Kubernetes For Beginners – An Introduction Part 3

Charles Wimmer

28 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 three in a series.

1 Container Lifecycle

Kubernetes provides application administrators with two hooks into the container lifecycle: PostStart and PreStop.[1] These hooks have two implementations. A hook may run a specific command inside the namespaces of the container, or it may execute an HTTP request against a port on the container.

[1] Container lifecycle hooks. `https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/`. Accessed: 2019-10-15

Immediately after Kubelet creates the container, it fires the PostStart hook. There is no guarantee it fires before the container's entrypoint. There is also no guarantee that it fires after the container's entrypoint. Due to these constraints, the use of the HTTP request as a PostStart hook is not consistent. If the request fires before the container has initialized the HTTP service, the request will not be received.

The PreStop hook is a bit more flexible than the PostStart hook. Kubelet fires the PreStop hook immediately before container termination. By default, it is blocking. That is to say that the container does not get terminated until after the PreStop hook completes. Hooks must complete within the container's termination grace period.

Figure 1: Container Lifecycle Example

```
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"]
```

2 *Authenticating to the Kubernetes API*

When accessing the Kubernetes API, Kubernetes either authenticates it as a Service Account or Normal User; otherwise, the request is considered anonymous.[2]

User accounts are intended to represent humans, and the Kubernetes API does not manage them. The Kubernetes API does not have representations of Normal user accounts. User accounts are authenticated by mechanisms outside of Kubernetes, as well. User accounts have a cluster scope; they are not tied to Namespaces.

Application Authors create Service Accounts by calls to the Kubernetes API. They have a record and authentication information stored in the Kubernetes database. Application Authors can include definitions of Service Accounts when creating applications. Service Accounts are Namespace scoped objects.

The table below summarizes the differences between Normal user accounts and Service Accounts.

[2] Authenticating. `https://kubernetes.io/docs/reference/access-authn-authz/authentication/`. Accessed: 2019-10-16

	Normal User	Service Account
Typically used for	Humans	Pods
Scope	Global	Namespace
Included in application definition?	No	Yes
Managed by Kubernetes	No	Yes
Typically authenticated by	External entity (e.g. LDAP/AD)	Internal (e.g. Tokens)
Represented by Kubernetes Object	No	Yes

–

Three controllers hosted by the Controller Manager and API

Server automate the lifecycle of Service Accounts.[3]

[3] Managing service accounts. https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/. Accessed: 2019-10-18

2.1 *Service Account Admission Controller*

The Service Account Admission Controller is a component of the API Server. It acts on each call to add or update a Pod to ensure the following:

- Each Pod must have a Service Account. If the Pod definition does not, the controller sets the Service Account to `Default`
- If a Pod does already have a Service Account defined, the controller ensures the Service Account exists. If the Service account does not exist, the controller rejects the Pod.
- If the Pod definition does not override the `ImagePullSecrets` field, then the controller adds the `ImagePullSecrets` from the Service Account.
- The controller adds a `Volume` to the Pod containing the token for API access.
- The controller adds a `volumeSource` to the container mounting the token for API access.

2.2 *Token Controller*

The Token Controller runs as part of the Controller Manager. It performs the following functions:

- Watches Service Account creation. When it observes a new Service Account, it ensures a matching API token secret exists.
- Watches Service Account deletion. When it observes a deletion, it removes the matching API token secret.
- Watches Secret creation of type `ServiceAccountToken`. When it observes a new Secret, it creates an API token for the Secret.
- Watches Secret deletion of type `ServiceAccountToken`. It invalidates API tokens as necessary.

The Token Controller signs tokens with a specific key. The same key must be used by the API Server to validate the tokens.

2.3 *Service Account Controller*

The Service Account Controller runs as part of the Controller Manager. It watches Namespaces and ensures each Namespace has a Default Service Account.

3 Container Compute Resource Management

At its core, Kubernetes is a resource management system. All the nodes of a cluster contribute CPU, memory, memory hugepages, disk, network, and potentially other resources to a collective pool. The Kubernetes scheduler matches resource requests with available resources. Kubelet and the container runtime ensure the workloads do not exceed those resource requests.

When requesting resources, the application author specifies a Request and a Limit. The Request is the minimum resource required by the Pod before the kube-scheduler assigns the Pod to a node. The Limit is the maximum amount of resources a pod should be allowed to use before taking corrective action.

Figure 2: Example of Pod with resource requests

```
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
spec:
  containers:
  - name: ubuntu
    image: ubuntu:trusty
    command: ["echo"]
    args: ["Hello World"]
    resources:
      limits:
        cpu: "1"
        memory: 200Mi
        hugepages-2Mi: 80Mi
        ephemeral-storage: 100Gi
      requests:
        cpu: 500m
        memory: 100Mi
        hugepages-2Mi: 80Mi
        ephemeral-storage: 1Ti
```

3.1 Handling Resource Limits

When creating a container with CPU limits, the container runtime is instructed to create a Cgroup with a hard cap. The Cgroup constrains the Pod from using more than its allocation of CPU.

When creating a container with a memory limit, a container is

eligible for termination when exceeding this value. If killed due to this out of memory condition, Kubelet will restart the container just like any other container failure.

When creating a Pod with a hugepages limit, the Request and Limit must be equal. Kubelet assigns hugepages to a Pod at startup. Overcommitting hugepages is not allowed. If an application requests more than its hugepages Limit, the allocation will fail.

When creating a Pod with ephemeral storage limits, the requested amount is considered by the scheduler when assigning a Pod to a Node. If a Pod's usage of ephemeral storage exceeds the sum of the container limits, Kubelet evicts the Pod from the node. The workload controller that started the Pod will be notified it is in a final state and start a new Pod to replace it.

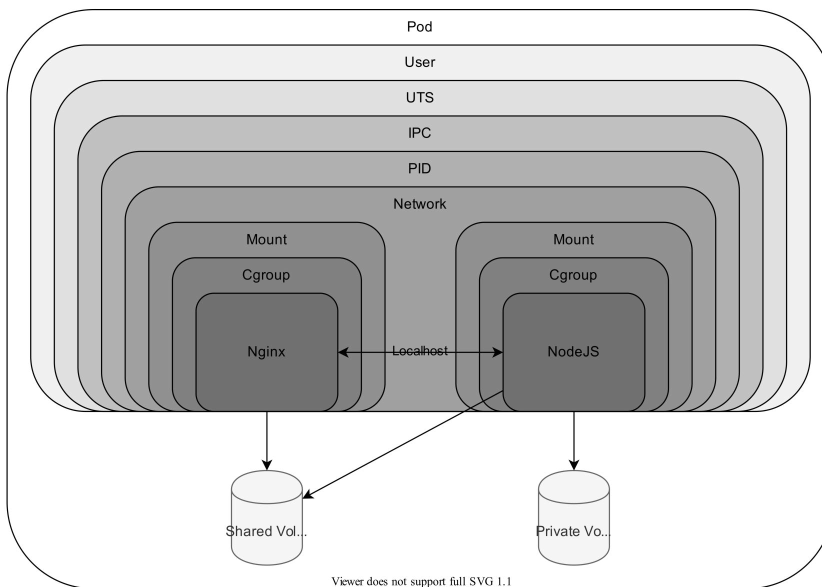

Figure 3: Namespaces and Volumes in a Pod

4 Pod Quality of Service Classes

When scheduled, each pod is placed into one of three Quality of Service (QoS) classes.[4] The classes are:

- Guaranteed
- Burstable
- BestEffort

[4] Configure quality of service for pods. `https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/`. Accessed: 2019-10-19

4.1 Guaranteed QoS

To be considered in the Guaranteed QoS, a pod must meet the following criteria:

- Every container in the Pod must have a CPU Request set.
- Every container in the pod must have a CPU Limit set.
- The CPU Request for every container must be equal to the CPU Limit for that container.
- Every container in the Pod must have a memory Request set.
- Every container in the pod must have a memory Limit set.
- The memory Request for every container must be equal to the memory Limit for that container.

Kubelet kills a container when its resource usage exceeds its resource requests. A Pod in the Guaranteed QoS may not have its containers killed unless the containers exceed their resource limit.

CPU is a compressible resource. That is to say, the Cgroup for each container can stop a container from using more than the requested limit. Since the CPU request and limit are equal, Kubelet may not select containers to be killed because of CPU.

Memory is not a compressible resource. Kubelet periodically checks memory usage of a container. If the memory use of each container stays within the limit, containers may not be killed due to memory.

Kubelet does not kill Containers of Pods inside the Guaranteed QoS unless they exceed their memory (or other incompressible resources) limit.[5]

[5] Kubelet may evict Pods in the Guaranteed QoS if the node has DiskPressure

Figure 4: Guaranteed QoS Example

```
$ cat guaranteed-qos.yaml
apiVersion: v1
kind: Pod
metadata:
  name: guaranteed-qos
  namespace: test-qos
spec:
  containers:
  - name: qos-nginx
    image: nginx
    resources:
      limits:
        memory: "100Mi"
        cpu: "800m"
      requests:
        memory: "100Mi"
        cpu: "800m"
$ kubectl create ns test-qos
namespace/test-qos created
$ kubectl apply -f guaranteed-qos.yaml
pod/guaranteed-qos created
$ kubectl get pod -n test-qos guaranteed-qos -o yaml | grep qosClass
  qosClass: Guaranteed
```

4.2 Burstable QoS

To be considered in the Burstable QoS, a pod must meet the following criteria:

- Pod does not meet the criteria for Guaranteed QoS.
- Pod has one or more resource limit or request set.

Kubelet may kill containers in Pods in the Burstable QoS when they exceed their resource requests. Kubelet kills containers in the Burstable QoS are only after containers in Pods in the Best Effort QoS.

Figure 5: Burstable QoS Example

```
$ cat burstable-qos.yaml
apiVersion: v1
kind: Pod
metadata:
  name: burstable-qos
  namespace: test-qos
spec:
  containers:
  - name: qos-nginx
    image: nginx
    resources:
      requests:
        memory: "100Mi"
        cpu: "800m"
$ kubectl create ns test-qos
namespace/test-qos created
$ kubectl apply -f burstable-qos.yaml
pod/burstable-qos created
$ kubectl get pod -n test-qos burstable-qos -o yaml | grep qosClass
  qosClass: Burstable
```

4.3 Best Effort QoS

Pods are placed in the Best Effort QoS when none of their containers express a resource request or limit.

Kubelet may kill containers in Pods in the Best Effort QoS when there is any contention for any resource. Kubelet kills containers in the Best Effort QoS before any containers in the other QoS classes.

```
$ cat best-effort-qos.yaml
apiVersion: v1
kind: Pod
metadata:
  name: best-effort-qos
  namespace: test-qos
spec:
  containers:
  - name: qos-nginx
    image: nginx
$ kubectl create ns test-qos
namespace/test-qos created
$ kubectl apply -f best-effort-qos.yaml
pod/best-effort-qos created
$ kubectl get pod -n test-qos best-effort-qos -o yaml | grep qosClass
  qosClass: BestEffort
```

Figure 6: Best Effort QoS Example

5 *Eviction Handling*

In order to preserve the stability of nodes, the Kubelet must evict pods when critical compute resources become low.[6] Kubernetes honors several eviction signals when making this decision.

[6] Configure out of resource handling. https://kubernetes.io/docs/tasks/administer-cluster/out-of-resource/. Accessed: 2019-10-20

Eviction Signal	Description	Default Value
memory.available	Difference between node capacity and current working set	100Mi
nodefs.available	Disk space free on filesystems used for logs and ephemeral volumes	<10%
nodefs.inodesFree	Inodes free on filesystems used for logs and ephemeral volumes	<5%
imagefs.available	Disk space free on filesystems used for container images	<15%
imagefs.inodes.Free	Inodes free on filesystems used for container images	<5%

6 *Soft Versus Hard Eviction Thresholds*

Kubelet has automatic built-in hard eviction thresholds. Also, operators may optionally configure it with soft eviction thresholds and a grace period. When soft thresholds are reached, Kubelet waits for a grace period before taking corrective action. If usage reaches hard thresholds, Kubelet takes corrective action immediately.

When Kubelet finally decides it needs to reclaim node resources, it takes several steps based on the type of signal that triggers the eviction. If the resource is one of the disk-based signals, it first attempts to free up space consumed by dead containers. If the signal has not been satisfied, then unused container images are deleted.

If Kubelet is not able to satisfy the eviction signal in any other way, it ranks Pods for eviction.

1. Pods in the Burstable and BestEffort QoS whose usage exceed requests are ranked in order of Priority[7] and then usage above request.
2. After evicting all pods whose usage exceeds requests, Pods in the Burstable QoS are evicted in order of Priority.
3. Pods in the Guaranteed QoS are evicted last, in order of Priority.

[7] Pod priority and preemption. `https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/`. Accessed: 2019-10-20

Since Pods in the Guaranteed QoS have limits and requests that are equal, Kubelet only evicts them when system daemons exceed the resource amounts reserved for them by Kubelet.

References

Authenticating. `https://kubernetes.io/docs/reference/access-authn-authz/authentication/`. Accessed: 2019-10-16.

Container lifecycle hooks. `https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/`. Accessed: 2019-10-15.

Configure out of resource handling. `https://kubernetes.io/docs/tasks/administer-cluster/out-of-resource/`. Accessed: 2019-10-20.

Managing service accounts. `https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/`. Accessed: 2019-10-18.

Pod priority and preemption. `https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/`. Accessed: 2019-10-20.

Configure quality of service for pods. `https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/`. Accessed: 2019-10-19.