Chapter 9. DaemonSets

ReplicaSets are generally about creating a service (e.g., a web server) with multiple replicas for redundancy. But that is not the only reason you may want to replicate a set of Pods within a cluster. Another reason to replicate a set of Pods is to schedule a single Pod on every node within the cluster. Generally, the motivation for replicating a Pod to every node is to land some sort of agent or daemon on each node, and the Kubernetes object for achieving this is the DaemonSet.

A DaemonSet ensures a copy of a Pod is running across a set of nodes in a Kubernetes cluster. DaemonSets are used to deploy system daemons such as log collectors and monitoring agents, which typically must run on every node. DaemonSets share similar functionality with ReplicaSets; both create Pods that are expected to be long-running services and ensure that the desired state and the observed state of the cluster match.

Given the similarities between DaemonSets and ReplicaSets, it’s important to understand when to use one over the other. ReplicaSets should be used when your application is completely decoupled from the node and you can run multiple copies on a given node without special consideration. DaemonSets should be used when a single copy of your application must run on all or a subset of the nodes in the cluster.

You should generally not use scheduling restrictions or other parameters to ensure that Pods do not colocate on the same node. If you find yourself wanting a single Pod per node, then a DaemonSet is the correct Kubernetes resource to use. Likewise, if you find yourself building a homogeneous replicated service to serve user traffic, then a ReplicaSet is probably the right Kubernetes resource to use.

DaemonSet Scheduler

By default a DaemonSet will create a copy of a Pod on every node unless a node selector is used, which will limit eligible nodes to those with a matching set of labels. DaemonSets determine which node a Pod will run on at Pod creation time by specifying the `nodeName` field in the Pod spec. As a result, Pods created by DaemonSets are ignored by the Kubernetes scheduler.

Like ReplicaSets, DaemonSets are managed by a reconciliation control loop that measures the desired state (a Pod is present on all nodes) with the observed state (is the Pod present on a particular node?). Given this information, the DaemonSet controller creates a Pod on each node that doesn't currently have a matching Pod.

If a new node is added to the cluster, then the DaemonSet controller notices that it is missing a Pod and adds the Pod to the new node.

> **NOTE**
>
> DaemonSets and ReplicaSets are a great demonstration of the value of Kubernetes's decoupled architecture. It might seem that the right design would be for a ReplicaSet to own the Pods it manages, and for Pods to be subresources of a ReplicaSet. Likewise, the Pods managed by a DaemonSet would be subresources of that DaemonSet. However, this kind of encapsulation would require that tools for dealing with Pods be written two different times, one for DaemonSets and one for ReplicaSets. Instead, Kubernetes uses a decoupled approach where Pods are top-level objects. This means that every tool you have learned for introspecting Pods in the context of ReplicaSets (e.g., `kubectl logs <pod-name>`) is equally applicable to Pods created by DaemonSets.

Creating DaemonSets

DaemonSets are created by submitting a DaemonSet configuration to the Kubernetes API server. The following DaemonSet will create a `fluentd` logging agent on every node in the target cluster (Example 9-1).

Example 9-1. fluentd.yaml

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: fluentd
  namespace: kube-system
  labels:
    app: fluentd
spec:
  template:
    metadata:
      labels:
        app: fluentd
    spec:
      containers:
      - name: fluentd
        image: fluent/fluentd:v0.14.10
        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
```

DaemonSets require a unique name across all DaemonSets in a given Kubernetes namespace. Each DaemonSet must include a Pod template spec, which will be used to create Pods as needed. This is where the similarities between ReplicaSets and DaemonSets end. Unlike ReplicaSets, DaemonSets will create Pods on every node in the cluster by default unless a node selector is used.

Once you have a valid DaemonSet configuration in place, you can use the

`kubectl apply` command to submit the DaemonSet to the Kubernetes API. In this section we will create a DaemonSet to ensure the fluentd HTTP server is running on every node in our cluster:

```
$ kubectl apply -f fluentd.yaml
daemonset "fluentd" created
```

Once the fluentd DaemonSet has been successfully submitted to the Kubernetes API, you can query its current state using the `kubectl describe` command:

```
$ kubectl describe daemonset fluentd
Name:           fluentd
Image(s):       fluent/fluentd:v0.14.10
Selector:       app=fluentd
Node-Selector:  <none>
Labels:         app=fluentd
Desired Number of Nodes Scheduled: 3
Current Number of Nodes Scheduled: 3
Number of Nodes Misscheduled: 0
Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed
```

This output indicates a `fluentd` Pod was successfully deployed to all three nodes in our cluster. We can verify this using the `kubectl get pods` command with the `-o` flag to print the nodes where each fluentd Pod was assigned:

```
$ kubectl get pods -o wide

NAME            AGE    NODE
fluentd-1q6c6   13m    k0-default-pool-35609c18-z7tb
fluentd-mwi7h   13m    k0-default-pool-35609c18-ydae
fluentd-zr6l7   13m    k0-default-pool-35609c18-pol3
```

With the `fluentd` DaemonSet in place, adding a new node to the cluster will result in a `fluentd` Pod being deployed to that node automatically:

```
$ kubectl get pods -o wide
NAME            AGE    NODE
fluentd-1q6c6   13m    k0-default-pool-35609c18-z7tb
fluentd-mwi7h   13m    k0-default-pool-35609c18-ydae
fluentd-oipmq   43s    k0-default-pool-35609c18-0xnl
fluentd-zr6l7   13m    k0-default-pool-35609c18-pol3
```

This is exactly the behavior you want when managing logging daemons and other cluster-wide services. No action was required from our end; this is how the Kubernetes DaemonSet controller reconciles its observed state with our desired

state.

Limiting DaemonSets to Specific Nodes

The most common use case for DaemonSets is to run a Pod across every node in a Kubernetes cluster. However, there are some cases where you want to deploy a Pod to only a subset of nodes. For example, maybe you have a workload that requires a GPU or access to fast storage only available on a subset of nodes in your cluster. In cases like these node labels can be used to tag specific nodes that meet workload requirements.

Adding Labels to Nodes

The first step in limiting DaemonSets to specific nodes is to add the desired set of labels to a subset of nodes. This can be achieved using the `kubectl label` command.

The following command adds the `ssd=true` label to a single node:

```
$ kubectl label nodes k0-default-pool-35609c18-z7tb ssd=true
node "k0-default-pool-35609c18-z7tb" labeled
```

Just like with other Kubernetes resources, listing nodes without a label selector returns all nodes in the cluster:

```
$ kubectl get nodes
NAME                           STATUS    AGE
k0-default-pool-35609c18-0xnl   Ready     23m
k0-default-pool-35609c18-pol3   Ready     1d
k0-default-pool-35609c18-ydae   Ready     1d
k0-default-pool-35609c18-z7tb   Ready     1d
```

Using a label selector we can filter nodes based on labels. To list only the nodes that have the `ssd` label set to `true`, use the `kubectl get nodes` command with the `--selector` flag:

```
$ kubectl get nodes --selector ssd=true
NAME                           STATUS    AGE
k0-default-pool-35609c18-z7tb   Ready     1d
```

Node Selectors

Node selectors can be used to limit what nodes a Pod can run on in a given Kubernetes cluster. Node selectors are defined as part of the Pod spec when creating a DaemonSet. The following DaemonSet configuration limits nginx to running only on nodes with the `ssd=true` label set (Example 9-2).

Example 9-2. nginx-fast-storage.yaml

```
apiVersion: extensions/v1beta1
kind: "DaemonSet"
metadata:
  labels:
    app: nginx
    ssd: "true"
  name: nginx-fast-storage
spec:
  template:
    metadata:
      labels:
        app: nginx
        ssd: "true"
    spec:
      nodeSelector:
        ssd: "true"
      containers:
        - name: nginx
          image: nginx:1.10.0
```

Let's see what happens when we submit the `nginx-fast-storage` DaemonSet to the Kubernetes API:

```
$ kubectl apply -f nginx-fast-storage.yaml
daemonset "nginx-fast-storage" created
```

Since there is only one node with the `ssd=true` label, the `nginx-fast-storage` Pod will only run on that node:

```
$ kubectl get pods -o wide
NAME                       STATUS    NODE
nginx-fast-storage-7b90t   Running   k0-default-pool-35609c18-z7tb
```

Adding the `ssd=true` label to additional nodes will case the `nginx-fast-storage` Pod to be deployed on those nodes. The inverse is also true: if a required label is removed from a node, the Pod will be removed by the DaemonSet controller.

WARNING

Removing labels from a node that are required by a DaemonSet's node selector will cause the Pod being managed by that DaemonSet to be removed from the node.

Updating a DaemonSet

DaemonSets are great for deploying services across an entire cluster, but what about upgrades? Prior to Kubernetes 1.6, the only way to update Pods managed by a DaemonSet was to update the DaemonSet and then manually delete each Pod that was managed by the DaemonSet so that it would be re-created with the new configuration. With the release of Kubernetes 1.6 DaemonSets gained an equivalent to the `Deployment` object that manages a DaemonSet rollout inside the cluster.

Updating a DaemonSet by Deleting Individual Pods

If you are running a pre-1.6 version of Kubernetes, you can perform a rolling delete of the Pods a `DaemonSet` manages using a `for` loop on your own machine to update one DaemonSet Pod every 60 seconds:

```
PODS=$(kubectl get pods -o jsonpath -template='{.items[*].metadata.name}'
for x in $PODS; do
  kubectl delete pods ${x}
  sleep 60
done
```

An alternative, easier approach is to just delete the entire DaemonSet and create a new DaemonSet with the updated configuration. However, this approach has a major drawback — downtime. When a DaemonSet is deleted all Pods managed by that DaemonSet will also be deleted. Depending on the size of your container images, recreating a DaemonSet may push you outside of your SLA thresholds, so it might be worth considering pulling updated container images across your cluster before updating a DaemonSet.

Rolling Update of a DaemonSet

With Kubernetes 1.6, DaemonSets can now be rolled out using the same rolling update strategy that deployments use. However, for reasons of backward compatability, the current default update strategy is the `delete` method described in the previous section. To set a DaemonSet to use the rolling update strategy, you need to configure the update strategy using the `spec.updateStrategy.type` field. That field should have the value `RollingUpdate`. When a DaemonSet has an update strategy of `RollingUpdate`, any change to the `spec.template` field (or subfields) in the DaemonSet will initiate a rolling update.

As with rolling updates of deployments (see Chapter 12), the rolling update strategy gradually updates members of a DaemonSet until all of the Pods are running the new configuration. There are two parameters that control the rolling update of a DaemonSet:

- `spec.minReadySeconds`, which determines how long a Pod must be "ready" before the rolling update proceeds to upgrade subsequent Pods
- `spec.updateStrategy.rollingUpdate.maxUnavailable`, which indicates how many Pods may be simultaneously updated by the rolling update

You will likely want to set `spec.minReadySeconds` to a reasonably long value, for example 30–60 seconds, to ensure that your Pod is truly healthy before the rollout proceeds.

The setting for `spec.updateStrategy.rollingUpdate.maxUnavailable` is more likely to be application-dependent. Setting it to `1` is a safe, general-purpose strategy, but it also takes a while to complete the rollout (number of nodes × `maxReadySeconds`). Increasing the maximum unavailability will make your rollout move faster, but increases the "blast radius" of a failed rollout. The characteristics of your application and cluster environment dictate the relative values of speed versus safety. A good approach might be to set `maxUnavailable` to `1` and only increase it if users or administrators complain about DaemonSet rollout speed.

Once a rolling update has started, you can use the `kubectl rollout` commands

to see the current status of a DaemonSet rollout.

For example, `kubectl rollout status daemonSets my-daemon-set` will show the current rollout status of a DaemonSet named `my-daemon-set`.

Deleting a DaemonSet

Deleting a DaemonSet is pretty straightforward using the `kubectl delete` command. Just be sure to supply the correct name of the DaemonSet you would like to delete:

```
$ kubectl delete -f fluentd.yaml
```

> **WARNING**
>
> Deleting a DaemonSet will also delete all the Pods being managed by that DaemonSet. Set the `--cascade` flag to `false` to ensure only the DaemonSet is deleted and not the Pods.

Summary

DaemonSets provide an easy-to-use abstraction for running a set of Pods on every node in a Kubernetes cluster, or if the case requires it, on a subset of nodes based on labels. The DaemonSet provides its own controller and scheduler to ensure key services like monitoring agents are always up and running on the right nodes in your cluster.

For some applications, you simply want to schedule a certain number of replicas; you don't really care where they run as long as they have sufficient resources and distribution to operate reliably. However, there is a different class of applications, like agents and monitoring applications, that need to be present on every machine in a cluster to function properly. These DaemonSets aren't really traditional serving applications, but rather add additional capabilities and features to the Kubernetes cluster itself. Because the DaemonSet is an active declarative object managed by a controller, it makes it easy to declare your intent that an agent run on every machine without explicitly placing it on every machine. This is especially useful in the context of an autoscaled Kubernetes cluster where nodes may constantly be coming and going without user intervention. In such cases, the DaemonSet automatically adds the proper agents to each node as it is added to the cluster by the autoscaler.