DaemonSet

A *DaemonSet* ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

Some typical uses of a DaemonSet are:

- running a cluster storage daemon, such as `glusterd`, `ceph`, on each node.
- running a logs collection daemon on every node, such as `fluentd` or `filebeat`.
- running a node monitoring daemon on every node, such as Prometheus Node Exporter, Flowmill, Sysdig Agent, `collectd`, Dynatrace OneAgent, AppDynamics Agent, Datadog agent, New Relic agent, Ganglia `gmond`, Instana Agent or Elastic Metricbeat.

In a simple case, one DaemonSet, covering all nodes, would be used for each type of daemon. A more complex setup might use multiple DaemonSets for a single type of daemon, but with different flags and/or different memory and cpu requests for different hardware types.

Writing a DaemonSet Spec

Create a DaemonSet

You can describe a DaemonSet in a YAML file. For example, the `daemonset.yaml` file below describes a DaemonSet that runs the fluentd-elasticsearch Docker image:

controllers/daemonset.yaml

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
      # this toleration is to have the daemonset runnable on master nodes
      # remove it if your masters can't run pods
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - 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
```

Create a DaemonSet based on the YAML file:

```
kubectl apply -f https://k8s.io/examples/controllers/daemonset.yaml
```

Required Fields

As with all other Kubernetes config, a DaemonSet needs `apiVersion`, `kind`, and `metadata` fields. For general information about working with config files, see deploying applications, configuring containers, and object management using kubectl documents.

The name of a DaemonSet object must be a valid DNS subdomain name.

A DaemonSet also needs a `.spec` section.

Pod Template

The `.spec.template` is one of the required fields in `.spec`.

The `.spec.template` is a pod template. It has exactly the same schema as a Pod, except it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields for a Pod, a Pod template in a DaemonSet has to specify appropriate labels (see pod selector).

A Pod Template in a DaemonSet must have a `RestartPolicy` equal to `Always`, or be unspecified, which defaults to `Always`.

Pod Selector

The `.spec.selector` field is a pod selector. It works the same as the `.spec.selector` of a Job.

As of Kubernetes 1.8, you must specify a pod selector that matches the labels of the `.spec.template`. The pod selector will no longer be defaulted when left empty. Selector defaulting was not compatible with `kubectl apply`. Also, once a DaemonSet is created, its `.spec.selector` can not be mutated. Mutating the pod selector can lead to the unintentional orphaning of Pods, and it was found to be confusing to users.

The `.spec.selector` is an object consisting of two fields:

- `matchLabels` - works the same as the `.spec.selector` of a ReplicationController.
- `matchExpressions` - allows to build more sophisticated selectors by specifying key, list of values and an operator that relates the key and values.

When the two are specified the result is ANDed.

If the `.spec.selector` is specified, it must match the `.spec.template.metadata.labels`. Config with these not matching will be rejected by the API.

Also you should not normally create any Pods whose labels match this selector, either directly, via another DaemonSet, or via another workload resource such [illegible]plicaSet. Otherwise, the DaemonSet Controller will think that those Pods were created by it. Kubernetes will not stop you from doing this. One case where you might want to do this is manually create a Pod with a different value on a node for testing.

Running Pods on Only Some Nodes

If you specify a `.spec.template.spec.nodeSelector`, then the DaemonSet controller will create Pods on nodes which match that node selector. Likewise if you specify a `.spec.template.spec.affinity`, then DaemonSet controller will create Pods on nodes which match that node affinity. If you do not specify either, then the DaemonSet controller will create Pods on all nodes.

How Daemon Pods are Scheduled

Scheduled by default scheduler

FEATURE STATE: `Kubernetes v1.18` stable

A DaemonSet ensures that all eligible nodes run a copy of a Pod. Normally, the node that a Pod runs on is selected by the Kubernetes scheduler. However, DaemonSet pods are created and scheduled by the DaemonSet controller instead. That introduces the following issues:

- Inconsistent Pod behavior: Normal Pods waiting to be scheduled are created and in `Pending` state, but DaemonSet pods are not created in `Pending` state. This is confusing to the user.
- Pod preemption is handled by default scheduler. When preemption is enabled, the DaemonSet controller will make scheduling decisions without considering pod priority and preemption.

`ScheduleDaemonSetPods` allows you to schedule DaemonSets using the default scheduler instead of the DaemonSet controller, 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. If node affinity of the DaemonSet pod already exists, it is replaced. The DaemonSet controller only performs these operations when creating or modifying DaemonSet pods, and no changes are made to the `spec.template` of the DaemonSet.

```
nodeAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
    nodeSelectorTerms:
    - matchFields:
      - key: metadata.name
        operator: In
        values:
        - target-host-name
```

In addition, `node.kubernetes.io/unschedulable:NoSchedule` toleration is added automatically to DaemonSet Pods. The default scheduler ignores `unschedulable` Nodes when scheduling DaemonSet Pods.

Taints and Tolerations

Although Daemon Pods respect taints and tolerations, the following tolerations are added to DaemonSet Pods automatically according to the related features.

Toleration Key	Effect	Version	Description
`node.kubernetes.io/not-ready`	NoExecute	1.13+	DaemonSet pods will not be evicted when there are node problems such as a network partition.
`node.kubernetes.io/unreachable`	NoExecute	1.13+	DaemonSet pods will not be evicted when there are node problems such as a network partition.
`node.kubernetes.io/disk-pressure`	NoSchedule	1.8+	
`node.kubernetes.io/memory-pressure`	NoSchedule	1.8+	
`node.kubernetes.io/unschedulable`	NoSchedule	1.12+	DaemonSet pods tolerate unschedulable attributes by default scheduler.
`node.kubernetes.io/network-unavailable`	NoSchedule	1.12+	DaemonSet pods, who uses host network, tolerate network-unavailable attributes by default scheduler.

Communicating with Daemon Pods

Some possible patterns for communicating with Pods in a DaemonSet are:

- **Push**: Pods in the DaemonSet are configured to send updates to another service, such as a stats database. They do not have clients.
- **NodeIP and Known Port**: Pods in the DaemonSet can use a `hostPort`, so that the pods are reachable via the node IPs. Clients know the list of node IPs somehow, and know the port by convention.
- **DNS**: Create a headless service with the same pod selector, and then discover DaemonSets using the `endpoints` resource or retrieve multiple A records from DNS.
- **Service**: Create a service with the same Pod selector, and use the service to reach a daemon on a random node. (No way to reach specific node.)

Updating a DaemonSet

If node labels are changed, the DaemonSet will promptly add Pods to newly matching nodes and delete Pods from newly not-matching nodes.

You can modify the Pods that a DaemonSet creates. However, Pods do not allow all fields to be updated. Also, the DaemonSet controller will use the original template the next time a node (even with the same name) is created.

You can delete a DaemonSet. If you specify `--cascade=false` with `kubectl`, then the Pods will be left on the nodes. If you subsequently create a new DaemonSet with the same selector, the new DaemonSet adopts the existing Pods. If any Pods need replacing the DaemonSet replaces them according to its `updateStrategy`.

You can perform a rolling update on a DaemonSet.

Alternatives to DaemonSet

Init Scripts

It is certainly possible to run daemon processes by directly starting them on a node (e.g. using `init`, `upstartd`, or `systemd`). This is perfectly fine. However, there are several advantages to running such processes via a DaemonSet:

- Ability to monitor and manage logs for daemons in the same way as applications.
- Same config language and tools (e.g. Pod templates, `kubectl`) for daemons and applications.
- Running daemons in containers with resource limits increases isolation between daemons from app containers. However, this can also be accomplished by running the daemons in a container but not in a Pod (e.g. start directly via Docker).

Bare Pods

It is possible to create Pods directly which specify a particular node to run on. However, a DaemonSet replaces Pods that are deleted or terminated for any reason, such as in the case of node failure or disruptive node maintenance, such as a kernel upgrade. For this reason, you should use a DaemonSet rather than creating individual Pods.

Static Pods

It is possible to create Pods by writing a file to a certain directory watched by Kubelet. These are called static pods. Unlike DaemonSet, static Pods cannot be managed with kubectl or other Kubernetes API clients. Static Pods do not depend on the apiserver, making them useful in cluster bootstrapping cases. Also, static Pods may be deprecated in the future.