

Create a file called deployment-with-pvc.yml This is a deployment for a MySQL database with persistant storage

---  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: mysql  
 labels:  
 app: mysql  
spec:  
 selector:  
 matchLabels:  
 app: mysql  
 tier: mysql  
 strategy:  
 type: Recreate  
 template:  
 metadata:  
 labels:  
 app: mysql  
 tier: mysql  
 spec:  
 containers:  
 - image: mysql:5.6  
 name: mysql  
 env:  
 - name: MYSQL\_ROOT\_PASSWORD  
 value: "password"  
 ports:  
 - containerPort: 3306  
 name: mysql  
 volumeMounts:  
 - name: mysql-persistent-storage  
 mountPath: /var/lib/mysql  
 volumes:  
 - name: mysql-persistent-storage  
 persistentVolumeClaim:  
 claimName: my-pvc

and create the deployment with:

$ kubectl apply -f deployment-with-pvc.yml

When you go into one of the pods you should be able to see the attached volume. Find the pod id with kubectl get pods and go into the pod with kubectl exec:

$ kubectl get pods  
  
  
$ kubectl exec -it [pod-name] bash  
root@example-3852110269-39rk9:/usr/src/app# df