K8s Installation:

Refer to the Installation guides depending on the infrastructure.

K8s Basics:

**Let’s start!**

kubectl version

kubectl get nodes

#Get a list of API Resources available in the cluster

kubectl api-resources | more

#Using kubectl explain

kubectl explain pods | more

**Let’s create a sample pod: manifests/mypod.yaml**

apiVersion: v1

kind: Pod

metadata:

name: hello-world

spec:

containers:

- name: hello-world

image: gcr.io/google-samples/hello-app:1.0

#Creating a pod with YAML

kubectl apply -f manifests/mypod.yaml

#Let's look more closely at what we need in pod.spec and pod.spec.containers

kubectl explain pod.spec | more

kubectl explain pod.spec.containers | more

#Get a list of our currently running pods

kubectl get pod

#Anatomy of an API Requests

#Get a specific Pod

kubectl get pod hello-world

#We can use the -v option to increase the verbosity of our request.

#Display requested resource URL. Focus on VERB, API Path and Response code

kubectl get pod hello-world -v 6

#Same output as 6, add HTTP Request Headers. Focus on application type, and User-Agent

kubectl get pod hello-world -v 7

#Same output as 7, adds Response Headers and truncated Response Body.

kubectl get pod hello-world -v 8

#Same output as 8, add full Response. Focus on the bottom, look for metadata

kubectl get pod hello-world -v 9

**Let’s access cluster resources via Kube API Service:**

#Start up a kubectl proxy session, this will authenticate use to the API Server

#Using our local kubeconfig for authentication and settings

kubectl proxy &

curl http://localhost:8001/api/v1/namespaces/default/pods/hello-world | head -n 20

fg

ctrl+c

#Watch, Exec and Log Requests

#A watch on Pods will watch on the resourceVersion on api/v1/namespaces/default/Pods

kubectl get pods --watch -v 6 &

#We can see kubectl keeps the TCP session open with the server...waiting for data.

netstat -plant | grep kubectl

#Delete the pod and we see the updates are written to our stdout. Watch stays, sicne we're watching All Pods in the default namespace.

kubectl delete pods hello-world

#But let's bring our Pod back...as we have more demos. :)

kubectl apply -f pod.yaml

#And kill off our watch

fg

ctrl+c

#Accessing logs

kubectl logs hello-world

kubectl logs hello-world -v 6

#Start kubectl proxy, we can access the resource URL directly.

kubectl proxy &

curl http://localhost:8001/api/v1/namespaces/default/pods/hello-world/log

#Kill our kubectl proxy, fg then ctrl+c

fg

ctrl+c

#Authentication failure Demo

cp ~/.kube/config ~/.kube/config.ORIG

#Make an edit to our username changing user: kubernetes-admin to user: kubernetes-admin1

vi ~/.kube/config

#Try to access our cluster, and we see GET on ./api/v1/namespaces/default/pods?limit=500 returns a 403. Status, failure.

kubectl get pods -v 6

#Let's put our backup kubeconfig back

cp ~/.kube/config.ORIG ~/.kube/config

#Test out access to the API Server

kubectl get pods

**Commands are going to be executed on the Master node [Kubectl is configured to access cluster]:**

kubectl uses a common syntax for all operations in the form of:

kubectl <command> <type> <name> <flags>

* command - The command or operation to perform. e.g. apply, create, delete, and get.
* type - The resource type or object.
* name - The name of the resource or object.
* flags - Optional flags to pass to the command.

**Example:**

$ kubectl create -f mypod.yaml

$ kubectl get pods

$ kubectl get pod mypod

$ kubectl delete pod mypod

# Context and kubeconfig

kubectl allows a user to interact with and manage multiple Kubernetes clusters. To do this, it requires what is known as a context. A context consists of a combination of cluster, namespace and user.

* cluster - A friendly name, server address, and certificate for the Kubernetes cluster.
* namespace (optional) - The logical cluster or environment to use. If none is provided, it will use the default default namespace.
* user - The credentials used to connect to the cluster. This can be a combination of client certificate and key, username/password, or token.

These contexts are stored in a local yaml based config file referred to as the kubeconfig. For \*nix based systems, the kubeconfig is stored in $HOME/.kube/config for Windows, it can be found in %USERPROFILE%/.kube/config

This config is viewable without having to view the file directly.

Command

$ kubectl config view

### kubectl config

Managing all aspects of contexts is done via the kubectl config command. Some examples include:

* See the active context with kubectl config current-context.
* Get a list of available contexts with kubectl config get-contexts.
* Switch to using another context with the kubectl config use-context <context-name> command.
* Add a new context with kubectl config set-context <context name> --cluster=<cluster name> --user=<user> --namespace=<namespace>.

There can be quite a few specifics involved when adding a context, for the available options, please see the [Configuring Multiple Clusters](https://kubernetes.io/docs/tasks/access-application-cluster/configure-access-multiple-clusters/) Kubernetes documentation.

Examples:

View the current contexts.

$ kubectl config get-contexts

View the current active context.

$ kubectl config current-context

# Kubectl Basics

There are several kubectl commands that are frequently used for any sort of day-to-day operations. get, create, apply, delete, describe, and logs. Other commands can be listed simply with kubectl --help, or kubectl <command> --help.

### kubectl get

kubectl get fetches and lists objects of a certain type or a specific object itself. It also supports outputting the information in several different useful formats including: json, yaml, wide (additional columns), or name (names only) via the -o or --output flag.

Command

kubectl get <type>

kubectl get <type> <name>

kubectl get <type> <name> -o <output format>

Examples

$ kubectl get namespaces

NAME STATUS AGE

default Active 4h

kube-public Active 4h

kube-system Active 4h

$

$kubectl get pod mypod -o wide

NAME READY STATUS RESTARTS AGE IP NODE

mypod 1/1 Running 0 5m 172.17.0.6 minikube

### kubectl create

kubectl create creates an object from the command line (stdin) or a supplied json/yaml manifest. The manifests can be specified with the -f or --filename flag that can point to either a file, or a directory containing multiple manifests.

Command

kubectl create <type> <parameters>

kubectl create -f <path to manifest>

Examples

$ kubectl create namespace dev

namespace "dev" created

$

$ kubectl create -f manifests/mypod.yaml

pod "mypod" created

### kubectl apply

kubectl apply is similar to kubectl create. It will essentially update the resource if it is already created, or simply create it if does not yet exist. When it updates the config, it will save the previous version of it in an annotation on the created object itself.

WARNING: If the object was not created initially with kubectl apply it's updating behavior will act as a two-way diff. For more information on this, please see the [kubectl apply](https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/#kubectl-apply) documentation.

Just like kubectl create it takes a json or yaml manifest with the -f flag or accepts input from stdin.

Command

kubectl apply -f <path to manifest>

Examples

$ kubectl apply -f manifests/mypod.yaml

Warning: kubectl apply should be used on resource created by either kubectl create --save-config or kubectl apply

pod "mypod" configured

### kubectl edit

kubectl edit modifies a resource in place without having to apply an updated manifest. It fetches a copy of the desired object and opens it locally with the configured text editor, set by the KUBE\_EDITOR or EDITOR Environment Variables. This command is useful for troubleshooting, but should be avoided in production scenarios as the changes will essentially be untracked.

Command

$ kubectl edit <type> <object name>

Examples

kubectl edit pod mypod

kubectl edit service myservice

### kubectl delete

kubectl delete deletes the object from Kubernetes.

Command

kubectl delete <type> <name>

Examples

$ kubectl delete pod mypod

pod "mypod" deleted

### kubectl describe

kubectl describe lists detailed information about the specific Kubernetes object. It is a very helpful troubleshooting tool.

Command

kubectl describe <type>

kubectl describe <type> <name>

kubectl logs

kubectl logs outputs the combined stdout and stderr logs from a pod. If more than one container exist in a pod the -c flag is used and the container name must be specified.

Command

kubectl logs <pod name>

kubectl logs <pod name> -c <container name>

Examples

$ kubectl logs mypod

172.17.0.1 - - [10/Mar/2018:18:14:15 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.57.0" "-"

172.17.0.1 - - [10/Mar/2018:18:14:17 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.57.0" "-"

### Exercise: The Basics

Objective: Explore the basics. Create a namespace, a pod, then use the kubectl commands to describe and delete what was created.

1. Create the dev namespace.

kubectl create namespace dev

1. Apply the manifest manifests/mypod.yaml.

kubectl apply -f manifests/mypod.yaml

1. Get the yaml output of the created pod mypod.

kubectl get pod mypod -o yaml

1. Describe the pod mypod.

kubectl describe pod mypod

1. Clean up the pod by deleting it.

kubectl delete pod mypod

Summary: The kubectl *"CRUD"* commands are used frequently when interacting with a Kubernetes cluster. These simple tasks become 2nd nature as more experience is gained.

# Accessing the Cluster

kubectl provides several mechanisms for accessing resources within the cluster remotely. For this section, the focus will be on using kubectl exec to get a remote shell within a container, and kubectl proxy to gain access to the services exposed through the API proxy.

### kubectl exec

kubectl exec executes a command within a Pod and can optionally spawn an interactive terminal within a remote container. When more than one container is present within a Pod, the -c or --container flag is required, followed by the container name.

If an interactive session is desired, the -i (--stdin) and -t (--tty) flags must be supplied.

Command

kubectl exec <pod name> -- <arg>

kubectl exec <pod name> -c <container name> -- <arg>

kubectl exec -i -t <pod name> -c <container name> -- <arg>

kubectl exec -it <pod name> -c <container name> -- <arg>

Example

$ kubectl exec mypod -c nginx -- printenv

PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

HOSTNAME=mypod

KUBERNETES\_SERVICE\_PORT\_HTTPS=443

KUBERNETES\_PORT=tcp://10.96.0.1:443

KUBERNETES\_PORT\_443\_TCP=tcp://10.96.0.1:443

KUBERNETES\_PORT\_443\_TCP\_PROTO=tcp

KUBERNETES\_PORT\_443\_TCP\_PORT=443

KUBERNETES\_PORT\_443\_TCP\_ADDR=10.96.0.1

KUBERNETES\_SERVICE\_HOST=10.96.0.1

KUBERNETES\_SERVICE\_PORT=443

NGINX\_VERSION=1.12.2

HOME=/root

$

$ kubectl exec -i -t mypod -c nginx -- /bin/sh

/ #

/ # cat /etc/alpine-release

3.5.2

### kubectl proxy

kubectl proxy enables access to both the Kubernetes API-Server and to resources running within the cluster securely using kubectl. By default it creates a connection to the API-Server that can be accessed at 127.0.0.1:8001 or an alternative port by supplying the -p or --port flag.

Command

kubectl proxy

kubectl proxy --port=<port>

Examples

$ kubectl proxy &

Starting to serve on 127.0.0.1:8001

<from another terminal>

$ curl 127.0.0.1:8001/version

{

"major": "",

"minor": "",

"gitVersion": "v1.9.0",

"gitCommit": "925c127ec6b946659ad0fd596fa959be43f0cc05",

"gitTreeState": "clean",

"buildDate": "2018-01-26T19:04:38Z",

"goVersion": "go1.9.1",

"compiler": "gc",

"platform": "linux/amd64"

}

The Kubernetes API-Server has the built-in capability to proxy to running services or pods within the cluster. This ability in conjunction with the kubectl proxy command allows a user to access those services or pods without having to expose them outside of the cluster.

http://<proxy\_address>/api/v1/namespaces/<namespace>/<services|pod>/<service\_name|pod\_name>[:port\_name]/proxy

* proxy\_address - The local proxy address - 127.0.0.1:8001
* namespace - The namespace owning the resources to proxy to.
* service|pod - The type of resource you are trying to access, either service or pod.
* service\_name|pod\_name - The name of the service or pod to be accessed.
* [:port] - An optional port to proxy to. Will default to the first one exposed.

Example

http://127.0.0.1:8001/api/v1/namespaces/default/pods/mypod/proxy/

http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/kubernetes-dashboard/proxy/

**Kubectl Cheat Sheet:** [**https://kubernetes.io/docs/reference/kubectl/cheatsheet/**](https://kubernetes.io/docs/reference/kubectl/cheatsheet/)

**Kubectl reference:**

[**https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands**](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands)

# Namespaces

Namespaces are a logical cluster or environment. They are the primary method of partitioning a cluster or scoping access.

### Exercise: Using Namespaces

Objectives: Learn how to create and switch between Kubernetes Namespaces using kubectl.

1. List the current namespaces

$ kubectl get namespaces

1. Create the dev namespace

$ kubectl create namespace dev

manifests/ns-example.yaml

apiVersion: v1

kind: Pod

metadata:

name: ns-example

namespace: dev

labels:

app: nginx

environment: prod

spec:

containers:

- name: nginx

image: nginx:stable-alpine

ports:

- containerPort: 80

Command

$ kubectl create -f manifests/ns-example.yaml

$ kubectl get pods --namespace=dev

$ kubectl get pods --namespace=  
$ kubectl config set-context --current --namespace=dev

<https://kubernetes.io/docs/tasks/administer-cluster/namespaces/#creating-a-new-namespace>

Summary: Namespaces function as the primary method of providing scoped names, access, and act as an umbrella for group based resource restriction. Creating and switching between them is quick and easy, but learning to use them is essential in the general usage of Kubernetes.

# Pods

A pod is the atomic unit of Kubernetes. It is the smallest *“unit of work”* or *“management resource”* within the system and is the foundational building block of all Kubernetes Workloads.

### Exercise: Creating Pods

Objective: Examine both single and multi-container Pods; including: viewing their attributes through the cli and their exposed Services through the API Server proxy.

1. Create a simple Pod called pod-example using the nginx:stable-alpine image and expose port 80. Use the manifest manifests/pod-example.yaml or the yaml below.

manifests/pod-example.yaml

apiVersion: v1

kind: Pod

metadata:

name: pod-example

labels:

app: nginx

environment: prod

spec:

containers:

- name: nginx

image: nginx:stable-alpine

ports:

- containerPort: 80

Command

$ kubectl create -f manifests/pod-example.yaml

1. Use kubectl to describe the Pod and note the available information.

$ kubectl describe pod pod-example

1. Use kubectl proxy to verify the web server running in the deployed Pod.

Command

$ kubectl proxy &

URL

[http://127.0.0.1:8001/api/v1/namespaces/default/pods/pod-example/proxy/](http://127.0.0.1:8001/api/v1/namespaces/dev/pods/pod-example/proxy/)

The default "Welcome to nginx!" page should be visible.

Delete all kubectl proces

killall kubectl

1. Using the same steps as above, create a new Pod called multi-container-example using the manifest manifests/pod-multi-container-example.yaml or create a new one yourself with the below yaml.

manifests/pod-multi-container-example.yaml

apiVersion: v1

kind: Pod

metadata:

name: multi-container-example

spec:

containers:

- name: nginx

image: nginx:stable-alpine

ports:

- containerPort: 80

volumeMounts:

- name: html

mountPath: /usr/share/nginx/html

- name: content

image: alpine:latest

volumeMounts:

- name: html

mountPath: /html

command: ["/bin/sh", "-c"]

args:

- while true; do

echo $(date)"<br />" >> /html/index.html;

sleep 5;

done

volumes:

- name: html

emptyDir: {}

Command

$ kubectl create -f manifests/pod-multi-container-example.yaml

Note: spec.containers is an array allowing you to use multiple containers within a Pod.

1. Use the proxy to verify the web server running in the deployed Pod.

Command

$ kubectl proxy

URL

http://127.0.0.1:8001/api/v1/namespaces/default/pods/multi-container-example/proxy/

There should be a repeating date-time-stamp.

Summary: Becoming familiar with creating and viewing the general aspects of a Pod is an important skill. While it is rare that one would manage Pods directly within Kubernetes, the knowledge of how to view, access and describe them is important and a common first-step in troubleshooting a possible Pod failure.

# Labels and Selectors

Labels are key-value pairs that are used to identify, describe and group together related sets of objects or resources.

Selectors use labels to filter or select objects, and are used throughout Kubernetes.

<https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/>

### Exercise: Using Labels and Selectors

Objective: Explore the methods of labeling objects in addition to filtering them with both equality and set-based selectors.

1. Label the Pod pod-example with app=nginx and environment=dev via kubectl.

$ kubectl label pod pod-example app=nginx environment=dev

1. View the labels with kubectl by passing the --show-labels flag

$ kubectl get pods --show-labels

1. Update the multi-container example manifest created previously with the labels app=nginx and environment=prod then apply it via kubectl.

manifests/pod-multi-container-example.yaml

apiVersion: v1

kind: Pod

metadata:

name: multi-container-example1

labels:

app: nginx

environment: dev

spec:

containers:

- name: nginx

image: nginx:stable-alpine

ports:

- containerPort: 80

volumeMounts:

- name: html

mountPath: /usr/share/nginx/html

- name: content

image: alpine:latest

volumeMounts:

- name: html

mountPath: /html

command: ["/bin/sh", "-c"]

args:

- while true; do

date >> /html/index.html;

sleep 5;

done

volumes:

- name: html

emptyDir: {}

Command

$ kubectl apply -f manifests/pod-multi-container-example.yaml

1. View the added labels with kubectl by passing the --show-labels flag once again.

$ kubectl get pods --show-labels

1. With the objects now labeled, use an [equality based selector](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#equality-based-requirement) targeting the prod environment.

$ kubectl get pods --selector environment=prod

1. Do the same targeting the nginx app with the short version of the selector flag (-l).

$ kubectl get pods -l app=nginx

1. Use a [set-based selector](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#set-based-requirement) to view all pods where the app label is nginx and filter out any that are in the prod environment.

$ kubectl get pods -l 'app in (nginx), environment notin (prod)'

Summary: Kubernetes makes heavy use of labels and selectors in near every aspect of it. The usage of selectors may seem limited from the cli, but the concept can be extended to when it is used with higher level resources and objects.

#Selector for multiple labels and adding on show-labels to see those labels in the output

kubectl get pods -l 'environment=prod,app=nginx --show-labels

kubectl get pods -l 'environment=prod,app!=nginx --show-labels

kubectl get pods -l 'environment in (prod,qa)'

kubectl get pods -l 'environment notin (prod,qa)'

#Output a particular label in column format

kubectl get pods -L tier

kubectl get pods -L tier,app

#Edit an existing label

kubectl label pod <<pod-name>> tier=non-prod --overwrite

kubectl get pod <<pod-name>> --show-labels

#Adding a new label

kubectl label pod <<pod-name>> another=Label

kubectl get pod <<pod-name>> --show-labels

#Removing an existing label

**kubectl label pod <<pod-name>> another-**

kubectl get pod <<pod-name>> --show-labels

#Performing an operation on a collection of pods based on a label query

kubectl label pod --all tier=non-prod --overwrite

kubectl get pod --show-labels

#Delete all pods matching our non-prod label

kubectl delete pod -l tier=non-prod

# Services

Services within Kubernetes are the unified method of accessing the exposed workloads of Pods. They are a durable resource (unlike Pods) that is given a static cluster-unique IP and provide simple load-balancing through kube-proxy.

### Exercise: The clusterIP Service

Objective: Create a ClusterIP service and view the different ways it is accessible within the cluster.

1. Create ClusterIP service clusterip that targets Pods labeled with app=nginx forwarding port 80 using either the yaml below, or the manifest manifests/service-clusterip.yaml.

manifests/service-clusterip.yaml

apiVersion: v1

kind: Service

metadata:

name: clusterip

spec:

selector:

app: nginx

ports:

- protocol: TCP

port: 80

targetPort: 80

Command

$ kubectl create -f manifests/service-clusterip.yaml

1. Describe the newly created service. Note the IP and the Endpoints fields.

$ kubectl describe service clusterip

1. View the service through kube proxy and refresh several times. It should serve up pages from both pods.

Command

$ kubectl proxy &

URL

http://127.0.0.1:8001/api/v1/namespaces/default/services/clusterip/proxy/

1. Lastly, verify that the generated DNS record has been created for the Service by using nslookup within the example-pod Pod that was provisioned in the above exercise.

$ kubectl exec pod-example -- nslookup clusterip.default.svc.cluster.local

**#kubectl exec hello-world -- nslookup hello-service.default.svc.cluster.local**

It should return a valid response with the IP matching what was noted earlier when describing the Service.

Summary: The ClusterIP Service is the most commonly used Service within Kubernetes. Every ClusterIP Service is given a cluster unique IP and DNS name that maps to one or more Pod Endpoints. It functions as the main method in which exposed Pod Services are consumed within a Kubernetes Cluster.

**In case you want to set the kubectl editor to nano use the following env variable**

**KUBE\_EDITOR="nano" kubectl edit pod pod-example**

### Exercise: Using NodePort

Objective: Create a NodePort based Service and explore how it is available both inside and outside the cluster.

1. Create a NodePort Service called nodeport that targets Pods with the labels app=nginx and environment=dev forwarding port 80 in cluster, and port 32410 on the node itself. Use either the yaml below, or the manifest manifests/service-nodeport.yaml.

manifests/service-nodeport.yaml

apiVersion: v1

kind: Service

metadata:

name: nodeport

spec:

type: NodePort

selector:

app: nginx

environment: prod

ports:

- nodePort: 32410

protocol: TCP

port: 80

targetPort: 80

Command

$ kubectl create -f manifests/service-nodeport.yaml

1. Describe the newly created Service Endpoint. Note the Service still has an internal cluster IP, and now additionally has a NodePort.

$ kubectl describe service nodeport

1. Open the newly exposed nodeport Service in a browser,

http://<<node\_ip>>:32410 OR

http://<<master\_ip>>:32410

1. Lastly, verify that the generated DNS record has been created for the Service by using nslookup within the example-pod Pod.

$ kubectl exec pod-example -- nslookup nodeport.default.svc.cluster.local

It should return a valid response with the IP matching what was noted earlier when describing the Service.

Summary: The NodePort Services extend the ClusterIP Service and additionally expose a port that is either statically defined, as above (port 32410) or dynamically taken from a range between 30000-32767. This port is then exposed on every node within the cluster and proxies to the created Service.

# ReplicaSets

ReplicaSets are the primary method of managing Pod replicas and their lifecycle. This includes their scheduling, scaling, and deletion.

Their job is simple, always ensure the desired number of replicas that match the selector are running.

### Exercise: Understanding ReplicaSets

Objective: Create and scale a ReplicaSet. Explore and gain an understanding of how the Pods are generated from the Pod template, and how they are targeted with selectors.

1. Begin by creating a ReplicaSet called rs-example with 3 replicas, using the nginx:stable-alpine image and configure the labels and selectors to target app=nginx and env=prod. The yaml block below or the manifest manifests/rs-example.yaml may be used.

manifests/rs-example.yaml

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: rs-example

spec:

replicas: 3

selector:

matchLabels:

app: nginx

env: prod

template:

metadata:

labels:

app: nginx

env: prod

spec:

containers:

- name: nginx

image: nginx:stable-alpine

ports:

- containerPort: 80

Command

$ kubectl create -f manifests/rs-example.yaml

1. Watch as the newly created ReplicaSet provisions the Pods based off the Pod Template.

$ kubectl get pods --watch --show-labels

Note that the newly provisioned Pods are given a name based off the ReplicaSet name appended with a 5 character random string. These Pods are labeled with the labels as specified in the manifest.

1. Scale ReplicaSet rs-example up to 5 replicas with the below command.

$ kubectl scale replicaset rs-example --replicas=5

Tip: replicaset can be substituted with rs when using kubectl.

1. Describe rs-example and take note of the Replicas and Pod Status field in addition to the Events.

$ kubectl describe rs rs-example

1. Now, using the scale command bring the replicas back down to 3.

$ kubectl scale rs rs-example --replicas=3

1. Watch as the ReplicaSet Controller terminates 2 of the Pods to bring the cluster back into it's desired state of 3 replicas.

$ kubectl get pods --show-labels --watch

1. Once rs-example is back down to 3 Pods. Create an independent Pod manually with the same labels as the one targeted by rs-example from the manifest manifests/pod-rs-example.yaml.

manifests/pod-rs-example.yaml

apiVersion: v1

kind: Pod

metadata:

name: pod-example

labels:

app: nginx

env: prod

spec:

containers:

- name: nginx

image: nginx:stable-alpine

ports:

- containerPort: 80

Command

$ kubectl create -f manifests/pod-rs-example.yaml

1. Immediately watch the Pods.

$ kubectl get pods --show-labels --watch

Note that the Pod is created and immediately terminated.

1. Describe rs-example and look at the events.

$ kubectl describe rs rs-example

There will be an entry with Deleted pod: pod-example. This is because a ReplicaSet targets ALL Pods matching the labels supplied in the selector.

Summary: ReplicaSets ensure a desired number of replicas matching the selector are present. They manage the lifecycle of ALL matching Pods. If the desired number of replicas matching the selector currently exist when the ReplicaSet is created, no new Pods will be created. If they are missing, then the ReplicaSet Controller will create new Pods based off the Pod Template till the desired number of Replicas are present.

Clean Up Command

kubectl delete rs rs-example

# Deployments

Deployments are a declarative method of managing Pods via ReplicaSets. They provide rollback functionality in addition to more granular update control mechanisms.

### Exercise: Using Deployments

Objective: Create, update and scale a Deployment as well as explore the relationship of Deployment, ReplicaSet and Pod.

1. Create a Deployment deploy-example. Configure it using the example yaml block below or use the manifest manifests/deploy-example.yaml. Additionally pass the --record flag to kubectl when you create the Deployment. The --record flag saves the command as an annotation, and it can be thought of similar to a git commit message.

manifests/deployment-example.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: deploy-example

spec:

replicas: 3

revisionHistoryLimit: 3

selector:

matchLabels:

app: nginx

strategy:

type: RollingUpdate

rollingUpdate:

maxSurge: 1

maxUnavailable: 0

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:stable-alpine

ports:

- containerPort: 80

Command

$ kubectl create -f manifests/deploy-example.yaml --record

1. Check the status of the Deployment.

$ kubectl get deployments

1. Once the Deployment is ready, view the current ReplicaSets and be sure to show the labels.

$ kubectl get rs --show-labels

Note the name and pod-template-hash label of the newly created ReplicaSet. The created ReplicaSet's name will include the pod-template-hash.

1. Describe the generated ReplicaSet.

$ kubectl describe rs deploy-example-<pod-template-hash>

Look at both the Labels and the Selectors fields. The pod-template-hash value has automatically been added to both the Labels and Selector of the ReplicaSet. Then take note of the Controlled By field. This will reference the direct parent object, and in this case the original deploy-example Deployment.

1. Now, get the Pods and pass the --show-labels flag.

$ kubectl get pods --show-labels

Just as with the ReplicaSet, the Pods name are labels include the pod-template-hash.

1. Describe one of the Pods.

$ kubectl describe pod deploy-example-<pod-template-hash-<random>

Look at the Controlled By field. It will contain a reference to the parent ReplicaSet, but not the parent Deployment.

Now that the relationship from Deployment to ReplicaSet to Pod is understood. It is time to update the deploy-example and see an update in action.

1. Update the deploy-example manifest and add a few additional labels to the Pod template. Once done, apply the change with the --record flag.

$ kubectl apply -f manifests/deploy-example.yaml --record

< or >

$ kubectl edit deploy deploy-example --record

Tip: deploy can be substituted for deployment when using kubectl.

1. Immediately watch the Pods.

$ kubectl get pods --show-labels --watch

The old version of the Pods will be phased out one at a time and instances of the new version will take its place. The way in which this is controlled is through the strategy stanza. For specific documentation this feature, see the [Deployment Strategy Documentation](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#strategy).

1. Now view the ReplicaSets.

$ kubectl get rs --show-labels

There will now be two ReplicaSets, with the previous version of the Deployment being scaled down to 0.

1. Now, scale the Deployment up as you would a ReplicaSet, and set the replicas=5.

$ kubectl scale deploy deploy-example --replicas=5

1. List the ReplicaSets.

$ kubectl get rs --show-labels

Note that there is NO new ReplicaSet generated. Scaling actions do NOT trigger a change in the Pod Template.

1. Just as before, describe the Deployment, ReplicaSet and one of the Pods. Note the Events and Controlled By fields. It should present a clear picture of relationship between objects during an update of a Deployment.

$ kubectl describe deploy deploy-example

$ kubectl describe rs deploy-example-<pod-template-hash>

$ kubectl describe pod deploy-example-<pod-template-hash-<random>

Summary: Deployments are the main method of managing applications deployed within Kubernetes. They create and supervise targeted ReplicaSets by generating a unique hash called the pod-template-hash and attaching it to child objects as a Label along with automatically including it in their Selector. This method of managing rollouts along with being able to define the methods and tolerances in the update strategy permits for a safe and seamless way of updating an application in place.

### Exercise: Rolling Back a Deployment

Objective: Learn how to view the history of a Deployment and rollback to older revisions.

1. Use the rollout command to view the history of the Deployment deploy-example.

$ kubectl rollout history deployment deploy-example

There should be two revisions. One for when the Deployment was first created, and another when the additional Labels were added. The number of revisions saved is based off of the revisionHistoryLimit attribute in the Deployment spec.

1. Look at the details of a specific revision by passing the --revision=<revision number> flag.

$ kubectl rollout history deployment deploy-example --revision=1

$ kubectl rollout history deployment deploy-example --revision=2

Viewing the specific revision will display a summary of the Pod Template.

1. Choose to go back to revision 1 by using the rollout undo command.

$ kubectl rollout undo deployment deploy-example --to-revision=1

Tip: The --to-revision flag can be omitted if you wish to just go back to the previous configuration.

1. Immediately watch the Pods.

$ kubectl get pods --show-labels --watch

They will cycle through rolling back to the previous revision.

1. Describe the Deployment deploy-example.

$ kubectl describe deployment deploy-example

The events will describe the scaling back of the previous and switching over to the desired revision.

Summary: Understanding how to use rollout command to both get a diff of the different revisions as well as be able to roll-back to a previously known good configuration is an important aspect of Deployments that cannot be left out.

Clean Up Command

kubectl delete deploy deploy-example

**Let’s explore readiness and liveness probe**

**Create a file hello-world-probes.yaml**

apiVersion: apps/v1

kind: Deployment

metadata:

name: hello-world

spec:

replicas: 1

selector:

matchLabels:

app: hello-world

template:

metadata:

labels:

app: hello-world

spec:

containers:

- name: hello-world

image: gcr.io/google-samples/hello-app:1.0

ports:

- containerPort: 8080

livenessProbe:

tcpSocket:

port: 8080

initialDelaySeconds: 10

periodSeconds: 5

readinessProbe:

httpGet:

path: /

port: 8080

initialDelaySeconds: 10

periodSeconds: 5

$ kubectl get deploy -o wide

**Let’s use readiness probe in a real scenario**

**Create a file named** deployment.probes-1.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: hello-world

spec:

replicas: 10

strategy:

type: RollingUpdate

rollingUpdate:

maxUnavailable: 10%

maxSurge: 2

revisionHistoryLimit: 20

selector:

matchLabels:

app: hello-world

template:

metadata:

labels:

app: hello-world

spec:

containers:

- name: hello-world

image: gcr.io/google-samples/hello-app:1.0

ports:

- containerPort: 8080

readinessProbe:

httpGet:

path: /index.html

port: 8080

initialDelaySeconds: 10

periodSeconds: 10

---

apiVersion: v1

kind: Service

metadata:

name: hello-world

spec:

selector:

app: hello-world

ports:

- port: 80

protocol: TCP

targetPort: 8080

#Let's deploy a Deployment with Readiness Probes

kubectl apply -f manifests/deployment.probes-1.yaml --record

#Available is still 0 because of our Readiness Probe's initialDelaySeconds is 10 seconds.

#Also, look there's a new annotation for our change-cause

#And check the Conditions,

# Progressing True NewReplicaSetCreated or ReplicaSetUpdated - depending on the state.

# Available False MinimumReplicasUnavailable

kubectl describe deployment hello-world

**Create a file named** deployment.probes-2.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: hello-world

spec:

replicas: 10

strategy:

type: RollingUpdate

rollingUpdate:

maxUnavailable: 10%

maxSurge: 2

revisionHistoryLimit: 20

selector:

matchLabels:

app: hello-world

template:

metadata:

labels:

app: hello-world

spec:

containers:

- name: hello-world

image: gcr.io/google-samples/hello-app:2.0

ports:

- containerPort: 8080

readinessProbe:

httpGet:

path: /index.html

port: 8080

initialDelaySeconds: 10

periodSeconds: 10

---

apiVersion: v1

kind: Service

metadata:

name: hello-world

spec:

selector:

app: hello-world

ports:

- port: 80

protocol: TCP

targetPort: 8080

#Let's update from v1 to v2 with Readiness Probes Controlling the rollout, and record our rollout

diff deployment.probes-1.yaml deployment.probes-2.yaml

kubectl apply -f deployment.probes-2.yaml --record

#Lots of pods, most are not ready yet, but progressing...how do we know it's progressing?

kubectl get replicaset

#Check again, Replicas and Conditions.

#Progressing is now ReplicaSetUpdated, will change to NewReplicaSetAvailable when it's Ready

#NewReplicaSet is THIS current RS, OldReplicaSet is populated during a Rollout, otherwise it's <None>

#We used the update strategy settings of max unavailable and max surge to slow this rollout down.

#This update takes about a minute to rollout

kubectl describe deployment hello-world

**Create a file named** deployment.probes-3.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: hello-world

spec:

replicas: 10

strategy:

type: RollingUpdate

rollingUpdate:

maxUnavailable: 10%

maxSurge: 2

revisionHistoryLimit: 2

selector:

matchLabels:

app: hello-world

template:

metadata:

labels:

app: hello-world

spec:

containers:

- name: hello-world

image: gcr.io/google-samples/hello-app:2.0

ports:

- containerPort: 8080

readinessProbe:

httpGet:

path: /index.html

port: 8081

initialDelaySeconds: 10

periodSeconds: 10

---

apiVersion: v1

kind: Service

metadata:

name: hello-world

spec:

selector:

app: hello-world

ports:

- port: 80

protocol: TCP

targetPort: 8080

#Let's update again, but I'm not going to tell you what I changed, we're going to troubleshoot it together

kubectl apply -f deployment.probes-3.yaml --record

#We stall at 4 out of 20 replicas updated...let's look

kubectl rollout status deployment hello-world

#Let's check the status of the Deployment, Replicas and Conditions,

#22 total (20 original + 2 max surge)

#18 available (20 original - 2 (10%) in the old RS)

#4 Unavailable, (only 2 pods in the old RS are offline, 4 in the new RS are not READY)

# Available True MinimumReplicasAvailable

# Progressing True ReplicaSetUpdated

kubectl describe deployment hello-world

#Let's look at our ReplicaSets, no Pods in the new RS are READY, but 4 our deployed.

#That RS with Desired 0 is from our V1 deployment, 18 is from our V2 deployment.

kubectl get replicaset

#Ready...that sounds familiar, let's check the deployment again

#What keeps a pod from reporting ready? A Readiness Probe...see that Readiness Probe, wrong port ;)

kubectl describe deployment hello-world

#We can read the Deployment's rollout history, and see our CHANGE-CAUSE annotations

kubectl rollout history deployment hello-world

#Let's rollback to revision 2 to undo that change...

kubectl rollout history deployment hello-world --revision=3

kubectl rollout history deployment hello-world --revision=2

kubectl rollout undo deployment hello-world --to-revision=2

#And check out our deployment to see if we get 20 Ready replicas

kubectl describe deployment | head

kubectl get deployment

# DaemonSets

DaemonSets ensure that all nodes matching certain criteria will run an instance of the supplied Pod.

They bypass default scheduling mechanisms and restrictions, and are ideal for cluster wide services such as log forwarding, or health monitoring.

### Exercise: Managing DaemonSets

Objective: Experience creating, updating, and rolling back a DaemonSet. Additionally delve into the process of how they are scheduled and how an update occurs.

1. Create DaemonSet ds-example and pass the --record flag. Use the example yaml block below as a base, or use the manifest manifests/ds-example.yaml directly.

manifests/ds-example.yaml

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: ds-example

spec:

revisionHistoryLimit: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

nodeSelector:

nodeType: edge

containers:

- name: nginx

image: nginx:stable-alpine

ports:

- containerPort: 80

Command

$ kubectl create -f manifests/ds-example.yaml --record

1. View the current DaemonSets.

$ kubectl get daemonset

As there are no matching nodes, no Pods should be scheduled.

1. Label the node01 node with nodeType=edge

$ kubectl label node <<your node>> nodeType=edge

1. View the current DaemonSets once again.

$ kubectl get daemonsets

There should now be a single instance of the DaemonSet ds-example deployed.

1. View the current Pods and display their labels with --show-labels.

$ kubectl get pods --show-labels

Note that the deployed Pod has a controller-revision-hash label. This is used like the pod-template-hash in a Deployment to track and allow for rollback functionality.

1. Describing the DaemonSet will provide you with status information regarding it's Deployment cluster wide.

$ kubectl describe ds ds-example

Tip: ds can be substituted for daemonset when using kubectl.

1. Update the DaemonSet by adding a few additional labels to the Pod Template and use the --record flag.

$ kubectl apply -f manifests/ds-example.yaml --record

< or >

$ kubectl edit ds ds-example --record

1. Watch the Pods and be sure to show the labels.

$ kubectl get pods --show-labels --watch

The old version of the DaemonSet will be phased out one at a time and instances of the new version will take its place. Similar to Deployments, DaemonSets have their own equivalent to a Deployment's strategy in the form of updateStrategy. The defaults are generally suitable, but other tuning options may be set. For reference, see the [Updating DaemonSet Documentation](https://kubernetes.io/docs/tasks/manage-daemon/update-daemon-set/#performing-a-rolling-update).

Summary: DaemonSets are usually used for important cluster-wide support services such as Pod Networking, Logging, or Monitoring. They differ from other workloads in that their scheduling bypasses normal mechanisms, and is centered around node placement. Like Deployments, they have their own pod-template-hash in the form of controller-revision-hash used for keeping track of Pod Template revisions and enabling rollback functionality.

### Optional: Working with DaemonSet Revisions

Objective: Explore using the rollout command to rollback to a specific version of a DaemonSet.

1. Use the rollout command to view the history of the DaemonSet ds-example

$ kubectl rollout history ds ds-example

There should be two revisions. One for when the Deployment was first created, and another when the additional Labels were added. The number of revisions saved is based off of the revisionHistoryLimit attribute in the DaemonSet spec.

1. Look at the details of a specific revision by passing the --revision=<revision number> flag.

$ kubectl rollout history ds ds-example --revision=1

$ kubectl rollout history ds ds-example --revision=2

Viewing the specific revision will display the Pod Template.

1. Choose to go back to revision 1 by using the rollout undo command.

$ kubectl rollout undo ds ds-example --to-revision=1

Tip: The --to-revision flag can be omitted if you wish to just go back to the previous configuration.

1. Immediately watch the Pods.

$ kubectl get pods --show-labels --watch

They will cycle through rolling back to the previous revision.

1. Describe the DaemonSet ds-example.

$ kubectl describe ds ds-example

The events will be sparse with a single host, however in an actual Deployment they will describe the status of updating the DaemonSet cluster wide, cycling through hosts one-by-one.

Summary: Being able to use the rollout command with DaemonSets is import in scenarios where one may have to quickly go back to a previously known-good version. This becomes even more important for 'infrastructure' like services such as Pod Networking.

Clean Up Command

kubectl delete ds ds-example

# 

# Jobs and CronJobs

The Job Controller ensures one or more Pods are executed and successfully terminate. Essentially a task executor that can be run in parallel.

CronJobs are an extension of the Job Controller, and enable Jobs to be run on a schedule.

### Exercise: Creating a Job

Objective: Create a Kubernetes Job and work to understand how the Pods are managed with completions and parallelism directives.

1. Create job job-example using the yaml below, or the manifest located at manifests/job-example.yaml

manifests/job-example.yaml

apiVersion: batch/v1

kind: Job

metadata:

name: job-example

spec:

backoffLimit: 4

completions: 4

parallelism: 2

template:

spec:

containers:

- name: hello

image: alpine:latest

command: ["/bin/sh", "-c"]

args: ["echo hello from $HOSTNAME!"]

restartPolicy: Never

Command

$ kubectl create -f manifests/job-example.yaml

1. Watch the Pods as they are being created.

$ kubectl get pods --show-labels --watch

Only two Pods are being provisioned at a time; adhering to the parallelism attribute. This is done until the total number of completions is satisfied. Additionally, the Pods are labeled with controller-uid, this acts as a unique ID for that specific Job.

When done, the Pods persist in a Completed state. They are not deleted after the Job is completed or failed. This is intentional to better support troubleshooting.

1. A summary of these events can be seen by describing the Job itself.

$ kubectl describe job job-example

1. Delete the job.

$ kubectl delete job job-example

1. View the Pods once more.

$ kubectl get pods

The Pods will now be deleted. They are cleaned up when the Job itself is removed.

Summary: Jobs are fire and forget one off tasks, batch processing or as an executor for a workflow engine. They *"run to completion"* or terminate gracefully adhering to the completions and parallelism directives.

### Exercise: Scheduling a CronJob

Objective: Create a CronJob based off a Job Template. Understand how the Jobs are generated and how to suspend a job in the event of a problem.

1. Create CronJob cronjob-example based off the yaml below, or use the manifest manifests/cronjob-example.yaml It is configured to run the Job from the earlier example every minute, using the cron schedule "\*/1 \* \* \* \*". This schedule is UTC ONLY.

manifests/cronjob-example.yaml

apiVersion: batch/v1beta1

kind: CronJob

metadata:

name: cronjob-example

spec:

schedule: "\*/1 \* \* \* \*"

successfulJobsHistoryLimit: 2

failedJobsHistoryLimit: 1

jobTemplate:

spec:

completions: 4

parallelism: 2

template:

spec:

containers:

- name: hello

image: alpine:latest

command: ["/bin/sh", "-c"]

args: ["echo hello from $HOSTNAME!"]

restartPolicy: Never

Command

$ kubectl create -f manifests/cronjob-example.yaml

1. **Give it some time to run**, and then list the Jobs.

$ kubectl get jobs

There should be at least one Job named in the format <cronjob-name>-<unix time stamp>. Note the timestamp of the oldest Job.

1. Give it a few minutes and list the Jobs once again

$ kubectl get jobs

The oldest Job should have been removed. The CronJob controller will purge Jobs according to the successfulJobHistoryLimit and failedJobHistoryLimit attributes. In this case, it is retaining strictly the last 3 successful Jobs.

1. Describe the CronJob cronjob-example

$ kubectl describe CronJob cronjob-example

The events will show the records of the creation and deletion of the Jobs.

1. Edit the CronJob cronjob-example and locate the Suspend field. Then set it to true.

$ kubectl edit CronJob cronjob-example

This will prevent the cronjob from firing off any future events, and is useful to do to initially troubleshoot an issue without having to delete the CronJob directly.

1. Delete the CronJob

$ kubectl delete cronjob cronjob-example

Deleting the CronJob WILL delete all child Jobs. Use Suspend to *'stop'* the Job temporarily if attempting to troubleshoot.

Summary: CronJobs are a useful extension of Jobs. They are great for backup or other day-to-day tasks, with the only caveat being they adhere to a UTC ONLY schedule.

Clean Up Commands

kubectl delete CronJob cronjob-example

# Volumes

Volumes within Kubernetes are storage that is tied to the Pod’s lifecycle.

A pod can have one or more type of volumes attached to it. These volumes are consumable by any of the containers within the pod.

They can survive Pod restarts; however their durability beyond that is dependent on the Volume Type.

### Exercise: Using Volumes with Pods

Objective: Understand how to add and reference volumes to a Pod and their containers.

1. Create a Pod with from the manifest manifests/volume-example.yaml or the yaml below.

manifests/volume-example.yaml

apiVersion: v1

kind: Pod

metadata:

name: volume-example

spec:

containers:

- name: nginx

image: nginx:stable-alpine

ports:

- containerPort: 80

volumeMounts:

- name: html

mountPath: /usr/share/nginx/html

readOnly: true

- name: content

image: alpine:latest

volumeMounts:

- name: html

mountPath: /html

command: ["/bin/sh", "-c"]

args:

- while true; do

echo $(date)"<br />" >> /html/index.html;

sleep 5;

done

volumes:

- name: html

emptyDir: {}

Command

$ kubectl create -f manifests/volume-example.yaml

Note the relationship between volumes in the Pod spec, and the volumeMounts directive in each container.

1. Exec into content container within the volume-example Pod, and cat the html/index.html file.

$ kubectl exec volume-example -c content -- /bin/sh -c "cat /html/index.html"

You should see a list of date time-stamps. This is generated by the script being used as the entrypoint (args) of the content container.

1. Now do the same within the nginx container, using cat to see the content of /usr/share/nginx/html/index.html example.

$ kubectl exec volume-example -c nginx -- /bin/sh -c "cat /usr/share/nginx/html/index.html"

You should see the same file.

1. Now try to append "nginx" to index.html from the nginx container.

$ kubectl exec volume-example -c nginx -- /bin/sh -c "echo nginx >> /usr/share/nginx/html/index.html"

It should error out and complain about the file being read only. The nginx container has no reason to write to the file, and mounts the same Volume as read-only. Writing to the file is handled by the content container.

Summary: Pods may have multiple volumes using different Volume types. Those volumes in turn can be mounted to one or more containers within the Pod by adding them to the volumeMounts list. This is done by referencing their name and supplying their mountPath. Additionally, volumes may be mounted both read-write or read-only depending on the application, enabling a variety of use-cases.

Clean Up Command

kubectl delete pod volume-example

# 

# Persistent Volumes and Claims

Persistent Volumes and Claims work in conjunction to serve as the direct method in which a Pod Consumes Persistent storage.

A PersistentVolume (PV) is a representation of a cluster-wide storage resource that is linked to a backing storage provider - NFS, GCEPersistentDisk, RBD etc.

A PersistentVolumeClaim acts as a namespaced *request* for storage that satisfies a set of requirements instead of mapping to the storage resource directly.

This separation of PV and PVC ensures that an application’s *‘claim’* for storage is portable across numerous backends or providers.

### Exercise: Understanding Persistent Volumes and Claims

Objective: Gain an understanding of the relationship between Persistent Volumes, Persistent Volume Claims, and the multiple ways they may be selected.

1. Create PV pv-sc-example from the manifest manifests/pv-sc-example.yaml or use the yaml below. Ensure to note that its labeled with type=hostpath, its Storage Class Name is set to mypvsc, and uses Delete for the Reclaim Policy.

manifests/pv-sc-example.yaml

kind: PersistentVolume

apiVersion: v1

metadata:

name: pv-sc-example

labels:

type: hostpath

spec:

capacity:

storage: 2Gi

accessModes:

- ReadWriteMany

persistentVolumeReclaimPolicy: Delete

storageClassName: mypvsc

hostPath:

type: DirectoryOrCreate

path: "/data/mypvsc"

Command

$ kubectl create -f manifests/pv-sc-example.yaml

1. Once created, list the available Persistent Volumes.

$ kubectl get pv

You should see the single PV pv-sc-example flagged with the status Available. Meaning no claim has been issued that targets it.

1. Create PVC pvc-selector-example from the manifest manifests/pvc-selector-example.yaml or the yaml below.

manifests/pvc-selector-example.yaml

kind: PersistentVolumeClaim

apiVersion: v1

metadata:

name: pvc-selector-example

spec:

accessModes:

- ReadWriteMany

resources:

requests:

storage: 1Gi

selector:

matchLabels:

type: hostpath

Command

$ kubectl create -f manifests/pvc-selector-example.yaml

Note that the selector targets type=hostpath.

1. Then describe the newly created PVC

$ kubectl describe pvc pvc-selector-example

The pvc pvc-selector-example should be in a Pending state with the Error Event FailedBinding and no Persistent Volumes available for this claim and no storage class is set. If a PV is given a storageClassName, ONLY PVCs that request that Storage Class may use it, even if the selector has a valid target.

1. Now create the PV pv-selector-example from the manifest manifests/pv-selector-example.yaml or the yaml below.

manifests/pv-selector-example.yaml

kind: PersistentVolume

apiVersion: v1

metadata:

name: pv-selector-example

labels:

type: hostpath

spec:

capacity:

storage: 2Gi

accessModes:

- ReadWriteMany

hostPath:

type: DirectoryOrCreate

path: "/data/mypvselector"

Command

$ kubectl create -f manifests/pv-selector-example.yaml

1. Give it a few moments and then look at the Persistent Volumes once again.

$ kubectl get pv

The PV pv-selector-example should now be in a Bound state, meaning that a PVC has been mapped or *"bound"* to it. Once bound, NO other PVCs may make a claim against the PV.

1. Create the pvc pvc-sc-example from the manifest manifests/pvc-sc-example.yaml or use the yaml below.

manifests/pvc-sc-example.yaml

kind: PersistentVolumeClaim

apiVersion: v1

metadata:

name: pvc-sc-example

spec:

accessModes:

- ReadWriteMany

storageClassName: mypvsc

resources:

requests:

storage: 1Gi

Command

$ kubectl create -f manifests/pvc-sc-example.yaml

Note that this PVC has a storageClassName reference and no selector.

1. Give it a few seconds and then view the current PVCs.

$ kubectl get pvc

The pvc-sc-example should be bound to the pv-sc-example Volume. It consumed the PV with the corresponding storageClassName.

1. Delete both PVCs.

$ kubectl delete pvc pvc-sc-example pvc-selector-example

1. Then list the PVs once again.

$ kubectl get pv

The pv-sc-example will not be listed. This is because it was created with a persistentVolumeReclaimPolicy of Delete meaning that as soon as the PVC was deleted, the PV itself was deleted.

PV pv-selector-example, was created without specifying a persistentVolumeReclaimPolicy and was in turn created with the default for PVs: Retain. It's state of Released means that it's associated PVC has been deleted. In this state no other PVC's may claim it, even if pvc-selector-example was created again. The PV must manually be reclaimed or deleted. This ensures the preservation of the state of the Volume in the event that its PVC was accidentally deleted giving an administrator time to do something with the data before reclaiming it.

1. Delete the PV pv-selector-example.

$ kubectl delete pv pv-selector-example

Summary: Persistent Volumes and Persistent Volume Claims when bound together provide the primary method of attaching durable storage to Pods. Claims may reference PVs by specifying a storageClassName, targeting them with a selector, or a combination of both. Once a PV is bound to a PVC, it becomes a tightly coupled relationship and no further PVCs may issue a claim against the PV, even if the binding PVC is deleted. How PVs are reclaimed is configured via the PV attribute persistentVolumeReclaimPolicy where they can either be deleted automatically when set to Delete or require manual intervention when set to Retain as a data-preservation safe-guard.

### Exercise: Using PersistentVolumeClaims

Objective: Learn how to consume a Persistent Volume Claim within a Pod, and explore some of the ways they may be used.

1. Create PV and associated PVC html using the manifest manifests/html-vol.yaml

manifest/html-vol.yaml

kind: PersistentVolume

apiVersion: v1

metadata:

name: html

labels:

type: hostpath

spec:

capacity:

storage: 1Gi

accessModes:

- ReadWriteMany

storageClassName: html

persistentVolumeReclaimPolicy: Delete

hostPath:

type: DirectoryOrCreate

path: "/tmp/html"

---

kind: PersistentVolumeClaim

apiVersion: v1

metadata:

name: html

spec:

accessModes:

- ReadWriteMany

storageClassName: html

resources:

requests:

storage: 1Gi

Command

$ kubectl create -f manifests/html-vol.yaml

1. Create Deployment writer from the manifest manifests/writer.yaml or use the yaml below. It is similar to the volume-example but now uses a persistentVolumeClaim Volume instead of an emptyDir.

manifests/writer.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: writer

spec:

replicas: 1

selector:

matchLabels:

app: writer

template:

metadata:

labels:

app: writer

spec:

containers:

- name: content

image: alpine:latest

volumeMounts:

- name: html

mountPath: /html

command: ["/bin/sh", "-c"]

args:

- while true; do

date >> /html/index.html;

sleep 5;

done

volumes:

- name: html

persistentVolumeClaim:

claimName: html

Command

$ kubectl create -f manifests/writer.yaml

Note that the claimName references the previously created PVC defined in the html-vol manifest.

1. Create a Deployment and Service reader from the manifest manifests/reader.yaml or use the yaml below.

manifests/reader.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: reader

spec:

replicas: 3

selector:

matchLabels:

app: reader

template:

metadata:

labels:

app: reader

spec:

containers:

- name: nginx

image: nginx:stable-alpine

ports:

- containerPort: 80

volumeMounts:

- name: html

mountPath: /usr/share/nginx/html

readOnly: true

volumes:

- name: html

persistentVolumeClaim:

claimName: html

---

apiVersion: v1

kind: Service

metadata:

name: reader

spec:

selector:

app: reader

ports:

- protocol: TCP

port: 80

targetPort: 80

Command

$ kubectl create -f manifests/reader.yaml

1. With the reader Deployment and Service created, use kubectl proxy to view the reader Service.

$ kubectl proxy

URL

http://127.0.0.1:8001/api/v1/namespaces/default/services/reader/proxy/

The reader Pods can reference the same Claim as the writer Pod. This is possible because the PV and PVC were created with the access mode ReadWriteMany.

1. Now try to append "nginx" to index.html from one of the reader Pods.

$ kubectl exec reader-<pod-hash>-<pod-id> -- /bin/sh -c "echo nginx >> /usr/share/nginx/html/index.html"

The reader Pods have mounted the Volume as read only. Just as it did with exercise 1, The command should error out with a message complaining about not being able to modify a read-only filesystem.

Summary: Using Persistent Volume Claims with Pods is quite easy. The attribute persistentVolumeClaim.claimName simply must reference the name of the desired PVC in the Pod's Volume definition. Multiple Pods may reference the same PVC as long as their access mode supports it.

Clean Up Command

kubectl delete -f manifests/reader.yaml -f manifests/writer.yaml -f manifests/html-vol.yaml

# 

# StatefulSets

The StatefulSet controller is tailored to managing Pods that must persist or maintain state. Pod identity including hostname, network, and storage can be considered persistent.

They ensure persistence by making use of three things:

* The StatefulSet controller enforcing predictable naming, and ordered provisioning/updating/deletion.
* A headless service to provide a unique network identity.
* A volume template to ensure stable per-instance storage.

### Exercise: Managing StatefulSets

Objective: Create, update, and delete a StatefulSet to gain an understanding of how the StatefulSet lifecycle differs from other workloads with regards to updating, deleting and the provisioning of storage.

1. Create StatefulSet sts-example using the yaml block below or the manifest manifests/sts-example.yaml.

manifests/sts-example.yaml

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: sts-example

spec:

replicas: 3

revisionHistoryLimit: 3

selector:

matchLabels:

app: stateful

serviceName: app

updateStrategy:

type: OnDelete

template:

metadata:

labels:

app: stateful

spec:

containers:

- name: nginx

image: nginx:stable-alpine

ports:

- containerPort: 80

volumeMounts:

- name: www

mountPath: /usr/share/nginx/html

volumeClaimTemplates:

- metadata:

name: www

spec:

accessModes: [ "ReadWriteOnce" ]

storageClassName: mypvsc

resources:

requests:

storage: 1Gi

Command

$ kubectl create -f manifests/sts-example.yaml

1. Immediately watch the Pods being created.

$ kubectl get pods --show-labels --watch

Unlike Deployments or DaemonSets, the Pods of a StatefulSet are created one-by-one, going by their ordinal index. Meaning, sts-example-0 will fully be provisioned before sts-example-1 starts up. Additionally, take notice of the controller-revision-hash label. This serves the same purpose as the controller-revision-hash label in a DaemonSet or the pod-template-hash in a Deployment. It provides a means of tracking the revision of the Pod Template and enables rollback functionality.

1. More information on the StatefulSet can be gleaned about the state of the StatefulSet by describing it.

$ kubectl describe statefulset sts-example

Within the events, notice that it is creating claims for volumes before each Pod is created.

1. View the current Persistent Volume Claims.

$ kubectl get pvc

The StatefulSet controller creates a volume for each instance based off the volumeClaimTemplate. It prepends the volume name to the Pod name. e.g. www-sts-example-0.

1. Update the StatefulSet's Pod Template and add a few additional labels.

$ kubectl apply -f manifests/sts-example.yaml --record

< or >

$ kubectl edit statefulset sts-example --record

1. Return to watching the Pods.

$ kubectl get pods --show-labels

None of the Pods are being updated to the new version of the Pod.

1. Delete the sts-example-2 Pod.

$ kubectl delete pod sts-example-2

1. Immediately get the Pods.

$ kubectl get pods --show-labels --watch

The new sts-example-2 Pod should be created with the new additional labels. The OnDelete Update Strategy will not spawn a new iteration of the Pod until the previous one was deleted. This allows for manual gating the update process for the StatefulSet.

1. Update the StatefulSet and change the Update Strategy Type to RollingUpdate.

$ kubectl apply -f manifests/sts-example.yaml --record

< or >

$ kubectl edit statefulset sts-example --record

1. Immediately watch the Pods once again.

$ kubectl get pods --show-labels --watch

Note that the Pods are sequentially updated in descending order, or largest to smallest based on the Pod's ordinal index. This means that if sts-example-2 was not updated already, it would be updated first, then sts-example-1 and finally sts-example-0.

1. Delete the StatefulSet sts-example

$ kubectl delete statefulset sts-example

1. View the Persistent Volume Claims.

$ kubectl get pvc

Created PVCs are NOT garbage collected automatically when a StatefulSet is deleted. They must be reclaimed independently of the StatefulSet itself.

1. Recreate the StatefulSet using the same manifest.

$ kubectl create -f manifests/sts-example.yaml --record

1. View the Persistent Volume Claims again.

$ kubectl get pvc

Note that new PVCs were NOT provisioned. The StatefulSet controller assumes if the matching name is present, that PVC is intended to be used for the associated Pod.

Summary: Like many applications where state must be taken into account, the planning and usage of StatefulSets requires forethought. The consistency brought by standard naming, ordered updates/deletes and templated storage does however make this task easier.

### Exercise: Understanding StatefulSet Network Identity

Objective: Create a *"headless service"* or a service without a ClusterIP (ClusterIP=None) for use with the StatefulSet sts-example, then explore how this enables consistent service discovery.

1. Create the headless service app using the app=stateful selector from the yaml below or the manifest manifests/service-sts-example.yaml.

manifests/service-sts-example.yaml

apiVersion: v1

kind: Service

metadata:

name: app

spec:

clusterIP: None

selector:

app: stateful

ports:

- protocol: TCP

port: 80

targetPort: 80

Command

$ kubectl create -f manifests/service-sts-example.yaml

1. Describe the newly created service

$ kubectl describe svc app

Notice that it does not have a clusterIP, but does have the Pod Endpoints listed. Headless services are unique in this behavior.

1. Query the DNS entry for the app service.

$ kubectl exec sts-example-0 -- nslookup app.default.svc.cluster.local

An A record will have been returned for each instance of the StatefulSet. Querying the service directly will do simple DNS round-robin load-balancing.

1. Finally, query one of instances directly.

$ kubectl exec sts-example-0 -- nslookup sts-example-1.app.default.svc.cluster.local

This is a unique feature to StatefulSets. This allows for services to directly interact with a specific instance of a Pod. If the Pod is updated and obtains a new IP, the DNS record will immediately point to it enabling consistent service discovery.

Summary: StatefulSet service discovery is unique within Kubernetes in that it augments a headless service (A service without a unique ClusterIP) to provide a consistent mapping to the individual Pods. These mappings take the form of an A record in format of: <StatefulSet Name>-<ordinal>.<service name>.<namespace>.svc.cluster.local and can be used consistently throughout other Workloads.

Clean Up Command

kubectl delete svc app

kubectl delete statefulset sts-example

kubectl delete pvc www-sts-example-0 www-sts-example-1 www-sts-example-2

**Misc learnings:**

1. **Resource constrained POD**

**Resource limits**

kubectl get pods

nano resource-limited-pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: frontend

spec:

containers:

- name: db

image: mysql

env:

- name: MYSQL\_ROOT\_PASSWORD

value: "password"

resources:

requests:

memory: "64Mi"

cpu: "250m"

limits:

memory: "128Mi"

cpu: "500m"

- name: wp

image: wordpress

resources:

requests:

memory: "64Mi"

cpu: "250m"

limits:

memory: "128Mi"

cpu: "500m"

kubectl create -f resource-limited-pod.yaml

kubectl get pods

kubectl describe pods frontend

kubectl explain pod.spec.containers.resources.requests

1. **Init Containers in POD**

**Pod initialization**

kubectl get pods

nano init.yaml

apiVersion: v1

kind: Pod

metadata:

name: init-demo

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

volumeMounts:

- name: workdir

mountPath: /usr/share/nginx/html

# These containers are run during pod initialization

initContainers:

- name: install

image: busybox

command:

- wget

- "-O"

- "/work-dir/index.html"

- http://kubernetes.io

volumeMounts:

- name: workdir

mountPath: "/work-dir"

dnsPolicy: Default

volumes:

- name: workdir

emptyDir: {}

kubectl create -f init.yaml

kubectl get pods

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

apt-get update

apt-get install curl

curl localhost

1. **Liveness and Readiness Probes**

[**https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/**](https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/)

kubectl get pods

nano exec-liveness.yaml

apiVersion: v1

kind: Pod

metadata:

labels:

test: liveness

name: liveness-exec

spec:

containers:

- name: liveness

image: k8s.gcr.io/busybox

args:

- /bin/sh

- -c

- touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600

livenessProbe:

exec:

command:

- cat

- /tmp/healthy

initialDelaySeconds: 5

periodSeconds: 5

kubectl create exec-liveness.yaml

(wait 5 sec)

kubectl describe pod liveness-exec

(wait 30 sec)

kubectl describe pod liveness-exec

The pods will get restarted after every 60 seconds when the tmp file is delete and there is probe which is done automatically.

nano http-liveness.yaml

apiVersion: v1

kind: Pod

metadata:

labels:

test: liveness

name: liveness-http

spec:

containers:

- name: liveness

image: k8s.gcr.io/liveness

args:

- /server

livenessProbe:

httpGet:

path: /healthz

port: 8080

httpHeaders:

- name: X-Custom-Header

value: Awesome

initialDelaySeconds: 3

periodSeconds: 3

kubectl create -f http-liveness.yaml

(wait 30 sec)

kubectl describe pod liveness-http

code in the container –

http.HandleFunc("/healthz", **func**(w http.ResponseWriter, r \*http.Request) {

duration := time.Now().Sub(started)

**if** duration.Seconds() > 10 {

w.WriteHeader(500)

w.Write([]byte(fmt.Sprintf("error: %v", duration.Seconds())))

} **else** {

w.WriteHeader(200)

w.Write([]byte("ok"))

}

})

nano tcp-liveness-readiness.yaml

apiVersion: v1

kind: Pod

metadata:

name: goproxy

labels:

app: goproxy

spec:

containers:

- name: goproxy

image: k8s.gcr.io/goproxy:0.1

ports:

- containerPort: 8080

readinessProbe:

tcpSocket:

port: 8080

initialDelaySeconds: 5

periodSeconds: 10

livenessProbe:

tcpSocket:

port: 8080

initialDelaySeconds: 15

periodSeconds: 20

kubectl create -f tcp-liveness-readiness.yaml

(wait 20 sec)

kubectl describe pod goproxy

1. **Container lifecycle events**

kubectl get pods

nano lifecycle.yaml

apiVersion: v1

kind: Pod

metadata:

name: lifecycle-demo

spec:

containers:

- name: lifecycle-demo-container

image: nginx

lifecycle:

postStart:

exec:

command: ["/bin/sh", "-c", "echo Hello from the postStart handler > /usr/share/message"]

preStop:

exec:

command: ["/usr/sbin/nginx","-s","quit"]

kubectl create -f lifecycle.yaml

kubectl get pod lifecycle-demo

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

cat /usr/share/message

1. **Run to completion JOB:**

kubectl get pods

nano job.yaml

apiVersion: batch/v1

kind: Job

metadata:

name: pi

spec:

template:

spec:

containers:

- name: pi

image: perl

command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]

restartPolicy: Never

backoffLimit: 4

kubectl create -f job.yaml

kubectl describe jobs/pi

pods=$(kubectl get pods --show-all --selector=job-name=pi --output=jsonpath={.items..metadata.name})

echo $pods

kubectl logs $pods

1. **Secrets and ConfigMap**
   1. **secret**

echo -n 'my-app' | base64

echo -n 'P@ssw0rd' | base64

nano secret.yaml

apiVersion: v1

kind: Secret

metadata:

name: test-secret

data:

username: bXktYXBw

password: UEBzc3cwcmQ=

kubectl create -f secret.yaml

kubectl get secret test-secret

kubectl describe secret test-secret

nano secret-pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: secret-test-pod

spec:

containers:

- name: test-container

image: nginx

volumeMounts:

# name must match the volume name below

- name: secret-volume

mountPath: /etc/secret-volume

# The secret data is exposed to Containers in the Pod through a Volume.

volumes:

- name: secret-volume

secret:

secretName: test-secret

kubectl create -f secret-pod.yaml

kubectl get pod secret-test-pod

kubectl exec -it secret-test-pod -- /bin/bash

cd /etc/secret-volume

ls

cat username; echo; cat password; echo

[**https://kubernetes.io/docs/concepts/configuration/secret/**](https://kubernetes.io/docs/concepts/configuration/secret/)

* 1. **configmap**

nano configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: special-config

namespace: default

data:

special.level: very

special.type: charm

nano config-pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: dapi-test-pod

spec:

containers:

- name: test-container

image: k8s.gcr.io/busybox

command: [ "/bin/sh", "-c", "env" ]

envFrom:

- configMapRef:

name: special-config

restartPolicy: Never

kubectl get pods

kubectl exec -it podname -- /bin/bash

"ls /etc/config/"

**Exercise from official Documentation:**

**Assigning Pods to Node using Nodeselector:**

[**https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes/**](https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes/)

**Horizontal Pod Auto scaling:**

[**https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/**](https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/)

**Stateful MySQl**

[**https://kubernetes.io/docs/tasks/run-application/run-single-instance-stateful-application/**](https://kubernetes.io/docs/tasks/run-application/run-single-instance-stateful-application/)

