Kubernetes For Beginners – An Introduction Part 2

Charles Wimmer

12 October 2019

This document provides an introduction to Kubernetes for the uninitiated. It is intended to give application developers enough information to get them started with Kubernetes deployments. This is part two in a series.

1 PersistentVolumes

PersistentVolumes are part of the storage subsystem in Kubernetes. They provide a set of APIs that abstract the implementation of the storage from the way it is consumed.[1]

[1] Persistent volumes. `https://kubernetes.io/docs/concepts/storage/persistent-volumes/`. Accessed: 2019-10-12

PersistentVolumes are storage resources that have a life cycle independent from the Pods that use it. A PersistentVolume may exist before Pods request it. Kubernetes may retain a PersistentVolume after destroying a pod that requested it.

Figure 1: PersistentVolume

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 100Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: nfs-filer
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /my/nfs/path
    server: 192.168.1.1
```

PersistentVolumes are cluster scoped resources similar to nodes.

PersistentVolumeClaims are Namespace scoped resources. When Kubernetes matches a PersistentVolumeClaim to a PersistentVolume, that volume is bound to a Namespace.

Figure 2: PersistentVolumeClaim

```
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 100Gi
  storageClassName: nfs-filer
```

Once bound to a Namespace, a PersistentVolume may then be mounted into a pod as a Volume.

Figure 3: Pod using ConfigMap item as file

```
---
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
      - mountPath: "/var/logs"
        name: my-logs
  volumes:
    - name: my-logs
      persistentVolumeClaim:
        claimName: my-pvc
```

PersistentVolumes may be created in two ways, statically or dynamically. Static creation is when an administrator creates PersistenVolumes. Dynamic creation occurs on-demand after a StorageClass is created and configured to support it. Kubernetes creates PersistentVolumes after the PersistentVolumeClaim is created, rather than before it.

Upon deletion, Kubernetes applies its persistentVolumeReclaim-

Policy. The valid values for this attribute are Retain, Recycle or Delete.

Application authors should use the Retain policy when describing a shared volume that is intended to be passed around to many pods over its lifetime. For example, if an NFS server had a large, static dataset on one of its mount points. The application author could instruct Kubernetes to retain the dataset.

Recycle instructs Kubernetes to do a simple cleaning such as `rm -rf` on the filesystem before handing it to the next consumer.

Delete instructs Kubernetes to delete the backing store when a Pod releases a claim.

TODO: Insert two diagrams of the PV lifecycle. One for static and one for dynamic.

2 *DaemonSets*

A DaemonSet defines a Pod that should be run on all nodes of a cluster.[2] The DaemonSet definition may restrict which nodes to run on based on matching a NodeSelector.[3] Nodes may reject Pods from running by applying Taints[4] which the DaemonSet does not Tolerate.

TODO: Cite some examples for DaemonSet

[2] Daemonset. `https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/`. Accessed: 2019-10-12

[3] Assigning pods to nodes. `https://kubernetes.io/docs/concepts/configuration/assign-pod-node/`. Accessed: 2019-10-12

[4] Taints and tolerations. `https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/`. Accessed: 2019-10-12

Figure 4: DaemonSet

```
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      nodeSelector:
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image: k8s.gcr.io/fluentd-elasticsearch:1.20
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

3 StatefulSets

Similar to Deployments, StatefulSets are workload controllers that launch a similar set of Pods. Unlike a Deployment, StatefulSets provide guarantees about ordering, uniqueness, and network identity of those pods.[5]

[5] Statefulsets. `https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/`. Accessed: 2019-10-12

StatefulSets create Pods sequentially in order {0..N-1}. StatefulSets delete Pods sequentially in reverse order {N-1..0}. Before starting up to the next Pod, StatefulSets guarantee that all predecessors are Running and Ready. Before terminating a Pod, StatefulSets guarantee all successors have finished terminating.

StatefulSets provide a unique, consistent network identity for each Pod. It does this by providing a unique DNS entry for each Pod. To begin with, each StatefulSet must be associated with a Service. Given this relationship, each Pod in a StatefulSet is assigned a DNS record of the form:

`${STATEFULSET_NAME}-${ORDINAL}.${SERVICE}.${NAMESPACE}.svc.cluster.local.`
For example, the third ordinal in a StatefulSet named 'broker' which is associated with a Service named 'kafka' started in the 'platform' Namespace would have the DNS record

`broker-3.kafka.platform.svc.cluster.local.`

Application Authors may change some of the guarantees about ordering. StatefulSets have a field named `podManagementPolicy` which defaults to `OrderedReady`. When set to the default, the ordering characteristics described above apply. When set to `Parallel`, all Pods are launched and terminated in parallel. The ordering guarantees are relaxed.

Figure 5: StatefulSet Example

```
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx # has to match .spec.template.metadata.labels
  serviceName: "nginx"
  replicas: 3 # by default is 1
  template:
    metadata:
      labels:
        app: nginx # has to match .spec.selector.matchLabels
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
        image: k8s.gcr.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "my-storage-class"
      resources:
        requests:
          storage: 1Gi
```

4 Jobs

Kubernetes supports the concept of a Job that is launched and runs until completion.[6] The Job controller implements a pattern that may be used by typical batch jobs.

[6] Jobs - run to completion. https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/. Accessed: 2019-10-12

This controller is well suited to a variety of workloads. Here are some examples:

- Non-parallel: The Job controller starts one Pod. Job is complete when Pod is complete.
- Parallel with fixed completion count: Job controller launches multiple Pods. One Pod must complete for each partition.
- Parallel with a work queue: Multiple pods launched. Job is complete when one pod exits successfully.
- Deadline: A Job controller kills a Pod after it has been running for a defined number of seconds.

Application authors may configure Job controllers in how they handle Pod and container failures. The `restartPolicy` field may be set to `OnFailure` or `Never`. Jobs are not appropriate for Pods that should run continuously. Consider a Deployment for those workloads. The `backoffLimit` field is used to configure the number of Pod restarts that the Job controller should attempt.

Figure 6: Job Example

```
---
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
```

5 CronJobs

The CronJob controller is closely related to the Job controller. A CronJob launches a Job at specific times.[7] It may be configured to launch a

[7] Cronjob. https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/. Accessed: 2019-10-12

Job once or repeated at specific times.

The CronJob does not have firm guarantees about launching jobs. Jobs are created "about once" per period. Due to the distributed nature of this controller, the following edge cases may occur:

- CronJob controller may create two Jobs.
- CronJob controller may create zero Jobs.

While these are rare edge cases, each Job must be able to handle these conditions.

Figure 7: CronJob Example

```
---
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: periodic-pi
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: pi
            image: perl
            command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
            restartPolicy: Never
          backoffLimit: 4
```

6 Container Health Checks

Kubernetes keeps distributed applications up and running. By default, this means ensuring the containers defined in a Pod continue to run. Sometimes applications enter a state where the application is unhealthy even though the processes are still running. Sometimes an application is healthy, but is temporarily overloaded and should not be asked to serve any more clients. Kubernetes has extension points that allow application authors to define what a healthy, well-behaving system looks like.[8] Using this information, Kubernetes has more tools to ensure the health of the application.

[8] Configure liveness, readiness and startup probes. `https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/`. Accessed: 2019-10-12

Three user-configurable probes inform Kubernetes about an application's health.

An application author may define a livenessProbe that describes if a container is healthy. If a livenessProbe fails, Kubernetes kills and restarts the container.

An application author may define a readinessProbe that describes if a container is capable of handline more load. If the readinessProbe fails, Kubernetes removes it from the list of Endpoints in the Service matching the Pod.

An application author may define a startupProbe[9] that describes if a container has completed its startup process. If a container has startupProbe defined, it does not use the livenessProbe on startup. After the startupProbe succeeds once, the livenessProbe takes over.

[9] startupProbes are available in Kuberetes 1.16 and newer

An application author may configure each of the probe types with one of three types of tests. HTTP(S), TCP, or Container Exec. For an HTTP probe, any HTTP response code greater or equal to 200 and less than 400 indicates success. For a TCP probe, a successful three-way handshake[10] indicates health. A Container Exec probe launches a process from inside the container. If that process returns a zero result code, the container is considered healthy.

[10] Handshaking. `https://en.wikipedia.org/wiki/Handshaking`. Accessed: 2019-10-12

An application author may configure each probe with timeouts and retries as appropriate.

Figure 8: Container Exec Probe Example

```
---
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
```

Figure 9: HTTP Probe Example

```
---
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: Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

Figure 10: TCP Probe Example

```
---
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
```

7 *NetworkPolicies*

By default, Kubernetes does not isolate Pods at the network level. All Pods may access network endpoints both within and outside the cluster. NetworkPolicies allow the application author to define which endpoints may be accessed by which Pods.

Conceptually, NetworkPolicies implement a Pod based firewall. When network policies are defined, they select zero or more Pods. When a NetworkPolicy selects a Pod, Kubernetes adds the rules in the policy to the rules for the Pod. By default, Pods accept traffic from any source. If a NetworkPolicy selects a pod, Kubernetes rejects all traffic unless it is explicitly allowed by policy.

Figure 11: NetworkPolicy Example

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5979
```

References

Cronjob. `https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/`. Accessed: 2019-10-12.

Daemonset. `https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/`. Accessed: 2019-10-12.

Jobs - run to completion. `https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/`. Accessed: 2019-10-12.

Assigning pods to nodes. `https://kubernetes.io/docs/concepts/configuration/assign-pod-node/`. Accessed: 2019-10-12.

Persistent volumes. `https://kubernetes.io/docs/concepts/storage/persistent-volumes/`. Accessed: 2019-10-12.

Configure liveness, readiness and startup probes. `https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/`. Accessed: 2019-10-12.

Statefulsets. `https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/`. Accessed: 2019-10-12.

Taints and tolerations. `https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/`. Accessed: 2019-10-12.

Handshaking. `https://en.wikipedia.org/wiki/Handshaking`. Accessed: 2019-10-12.