HOW TO USE & MANAGE KUBERNETES DAEMONSETS



Kubernetes is a leading open-source engine that orchestrates containerized applications.

In this article, we will have a look at the DaemonSet feature offered by Kubernetes. We'll walk you through use cases and how to create, update, communicate with, and delete DaemonSets.

(This article is part of our Kubernetes Guide. Use the right-hand menu to navigate.)

What is a Kubernetes DaemonSet?

The DaemonSet feature is used to ensure that some or all of your pods are scheduled and running on every single available node. This essentially runs a copy of the desired pod across all nodes.

- When a new node is added to a Kubernetes cluster, a new pod will be added to that newly attached node.
- When a node is removed, the DaemonSet controller ensures that the pod associated with that node is garbage collected. Deleting a DaemonSet will clean up all the pods that DaemonSet has created.

DaemonSets are an integral part of the Kubernetes cluster facilitating administrators to easily configure services (pods) across all or a subset of nodes.

DaemonSet use cases

DaemonSets can improve the performance of a Kubernetes cluster by distributing maintenance tasks and support services via deploying Pods across all nodes. They are well suited for long-

running services like monitoring or log collection. Following are some example use cases of DaemonSets:

- To run a daemon for cluster storage on each node, such as glusterd and ceph.
- To run a daemon for logs collection on each node, such as Fluentd and logstash.
- To run a daemon for node monitoring on every note, such as Prometheus Node Exporter, collectd, or Datadog agent.

Depending on the requirement, you can set up multiple DaemonSets for a single type of daemon, with different flags, memory, CPU, etc. that supports multiple configurations and hardware types.

Scheduling DaemonSet pods

By default, the node that a pod runs on is decided by the Kubernetes scheduler. However, DaemonSet pods are created and scheduled by the DaemonSet controller. Using the DaemonSet controller can lead to Inconsistent Pod behavior and issues in Pod priority preemption.

To mitigate these issues, Kubernetes (ScheduleDaemonSetPods) allows users to schedule DaemonSets using the default scheduler instead of the DaemonSet controller. This is done by adding the NodeAffinity term to the DaemonSet pods instead of the .spec.nodeName term. The default scheduler is then used to bind the Pod to the target host.

The following is a sample NodeAffinity configuration:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          # Key Name
          - key: disktype
            operator: In
            # Value
            values:
            - ssd
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
```

Above, we configured NodeAffinity so that a pod will only be created on a node that has the "disktype=ssd" label.

Additionally, DaemonSet pods adhere to taints and tolerations in Kubernetes. The node.kubernetes.io/unschedulable:NoSchedule toleration is automatically added to DaemonSet

pods. (For more information about taints and tolerations, please refer to the official Kubernetes documentation.)

How to create a DaemonSet

As for every other component in Kubernetes, DaemonSets are configured using a YAML file. Let's have a look at the structure of a DaemonSet file.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: test-daemonset
  namespace: test-daemonset-namespace
  Labels:
    app-type: test-app-type
spec:
  template:
    metadata:
      labels:
        name: test-daemonset-container
  selector:
    matchLabels:
      name: test-daemonset-container
```

As you can notice in the above structure, the **apiVersion**, **kind,** and **metadata** are required fields in every Kubernetes manifest. The DaemonSet specific fields come under the spec section—these fields are both mandatory.

- **template.** This is the pod definition for the Pod that needs to be deployed across all the nodes. A pod template in a DaemonSet must have its RestartPolicy set to "Always," and by default it will take "Always" if you havne't specified a RestartPolicy.
- **selector.** The selector for the pods managed by the DaemonSet. This value must be a label specified in the pod template. (In the above example, we have used the name: test-daemonset-container as the selector.) This value is fixed and cannot be changed after the initial creation of the DaemonSet. Changing this value will cause pods created via that DaemonSet to be orphaned. Kubernetes offers two ways to match matchLabels and matchExpressions for creating complex selectors.

Other optional fields

- **template.spec.nodeSelector** - This can be used to specify a subset of nodes that will create the Pod matching the specified selector.
- **template.spec.affinity** - This field can be configured to set the affinity that would run the pod only on nodes that match the configured affinity.

Creating a DaemonSet

Now let's go ahead with creating a sample DaemonSet. Here, we will be using a “fluentd-

elasticsearch" image that will run on every node in a Kubernetes cluster. Each pod would then collect logs and send the data to ElasticSearch.

daemonset-example.yaml

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch-test
  namespace: default # Name Space
  labels:
    k8s-app: fluentd-logging
spec:
  selector: # Selector
    matchLabels:
      name: fluentd-elasticsearch-test-deamonset
  template: # Pod Template
    metadata:
      labels:
        name: fluentd-elasticsearch-test-deamonset
    spec:
      tolerations: # Tolerations
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers: # Container Details
      - name: fluentd-elasticsearch
        image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

First, let's create the DaemonSet using the kubectl create command and retrieve the DaemonSet and pod information as follows:

`kubectl create -f daemonset-example.yaml`

```
> kubectl create -f daemonset-example.yaml
daemonset.apps/fluentd-elasticsearch-test created
```

`kubectl get daemonset`

```
> kubectl get daemonset
NAME                          DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
fluentd-elasticsearch-test    1         1         1       1            1           <none>          25s
```

`kubectl get pod -o wide`

```
> kubectl get pod -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP           NODE
fluentd-elasticsearch-test-kn7g8    1/1     Running   0          18s   10.1.0.148   docker-desktop
```

As you can see from the above output, our DaemonSet has been successfully deployed.

Depending on the nodes available on the cluster, it will scale automatically to match the number of nodes or a subset of nodes on the configuration. (Here, the number of nodes will be one as we are running this on a test environment with a single node Kubernetes cluster.)

Updating DaemonSets

When it comes to updating DaemonSets, if a node label is changed, DaemonSet will automatically add new pods to matching nodes while deleting pods from non-matching nodes. We can use the "kubectl apply" command to update a DaemonSet, as shown below.

`kubectl apply -f <<DaemonSet>>`

There are two strategies that can be followed when updating DaemonSets:

- The default strategy in Kubernetes, this will delete old DaemonSet pods and automatically create new pods when a DaemonSet template is updated.
- When using this option, new DaemonSet pods are created only after a user manually deletes old DaemonSet pods.

These strategies can be configured using the spec.updateStrategy.type option.

Deleting DaemonSets

Deleting a DaemonSet is a simple task. To do that, simply run the kubectl delete command with the DaemonSet. This would delete the DaemonSet with all the underlying pods it has created.

`kubectl delete <<DaemonSet>>`

We can use the cascade=false flag in the kubectl delete command to only delete the DaemonSet without deleting the pods.

Communicating with pods created by DaemonSet

There are multiple methods to communicate with pods created by DaemonSets. Here are some available options:

- **Push.** This way, pods can be configured to send information to other services (monitoring service, stats database). However, they do not receive any data.
- **NodeIP & Known Port.** Pods are reachable via the node IPs using hostPort. Users can then utilize the known ports and the node IP to communicate with the pods.
- **DNS.** In this method, users can configure a headless service with the same pod selector to discover DaemonSet using the endpoints resource.
- **Service.** To select a random node in a DaemonSet, which we can use to create a service with the same pod selector.

DaemonSet summary

In this article, we learned about Kubernetes DaemonSets. These configurations can easily facilitate monitoring, storage, or logging services that can be used to increase the performance and reliability of both the Kubernetes cluster and the containers.

Related reading

- BMC DevOps Blog
- Kubernetes Guide, with 20+ articles and tutorials
- Kubernetes Best Practices for Enhanced Cluster Efficiency
- Kubernetes vs Docker Swarm: Comparing Container Orchestration Tools
- Container Management Platforms: Which Are Most Popular?