# DaemonSet

Archived: 2026-04-06 01:10:58 UTC

A DaemonSet defines Pods that provide node-local facilities. These might be fundamental to the operation of your cluster, such as a networking helper tool, or be part of an add-on.

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 on every node
- running a logs collection daemon on every node
- running a node monitoring daemon on every node

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:

```
 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:
      # these tolerations are to have the daemonset runnable on control plane nodes
      # remove them if your control plane nodes should not run pods
      - key: node-role.kubernetes.io/control-plane
        operator: Exists
        effect: NoSchedule
      - key: node-role.kubernetes.io/master
        operator: Exists
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image: quay.io/fluentd_elasticsearch/fluentd:v5.0.1
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: varlog
          mountPath: /var/log
      # it may be desirable to set a high priority class to ensure that a DaemonSet Pod
      # preempts running Pods
      # priorityClassName: important
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
```

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 running stateless applications and object management using kubectl.

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.

You must specify a pod selector that matches the labels of the `.spec.template`. 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.

The `.spec.selector` must match the `.spec.template.metadata.labels`. Config with these two not matching will be rejected by the API.

### Running Pods on select 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

A DaemonSet can be used to ensure that all eligible nodes run a copy of a Pod. The DaemonSet controller creates a Pod for each eligible node and adds the `spec.affinity.nodeAffinity` field of the Pod to match the target host. After the Pod is created, the default scheduler typically takes over and then binds the Pod to the target host by setting the `.spec.nodeName` field. If the new Pod cannot fit on the node, the default scheduler may preempt (evict) some of the existing Pods based on the priority of the new Pod.

**Note:**

If it's important that the DaemonSet pod run on each node, it's often desirable to set the `.spec.template.spec.priorityClassName` of the DaemonSet to a PriorityClass with a higher priority to ensure that this eviction occurs.

The user can specify a different scheduler for the Pods of the DaemonSet, by setting the `.spec.template.spec.schedulerName` field of the DaemonSet.

The original node affinity specified at the `.spec.template.spec.affinity.nodeAffinity` field (if specified) is taken into consideration by the DaemonSet controller when evaluating the eligible nodes, but is replaced on the created Pod with the node affinity that matches the name of the eligible node.

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

### Taints and tolerations

The DaemonSet controller automatically adds a set of tolerations to DaemonSet Pods:

| Toleration key | Effect | Details |
| --- | --- | --- |
| `node.kubernetes.io/not-ready` | `NoExecute` | DaemonSet Pods can be scheduled onto nodes that are not healthy or ready to accept Pods. Any DaemonSet Pods running on such nodes will not be evicted. |
| `node.kubernetes.io/unreachable` | `NoExecute` | DaemonSet Pods can be scheduled onto nodes that are unreachable from the node controller. Any DaemonSet Pods running on such nodes will not be evicted. |
| `node.kubernetes.io/disk-pressure` | `NoSchedule` | DaemonSet Pods can be scheduled onto nodes with disk pressure issues. |
| `node.kubernetes.io/memory-pressure` | `NoSchedule` | DaemonSet Pods can be scheduled onto nodes with memory pressure issues. |
| `node.kubernetes.io/pid-pressure` | `NoSchedule` | DaemonSet Pods can be scheduled onto nodes with process pressure issues. |

| Toleration key | Effect | Details |
| --- | --- | --- |
| `node.kubernetes.io/unschedulable` | `NoSchedule` | DaemonSet Pods can be scheduled onto nodes that are unschedulable. |
| `node.kubernetes.io/network-unavailable` | `NoSchedule` | **Only added for DaemonSet Pods that request host networking**, i.e., Pods having `spec.hostNetwork: true`. Such DaemonSet Pods can be scheduled onto nodes with unavailable network. |

You can add your own tolerations to the Pods of a DaemonSet as well, by defining these in the Pod template of the DaemonSet.

Because the DaemonSet controller sets the `node.kubernetes.io/unschedulable:NoSchedule` toleration automatically, Kubernetes can run DaemonSet Pods on nodes that are marked as *unschedulable*.

If you use a DaemonSet to provide an important node-level function, such as cluster networking, it is helpful that Kubernetes places DaemonSet Pods on nodes before they are ready. For example, without that special toleration, you could end up in a deadlock situation where the node is not marked as ready because the network plugin is not running there, and at the same time the network plugin is not running on that node because the node is not yet ready.

## 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. Use Service Internal Traffic Policy to limit to pods on the same 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=orphan` 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.

### 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.

### Deployments

DaemonSets are similar to Deployments in that they both create Pods, and those Pods have processes which are not expected to terminate (e.g. web servers, storage servers).

Use a Deployment for stateless services, like frontends, where scaling up and down the number of replicas and rolling out updates are more important than controlling exactly which host the Pod runs on. Use a DaemonSet when it is important that a copy of a Pod always run on all or certain hosts, if the DaemonSet provides node-level functionality that allows other Pods to run correctly on that particular node.

For example, network plugins often include a component that runs as a DaemonSet. The DaemonSet component makes sure that the node where it's running has working cluster networking.

## What's next

- Learn about Pods:
  - Learn about static Pods, which are useful for running Kubernetes control plane components.
- Find out how to use DaemonSets:
  - Perform a rolling update on a DaemonSet.
  - Perform a rollback on a DaemonSet (for example, if a roll out didn't work how you expected).
- Understand how Kubernetes assigns Pods to Nodes.
- Learn about device plugins and add ons, which often run as DaemonSets.
- `DaemonSet` is a top-level resource in the Kubernetes REST API. Read the DaemonSet object definition to understand the API for daemon sets.

---