Taints and Tolerations

Node affinity is a property of Pods that *attracts* them to a set of nodes (either as a preference or a hard requirement). *Taints* are the opposite -- they allow a node to repel a set of pods.

Tolerations are applied to pods. Tolerations allow the scheduler to schedule pods with matching taints. Tolerations allow scheduling but don't guarantee scheduling: the scheduler also evaluates other parameters as part of its function.

Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints.

Concepts

You add a taint to a node using kubectl taint. For example,

```
kubectl taint nodes node1 key1=value1:NoSchedule
```

places a taint on node `node1`. The taint has key `key1`, value `value1`, and taint effect `NoSchedule`. This means that no pod will be able to schedule onto `node1` unless it has a matching toleration.

To remove the taint added by the command above, you can run:

```
kubectl taint nodes node1 key1=value1:NoSchedule-
```

You specify a toleration for a pod in the PodSpec. Both of the following tolerations "match" the taint created by the `kubectl taint` line above, and thus a pod with either toleration would be able to schedule onto `node1`:

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
```

```
tolerations:
- key: "key1"
  operator: "Exists"
  effect: "NoSchedule"
```

Here's an example of a pod that uses tolerations:

pods/pod-with-toleration.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  tolerations:
  - key: "example-key"
    operator: "Exists"
    effect: "NoSchedule"
```

The default value for `operator` is `Equal`.

A toleration "matches" a taint if the keys are the same and the effects are the same, and:

- the `operator` is `Exists` (in which case no `value` should be specified), or
- the `operator` is `Equal` and the values should be equal.

> **Note:**
> There are two special cases:
>
> An empty `key` with operator `Exists` matches all keys, values and effects which means this will tolerate everything.
>
> An empty `effect` matches all effects with key `key1`.

The above example used the `effect` of `NoSchedule`. Alternatively, you can use the `effect` of `PreferNoSchedule`.

The allowed values for the `effect` field are:

`NoExecute`

This affects pods that are already running on the node as follows:
- Pods that do not tolerate the taint are evicted immediately
- Pods that tolerate the taint without specifying `tolerationSeconds` in their toleration specification remain bound forever
- Pods that tolerate the taint with a specified `tolerationSeconds` remain bound for the specified amount of time. After that time elapses, the node lifecycle controller evicts the Pods from the node.

`NoSchedule`

No new Pods will be scheduled on the tainted node unless they have a matching toleration. Pods currently running on the node are **not** evicted.

`PreferNoSchedule`

`PreferNoSchedule` is a "preference" or "soft" version of `NoSchedule`. The control plane will *try* to avoid placing a Pod that does not tolerate the taint on the node, but it is not guaranteed.

You can put multiple taints on the same node and multiple tolerations on the same pod. The way Kubernetes processes multiple taints and tolerations is like a filter: start with all of a node's taints, then ignore the ones for which the pod has a matching toleration; the remaining un-ignored taints have the indicated effects on the pod. In particular,

- if there is at least one un-ignored taint with effect `NoSchedule` then Kubernetes will not schedule the pod onto that node
- if there is no un-ignored taint with effect `NoSchedule` but there is at least one un-ignored taint with effect `PreferNoSchedule` then Kubernetes will *try* to not schedule the pod onto the node
- if there is at least one un-ignored taint with effect `NoExecute` then the pod will be evicted from the node (if it is already running on the node), and will not be scheduled onto the node (if it is not yet running on the node).

For example, imagine you taint a node like this

```
kubectl taint nodes node1 key1=value1:NoSchedule
kubectl taint nodes node1 key1=value1:NoExecute
kubectl taint nodes node1 key2=value2:NoSchedule
```

And a pod has two tolerations:

```yaml
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
```

In this case, the pod will not be able to schedule onto the node, because there is no toleration matching the third taint. But it will be able to continue running if it is already running on the node when the taint is added, because the third taint is the only one of the three that is not tolerated by the pod.

Normally, if a taint with effect `NoExecute` is added to a node, then any pods that do not tolerate the taint will be evicted immediately, and pods that do tolerate the taint will never be evicted. However, a toleration with `NoExecute` effect can specify an optional `tolerationSeconds` field that dictates how long the pod will stay bound to the node after the taint is added. For example,

```yaml
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
  tolerationSeconds: 3600
```

means that if this pod is running and a matching taint is added to the node, then the pod will stay bound to the node for 3600 seconds, and then be evicted. If the taint is removed before that time, the pod will not be evicted.

Example Use Cases

Taints and tolerations are a flexible way to steer pods *away* from nodes or evict pods that shouldn't be running. A few of the use cases are

- **Dedicated Nodes**: If you want to dedicate a set of nodes for exclusive use by a particular set of users, you can add a taint to those nodes (say, `kubectl taint nodes nodename dedicated=groupName:NoSchedule`) and then add a corresponding toleration to their pods (this would be done most easily by writing a custom admission controller). The pods with the tolerations will then be allowed to use the tainted (dedicated) nodes as well as any other nodes in the cluster. If you want to dedicate the nodes to them *and* ensure they *only* use the dedicated nodes, then you should additionally add a label similar to the taint to the same set of nodes (e.g. `dedicated=groupName`), and the admission controller should additionally add a node affinity to require that the pods can only schedule onto nodes labeled with `dedicated=groupName`.

- **Nodes with Special Hardware**: In a cluster where a small subset of nodes have specialized hardware (for example GPUs), it is desirable to keep pods that don't need the specialized hardware off of those nodes, thus leaving room for later-arriving pods that do need the specialized hardware. This can be done by tainting the nodes that have the specialized hardware (e.g. `kubectl taint nodes nodename special=true:NoSchedule` or `kubectl taint nodes nodename special=true:PreferNoSchedule`) and adding a corresponding toleration to pods that use the special hardware. As in the dedicated nodes use case, it is probably easiest to apply the tolerations using a custom admission controller. For example, it is recommended to use Extended Resources to represent the special hardware, taint your special hardware nodes with the extended resource name and run the ExtendedResourceToleration admission controller. Now, because the nodes are tainted, no pods without the toleration will schedule on them. But when you submit a pod that requests the extended resource, the `ExtendedResourceToleration`

admission controller will automatically add the correct toleration to the pod and that pod will schedule on the special hardware nodes. This will make sure that these special hardware nodes are dedicated for pods requesting such hardware and you don't have to manually add tolerations to your pods.
- **Taint based Evictions**: A per-pod-configurable eviction behavior when there are node problems, which is described in the next section.

Taint based Evictions

FEATURE STATE: `Kubernetes v1.18 [stable]`

The node controller automatically taints a Node when certain conditions are true. The following taints are built in:

- `node.kubernetes.io/not-ready`: Node is not ready. This corresponds to the NodeCondition `Ready` being "`False`".
- `node.kubernetes.io/unreachable`: Node is unreachable from the node controller. This corresponds to the NodeCondition `Ready` being "`Unknown`".
- `node.kubernetes.io/memory-pressure`: Node has memory pressure.
- `node.kubernetes.io/disk-pressure`: Node has disk pressure.
- `node.kubernetes.io/pid-pressure`: Node has PID pressure.
- `node.kubernetes.io/network-unavailable`: Node's network is unavailable.
- `node.kubernetes.io/unschedulable`: Node is unschedulable.
- `node.cloudprovider.kubernetes.io/uninitialized`: When the kubelet is started with an "external" cloud provider, this taint is set on a node to mark it as unusable. After a controller from the cloud-controller-manager initializes this node, the kubelet removes this taint.

In case a node is to be drained, the node controller or the kubelet adds relevant taints with `NoExecute` effect. This effect is added by default for the `node.kubernetes.io/not-ready` and `node.kubernetes.io/unreachable` taints. If the fault condition returns to normal, the kubelet or node controller can remove the relevant taint(s).

In some cases when the node is unreachable, the API server is unable to communicate with the kubelet on the node. The decision to delete the pods cannot be communicated to the kubelet until communication with the API server is re-established. In the meantime, the pods that are scheduled for deletion may continue to run on the partitioned node.

Note: The control plane limits the rate of adding new taints to nodes. This rate limiting manages the number of evictions that are triggered when many nodes become unreachable at once (for example: if there is a network disruption).

You can specify `tolerationSeconds` for a Pod to define how long that Pod stays bound to a failing or unresponsive Node.

For example, you might want to keep an application with a lot of local state bound to node for a long time in the event of network partition, hoping that the partition will recover and thus the pod eviction can be avoided. The toleration you set for that Pod might look like:

```yaml
tolerations:
- key: "node.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 6000
```

Note:
Kubernetes automatically adds a toleration for `node.kubernetes.io/not-ready` and `node.kubernetes.io/unreachable` with `tolerationSeconds=300`, unless you, or a controller, set those tolerations explicitly.

These automatically-added tolerations mean that Pods remain bound to Nodes for 5 minutes after one of these problems is detected.

DaemonSet pods are created with `NoExecute` tolerations for the following taints with no `tolerationSeconds`:

- `node.kubernetes.io/unreachable`
- `node.kubernetes.io/not-ready`

This ensures that DaemonSet pods are never evicted due to these problems.

Taint Nodes by Condition

The control plane, using the node controller, automatically creates taints with a `NoSchedule` effect for node conditions.

The scheduler checks taints, not node conditions, when it makes scheduling decisions. This ensures that node conditions don't directly affect scheduling. For example, if the `DiskPressure` node condition is active, the control plane adds the `node.kubernetes.io/disk-pressure` taint and does not schedule new pods onto the affected node. If the `MemoryPressure` node condition is active, the control plane adds the `node.kubernetes.io/memory-pressure` taint.

You can ignore node conditions for newly created pods by adding the corresponding Pod tolerations. The control plane also adds the `node.kubernetes.io/memory-pressure` toleration on pods that have a QoS class other than `BestEffort`. This is because Kubernetes treats pods in the `Guaranteed` or `Burstable` QoS classes (even pods with no memory request set) as if they are able to cope with memory pressure, while new `BestEffort` pods are not scheduled onto the affected node.

The DaemonSet controller automatically adds the following `NoSchedule` tolerations to all daemons, to prevent DaemonSets from breaking.

- `node.kubernetes.io/memory-pressure`
- `node.kubernetes.io/disk-pressure`
- `node.kubernetes.io/pid-pressure` (1.14 or later)
- `node.kubernetes.io/unschedulable` (1.10 or later)
- `node.kubernetes.io/network-unavailable` (*host network only*)

Adding these tolerations ensures backward compatibility. You can also add arbitrary tolerations to DaemonSets.

What's next

- Read about Node-pressure Eviction and how you can configure it
- Read about Pod Priority

Feedback

Was this page helpful?

Yes No

Last modified February 19, 2024 at 1:21 PM PST: Fix spelling mistake in scheduling section (e839bf7aee)