**Chapter 16. Advanced scheduling**

**This chapter covers**

* Using node taints and pod tolerations to keep pods away from certain nodes
* Defining node affinity rules as an alternative to node selectors
* Co-locating pods using pod affinity
* Keeping pods away from each other using pod anti-affinity

Kubernetes allows you to affect where pods are scheduled. Initially, this was only done by specifying a node selector in the pod specification, but additional mechanisms were later added that expanded this functionality. They’re covered in this chapter.

**16.1. USING TAINTS AND TOLERATIONS TO REPEL PODS FROM CERTAIN NODES**

The first two features related to advanced scheduling that we’ll explore here are the node taints and pods’ tolerations of those taints. They’re used for restricting which pods can use a certain node. A pod can only be scheduled to a node if it tolerates the node’s taints.

This is somewhat different from using node selectors and node affinity, which you’ll learn about later in this chapter. Node selectors and node affinity rules make it possible to select which nodes a pod can or can’t be scheduled to by specifically adding that information to the pod, whereas taints allow rejecting deployment of pods to certain nodes by only adding taints to the node without having to modify existing pods. Pods that you want deployed on a tainted node need to opt in to use the node, whereas with node selectors, pods explicitly specify which node(s) they want to be deployed to.

**16.1.1. Introducing taints and tolerations**

The best path to learn about node taints is to see an existing taint. [Appendix B](https://learning.oreilly.com/library/view/kubernetes-in-action/9781617293726/kindle_split_032.html#app02) shows how to set up a multi-node cluster with the kubeadm tool. By default, the master node in such a cluster is tainted, so only Control Plane pods can be deployed on it.

**Displaying a node’s taints**

You can see the node’s taints using kubectl describe node, as shown in the following listing.

**Listing 16.1. Describing the master node in a cluster created with kubeadm**

**$ kubectl describe node master.k8s**

Name: master.k8s

Role:

Labels: beta.kubernetes.io/arch=amd64

beta.kubernetes.io/os=linux

kubernetes.io/hostname=master.k8s

node-role.kubernetes.io/master=

Annotations: node.alpha.kubernetes.io/ttl=0

volumes.kubernetes.io/controller-managed-attach-detach=true

Taints: node-role.kubernetes.io/master:NoSchedule ***1***

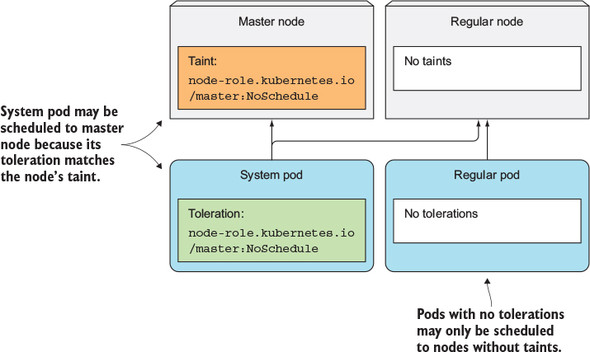
...

* ***1*** **The master node has one taint.**

The master node has a single taint. Taints have a *key*, *value*, and an *effect*, and are represented as <key>=<value>:<effect>. The master node’s taint shown in the previous listing has the key node-role.kubernetes.io/master, a null value (not shown in the taint), and the effect of NoSchedule.

This taint prevents pods from being scheduled to the master node, unless those pods tolerate this taint. The pods that tolerate it are usually system pods (see [figure 16.1](https://learning.oreilly.com/library/view/kubernetes-in-action/9781617293726/kindle_split_028.html#ch16fig01)).

**Figure 16.1. A pod is only scheduled to a node if it tolerates the node’s taints.**



**Displaying a pod’s tolerations**

In a cluster installed with kubeadm, the kube-proxy cluster component runs as a pod on every node, including the master node, because master components that run as pods may also need to access Kubernetes Services. To make sure the kube-proxy pod also runs on the master node, it includes the appropriate toleration. In total, the pod has three tolerations, which are shown in the following listing.

**Listing 16.2. A pod’s tolerations**

**$ kubectl describe po kube-proxy-80wqm -n kube-system**

...

Tolerations: node-role.kubernetes.io/master=:NoSchedule

node.alpha.kubernetes.io/notReady=:Exists:NoExecute

node.alpha.kubernetes.io/unreachable=:Exists:NoExecute

...

As you can see, the first toleration matches the master node’s taint, allowing this kube-proxy pod to be scheduled to the master node.

|  |
| --- |
|  |

**Note**

Disregard the equal sign, which is shown in the pod’s tolerations, but not in the node’s taints. Kubectl apparently displays taints and tolerations differently when the taint’s/toleration’s value is null.

|  |
| --- |
|  |

**Understanding taint effects**

The two other tolerations on the kube-proxy pod define how long the pod is allowed to run on nodes that aren’t ready or are unreachable (the time in seconds isn’t shown, but can be seen in the pod’s YAML). Those two tolerations refer to the NoExecute instead of the NoSchedule effect.

Each taint has an effect associated with it. Three possible effects exist:

* NoSchedule, which means pods won’t be scheduled to the node if they don’t tolerate the taint.
* PreferNoSchedule is a soft version of NoSchedule, meaning the scheduler will try to avoid scheduling the pod to the node, but will schedule it to the node if it can’t schedule it somewhere else.
* NoExecute, unlike NoSchedule and PreferNoSchedule that only affect scheduling, also affects pods already running on the node. If you add a NoExecute taint to a node, pods that are already running on that node and don’t tolerate the NoExecute taint will be evicted from the node.

**16.1.2. Adding custom taints to a node**

Imagine having a single Kubernetes cluster where you run both production and non-production workloads. It’s of the utmost importance that non-production pods never run on the production nodes. This can be achieved by adding a taint to your production nodes. To add a taint, you use the kubectl taint command:

**$ kubectl taint node node1.k8s node-type=production:NoSchedule**

node "node1.k8s" tainted

This adds a taint with key node-type, value production and the NoSchedule effect. If you now deploy multiple replicas of a regular pod, you’ll see none of them are scheduled to the node you tainted, as shown in the following listing.

**Listing 16.3. Deploying pods without a toleration**

**$ kubectl run test --image busybox --replicas 5 -- sleep 99999**

deployment "test" created

**$ kubectl get po -o wide**

NAME READY STATUS RESTARTS AGE IP NODE

test-196686-46ngl 1/1 Running 0 12s 10.47.0.1 node2.k8s

test-196686-73p89 1/1 Running 0 12s 10.47.0.7 node2.k8s

test-196686-77280 1/1 Running 0 12s 10.47.0.6 node2.k8s

test-196686-h9m8f 1/1 Running 0 12s 10.47.0.5 node2.k8s

test-196686-p85ll 1/1 Running 0 12s 10.47.0.4 node2.k8s

Now, no one can inadvertently deploy pods onto the production nodes.

**16.1.3. Adding tolerations to pods**

To deploy production pods to the production nodes, they need to tolerate the taint you added to the nodes. The manifests of your production pods need to include the YAML snippet shown in the following listing.

**Listing 16.4. A production Deployment with a toleration: production-deployment.yaml**

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: prod

spec:

replicas: 5

template:

spec:

...

tolerations:

- key: node-type ***1***

Operator: Equal ***1***

value: production ***1***

effect: NoSchedule ***1***

* ***1*** This toleration allows the pod to be scheduled to production nodes.

If you deploy this Deployment, you’ll see its pods get deployed to the production node, as shown in the next listing.

**Listing 16.5. Pods with the toleration are deployed on production node1**

**$ kubectl get po -o wide**

NAME READY STATUS RESTARTS AGE IP NODE

prod-350605-1ph5h 0/1 Running 0 16s 10.44.0.3 **node1**.k8s

prod-350605-ctqcr 1/1 Running 0 16s 10.47.0.4 node2.k8s

prod-350605-f7pcc 0/1 Running 0 17s 10.44.0.6 **node1**.k8s

prod-350605-k7c8g 1/1 Running 0 17s 10.47.0.9 node2.k8s

prod-350605-rp1nv 0/1 Running 0 17s 10.44.0.4 **node1**.k8s

As you can see in the listing, production pods were also deployed to node2, which isn’t a production node. To prevent that from happening, you’d also need to taint the non-production nodes with a taint such as node-type=non-production:NoSchedule. Then you’d also need to add the matching toleration to all your non-production pods.

**16.1.4. Understanding what taints and tolerations can be used for**

Nodes can have more than one taint and pods can have more than one toleration. As you’ve seen, taints can only have a key and an effect and don’t require a value. Tolerations can tolerate a specific value by specifying the Equal operator (that’s also the default operator if you don’t specify one), or they can tolerate any value for a specific taint key if you use the Existsoperator.

**Using taints and tolerations during scheduling**

Taints can be used to prevent scheduling of new pods (NoSchedule effect) and to define unpreferred nodes (PreferNoSchedule effect) and even evict existing pods from a node (NoExecute).

You can set up taints and tolerations any way you see fit. For example, you could partition your cluster into multiple partitions, allowing your development teams to schedule pods only to their respective nodes. You can also use taints and tolerations when several of your nodes provide special hardware and only part of your pods need to use it.

**Configuring how long after a node failure a pod is rescheduled**

You can also use a toleration to specify how long Kubernetes should wait before rescheduling a pod to another node if the node the pod is running on becomes unready or unreachable. If you look at the tolerations of one of your pods, you’ll see two tolerations, which are shown in the following listing.

**Listing 16.6. Pod with default tolerations**

**$ kubectl get po prod-350605-1ph5h -o yaml**

...

tolerations:

- effect: NoExecute ***1***

key: node.alpha.kubernetes.io/notReady ***1***

operator: Exists ***1***

tolerationSeconds: 300 ***1***

- effect: NoExecute ***2***

key: node.alpha.kubernetes.io/unreachable ***2***

operator: Exists ***2***

tolerationSeconds: 300 ***2***

* ***1*** **The pod tolerates the node being notReady for 300 seconds, before it needs to be rescheduled.**
* ***2*** **The same applies to the node being unreachable.**

These two tolerations say that this pod tolerates a node being notReady or unreachablefor 300 seconds. The Kubernetes Control Plane, when it detects that a node is no longer ready or no longer reachable, will wait for 300 seconds before it deletes the pod and reschedules it to another node.

These two tolerations are automatically added to pods that don’t define them. If that five-minute delay is too long for your pods, you can make the delay shorter by adding those two tolerations to the pod’s spec.

|  |
| --- |
|  |

**Note**

This is currently an alpha feature, so it may change in future versions of Kubernetes. Taint-based evictions also aren’t enabled by default. You enable them by running the Controller Manager with the --feature-gates=Taint-BasedEvictions=true option.

|  |
| --- |
|  |

**16.2. USING NODE AFFINITY TO ATTRACT PODS TO CERTAIN NODES**

As you’ve learned, taints are used to keep pods away from certain nodes. Now you’ll learn about a newer mechanism called *node affinity*, which allows you to tell Kubernetes to schedule pods only to specific subsets of nodes.

**Comparing node affinity to node selectors**

The initial node affinity mechanism in early versions of Kubernetes was the node-Selectorfield in the pod specification. The node had to include all the labels specified in that field to be eligible to become the target for the pod.

Node selectors get the job done and are simple, but they don’t offer everything that you may need. Because of that, a more powerful mechanism was introduced. Node selectors will eventually be deprecated, so it’s important you understand the new node affinity rules.

Similar to node selectors, each pod can define its own node affinity rules. These allow you to specify either hard requirements or preferences. By specifying a preference, you tell Kubernetes which nodes you prefer for a specific pod, and Kubernetes will try to schedule the pod to one of those nodes. If that’s not possible, it will choose one of the other nodes.

**Examining the default node labels**

Node affinity selects nodes based on their labels, the same way node selectors do. Before you see how to use node affinity, let’s examine the labels of one of the nodes in a Google Kubernetes Engine cluster (GKE) to see what the default node labels are. They’re shown in the following listing.

**Listing 16.7. Default labels of a node in GKE**

**$ kubectl describe node gke-kubia-default-pool-db274c5a-mjnf**

Name: gke-kubia-default-pool-db274c5a-mjnf

Role:

Labels: beta.kubernetes.io/arch=amd64

beta.kubernetes.io/fluentd-ds-ready=true

beta.kubernetes.io/instance-type=f1-micro

beta.kubernetes.io/os=linux

cloud.google.com/gke-nodepool=default-pool

failure-domain.beta.kubernetes.io/region=europe-west1 ***1***

failure-domain.beta.kubernetes.io/zone=europe-west1-d ***1***

kubernetes.io/hostname=gke-kubia-default-pool-db274c5a-mjnf ***1***

* ***1*** **These three labels are the most important ones related to node affinity.**

The node has many labels, but the last three are the most important when it comes to node affinity and pod affinity, which you’ll learn about later. The meaning of those three labels is as follows:

* failure-domain.beta.kubernetes.io/region specifies the geographical region the node is located in.
* failure-domain.beta.kubernetes.io/zone specifies the availability zone the node is in.
* kubernetes.io/hostname is obviously the node’s hostname.

These and other labels can be used in pod affinity rules. In [chapter 3](https://learning.oreilly.com/library/view/kubernetes-in-action/9781617293726/kindle_split_014.html#ch03), you already learned how you can add a custom label to nodes and use it in a pod’s node selector. You used the custom label to deploy pods only to nodes with that label by adding a node selector to the pods. Now, you’ll see how to do the same using node affinity rules.

**16.2.1. Specifying hard node affinity rules**

In the example in [chapter 3](https://learning.oreilly.com/library/view/kubernetes-in-action/9781617293726/kindle_split_014.html#ch03), you used the node selector to deploy a pod that requires a GPU only to nodes that have a GPU. The pod spec included the nodeSelector field shown in the following listing.

**Listing 16.8. A pod using a node selector: kubia-gpu-nodeselector.yaml**

apiVersion: v1

kind: Pod

metadata:

name: kubia-gpu

spec:

nodeSelector: ***1***

gpu: "true" ***1***

...

* ***1*** **This pod is only scheduled to nodes that have the gpu=true label.**

The nodeSelector field specifies that the pod should only be deployed on nodes that include the gpu=true label. If you replace the node selector with a node affinity rule, the pod definition will look like the following listing.

**Listing 16.9. A pod using a nodeAffinity rule: kubia-gpu-nodeaffinity.yaml**

apiVersion: v1

kind: Pod

metadata:

name: kubia-gpu

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: gpu

operator: In

values:

- "true"

The first thing you’ll notice is that this is much more complicated than a simple node selector. But that’s because it’s much more expressive. Let’s examine the rule in detail.

**Making sense of the long nodeAffinity attribute name**

As you can see, the pod’s spec section contains an affinity field that contains a node-Affinity field, which contains a field with an extremely long name, so let’s focus on that first.

Let’s break it down into two parts and examine what they mean:

* requiredDuringScheduling... means the rules defined under this field specify the labels the node must have for the pod to be scheduled to the node.
* ...IgnoredDuringExecution means the rules defined under the field don’t affect pods already executing on the node.

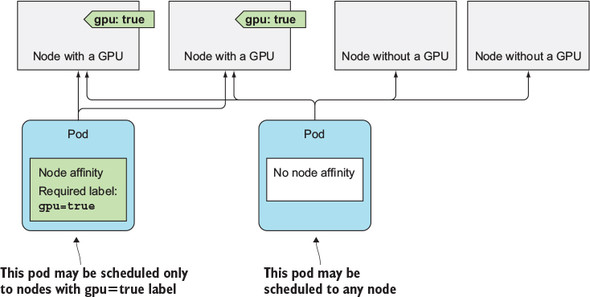
At this point, let me make things easier for you by letting you know that affinity currently only affects pod scheduling and never causes a pod to be evicted from a node. That’s why all the rules right now always end with IgnoredDuringExecution. Eventually, Kubernetes will also support RequiredDuringExecution, which means that if you remove a label from a node, pods that require the node to have that label will be evicted from such a node. As I’ve said, that’s not yet supported in Kubernetes, so let’s not concern ourselves with the second part of that long field any longer.

**Understanding nodeSelectorTerms**

By keeping what was explained in the previous section in mind, it’s easy to understand that the nodeSelectorTerms field and the matchExpressions field define which expressions the node’s labels must match for the pod to be scheduled to the node. The single expression in the example is simple to understand. The node must have a gpu label whose value is set to true.

This pod will therefore only be scheduled to nodes that have the gpu=true label, as shown in [figure 16.2](https://learning.oreilly.com/library/view/kubernetes-in-action/9781617293726/kindle_split_028.html#ch16fig02).

**Figure 16.2. A pod’s node affinity specifies which labels a node must have for the pod to be scheduled to it.**



Now comes the more interesting part. Node also affinity allows you to prioritize nodes during scheduling. We’ll look at that next.

**16.2.2. Prioritizing nodes when scheduling a pod**

The biggest benefit of the newly introduced node affinity feature is the ability to specify which nodes the Scheduler should prefer when scheduling a specific pod. This is done through the preferredDuringSchedulingIgnoredDuringExecution field.

Imagine having multiple datacenters across different countries. Each datacenter represents a separate availability zone. In each zone, you have certain machines meant only for your own use and others that your partner companies can use. You now want to deploy a few pods and you’d prefer them to be scheduled to zone1 and to the machines reserved for your company’s deployments. If those machines don’t have enough room for the pods or if other important reasons exist that prevent them from being scheduled there, you’re okay with them being scheduled to the machines your partners use and to the other zones. Node affinity allows you to do that.

**Labeling nodes**

First, the nodes need to be labeled appropriately. Each node needs to have a label that designates the availability zone the node belongs to and a label marking it as either a dedicated or a shared node.

[Appendix B](https://learning.oreilly.com/library/view/kubernetes-in-action/9781617293726/kindle_split_032.html#app02) explains how to set up a three-node cluster (one master and two worker nodes) in VMs running locally. In the following examples, I’ll use the two worker nodes in that cluster, but you can also use Google Kubernetes Engine or any other multi-node cluster.

|  |
| --- |
|  |

**Note**

Minikube isn’t the best choice for running these examples, because it runs only one node.

|  |
| --- |
|  |

First, label the nodes, as shown in the next listing.

**Listing 16.10. Labeling nodes**

**$ kubectl label node node1.k8s availability-zone=zone1**

node "node1.k8s" labeled

**$ kubectl label node node1.k8s share-type=dedicated**

node "node1.k8s" labeled

**$ kubectl label node node2.k8s availability-zone=zone2**

node "node2.k8s" labeled

**$ kubectl label node node2.k8s share-type=shared**

node "node2.k8s" labeled

**$ kubectl get node -L availability-zone -L share-type**

NAME STATUS AGE VERSION AVAILABILITY-ZONE SHARE-TYPE

master.k8s Ready 4d v1.6.4 <none> <none>

node1.k8s Ready 4d v1.6.4 **zone1 dedicated**

node2.k8s Ready 4d v1.6.4 **zone2 shared**

**Specifying preferential node affinity rules**

With the node labels set up, you can now create a Deployment that prefers dedicated nodes in zone1. The following listing shows the Deployment manifest.

**Listing 16.11. Deployment with preferred node affinity: preferred-deployment.yaml**

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: pref

spec:

template:

...

spec:

affinity:

nodeAffinity:

preferredDuringSchedulingIgnoredDuringExecution: ***1***

- weight: 80 ***2***

preference: ***2***

matchExpressions: ***2***

- key: availability-zone ***2***

operator: In ***2***

values: ***2***

- zone1 ***2***

- weight: 20 ***3***

preference: ***3***

matchExpressions: ***3***

- key: share-type ***3***

operator: In ***3***

values: ***3***

- dedicated ***3***

...

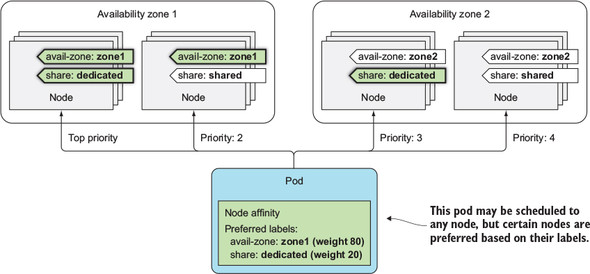
* ***1*** You’re specifying preferences, not hard requirements.
* ***2*** **You prefer the pod to be scheduled to zone1. This is your most important preference.**
* ***3*** **You also prefer that your pods be scheduled to dedicated nodes, but this is four times less important than your zone preference.**

Let’s examine the listing closely. You’re defining a node affinity preference, instead of a hard requirement. You want the pods scheduled to nodes that include the labels availability-zone=zone1 and share-type=dedicated. You’re saying that the first preference rule is important by setting its weight to 80, whereas the second one is much less important (weight is set to 20).

**Understanding how node preferences work**

If your cluster had many nodes, when scheduling the pods of the Deployment in the previous listing, the nodes would be split into four groups, as shown in [figure 16.3](https://learning.oreilly.com/library/view/kubernetes-in-action/9781617293726/kindle_split_028.html#ch16fig03). Nodes whose availability-zone and share-type labels match the pod’s node affinity are ranked the highest. Then, because of how the weights in the pod’s node affinity rules are configured, next come the shared nodes in zone1, then come the dedicated nodes in the other zones, and at the lowest priority are all the other nodes.

**Figure 16.3. Prioritizing nodes based on a pod’s node affinity preferences**



**Deploying the pods in the two-node cluster**

If you create this Deployment in your two-node cluster, you should see most (if not all) of your pods deployed to node1. Examine the following listing to see if that’s true.

**Listing 16.12. Seeing where pods were scheduled**

**$ kubectl get po -o wide**

NAME READY STATUS RESTARTS AGE IP NODE

pref-607515-1rnwv 1/1 Running 0 4m 10.47.0.1 node2.k8s

pref-607515-27wp0 1/1 Running 0 4m 10.44.0.8 **node1**.k8s

pref-607515-5xd0z 1/1 Running 0 4m 10.44.0.5 **node1**.k8s

pref-607515-jx9wt 1/1 Running 0 4m 10.44.0.4 **node1**.k8s

pref-607515-mlgqm 1/1 Running 0 4m 10.44.0.6 **node1**.k8s

Out of the five pods that were created, four of them landed on node1 and only one landed on node2. Why did one of them land on node2 instead of node1? The reason is that besides the node affinity prioritization function, the Scheduler also uses other prioritization functions to decide where to schedule a pod. One of those is the Selector-SpreadPriority function, which makes sure pods belonging to the same ReplicaSet or Service are spread around different nodes so a node failure won’t bring the whole service down. That’s most likely what caused one of the pods to be scheduled to node2.

You can try scaling the Deployment up to 20 or more and you’ll see the majority of pods will be scheduled to node1. In my test, only two out of the 20 were scheduled to node2. If you hadn’t defined any node affinity preferences, the pods would have been spread around the two nodes evenly.

**16.3. CO-LOCATING PODS WITH POD AFFINITY AND ANTI-AFFINITY**

You’ve seen how node affinity rules are used to influence which node a pod is scheduled to. But these rules only affect the affinity between a pod and a node, whereas sometimes you’d like to have the ability to specify the affinity between pods themselves.

For example, imagine having a frontend and a backend pod. Having those pods deployed near to each other reduces latency and improves the performance of the app. You could use node affinity rules to ensure both are deployed to the same node, rack, or datacenter, but then you’d have to specify exactly which node, rack, or datacenter to schedule them to, which is not the best solution. It’s better to let Kubernetes deploy your pods anywhere it sees fit, while keeping the frontend and backend pods close together. This can be achieved using *pod affinity*. Let’s learn more about it with an example.

**16.3.1. Using inter-pod affinity to deploy pods on the same node**

You’ll deploy a backend pod and five frontend pod replicas with pod affinity configured so that they’re all deployed on the same node as the backend pod.

First, deploy the backend pod:

**$ kubectl run backend -l app=backend --image busybox -- sleep 999999**

deployment "backend" created

This Deployment is not special in any way. The only thing you need to note is the app=backend label you added to the pod using the -l option. This label is what you’ll use in the frontend pod’s podAffinity configuration.

**Specifying pod affinity in a pod definition**

The frontend pod’s definition is shown in the following listing.

**Listing 16.13. Pod using podAffinity: frontend-podaffinity-host.yaml**

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: frontend

spec:

replicas: 5

template:

...

spec:

affinity:

podAffinity: ***1***

requiredDuringSchedulingIgnoredDuringExecution: ***2***

- topologyKey: kubernetes.io/hostname ***3***

labelSelector: ***3***

matchLabels: ***3***

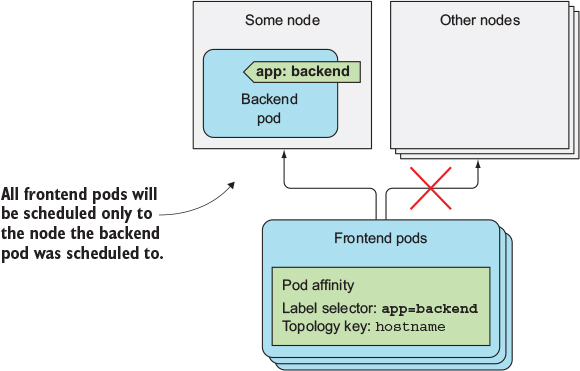
app: backend ***3***

...

* ***1*** Defining podAffinity rules
* ***2*** **Defining a hard requirement, not a preference**
* ***3*** **The pods of this Deployment must be deployed on the same node as the pods that match the selector.**

The listing shows that this Deployment will create pods that have a hard requirement to be deployed on the same node (specified by the topologyKey field) as pods that have the app=backend label (see [figure 16.4](https://learning.oreilly.com/library/view/kubernetes-in-action/9781617293726/kindle_split_028.html#ch16fig04)).

**Figure 16.4. Pod affinity allows scheduling pods to the node where other pods with a specific label are.**



|  |
| --- |
|  |

**Note**

Instead of the simpler matchLabels field, you could also use the more expressive matchExpressions field.

|  |
| --- |
|  |

**Deploying a pod with pod affinity**

Before you create this Deployment, let’s see which node the backend pod was scheduled to earlier:

**$ kubectl get po -o wide**

NAME READY STATUS RESTARTS AGE IP NODE

backend-257820-qhqj6 1/1 Running 0 8m 10.47.0.1 **node2**.k8s

When you create the frontend pods, they should be deployed to node2 as well. You’re going to create the Deployment and see where the pods are deployed. This is shown in the next listing.

**Listing 16.14. Deploying frontend pods and seeing which node they’re scheduled to**

**$ kubectl create -f frontend-podaffinity-host.yaml**

deployment "frontend" created

**$ kubectl get po -o wide**

NAME READY STATUS RESTARTS AGE IP NODE

backend-257820-qhqj6 1/1 Running 0 8m 10.47.0.1 **node2**.k8s

frontend-121895-2c1ts 1/1 Running 0 13s 10.47.0.6 **node2**.k8s

frontend-121895-776m7 1/1 Running 0 13s 10.47.0.4 **node2**.k8s

frontend-121895-7ffsm 1/1 Running 0 13s 10.47.0.8 **node2**.k8s

frontend-121895-fpgm6 1/1 Running 0 13s 10.47.0.7 **node2**.k8s

frontend-121895-vb9ll 1/1 Running 0 13s 10.47.0.5 **node2**.k8s

All the frontend pods were indeed scheduled to the same node as the backend pod. When scheduling the frontend pod, the Scheduler first found all the pods that match the labelSelector defined in the frontend pod’s podAffinity configuration and then scheduled the frontend pod to the same node.

**Understanding how the scheduler uses pod affinity rules**

What’s interesting is that if you now delete the backend pod, the Scheduler will schedule the pod to node2 even though it doesn’t define any pod affinity rules itself (the rules are only on the frontend pods). This makes sense, because otherwise if the backend pod were to be deleted by accident and rescheduled to a different node, the frontend pods’ affinity rules would be broken.

You can confirm the Scheduler takes other pods’ pod affinity rules into account, if you increase the Scheduler’s logging level and then check its log. The following listing shows the relevant log lines.

**Listing 16.15. Scheduler log showing why the backend pod is scheduled to node2**

... Attempting to schedule pod: default/backend-257820-qhqj6

... ...

... backend-qhqj6 -> node2.k8s: Taint Toleration Priority, Score: (10)

... backend-qhqj6 -> node1.k8s: Taint Toleration Priority, Score: (10)

**... backend-qhqj6 -> node2.k8s: InterPodAffinityPriority, Score: (10)**

**... backend-qhqj6 -> node1.k8s: InterPodAffinityPriority, Score: (0)**

... backend-qhqj6 -> node2.k8s: SelectorSpreadPriority, Score: (10)

... backend-qhqj6 -> node1.k8s: SelectorSpreadPriority, Score: (10)

... backend-qhqj6 -> node2.k8s: NodeAffinityPriority, Score: (0)

... backend-qhqj6 -> node1.k8s: NodeAffinityPriority, Score: (0)

... Host node2.k8s => Score 100030

... Host node1.k8s => Score 100022

... Attempting to bind backend-257820-qhqj6 to node2.k8s

If you focus on the two lines in bold, you’ll see that during the scheduling of the backend pod, node2 received a higher score than node1 because of inter-pod affinity.

**16.3.2. Deploying pods in the same rack, availability zone, or geographic region**

In the previous example, you used podAffinity to deploy frontend pods onto the same node as the backend pods. You probably don’t want all your frontend pods to run on the same machine, but you’d still like to keep them close to the backend pod—for example, run them in the same availability zone.

**Co-locating pods in the same availability zone**

The cluster I’m using runs in three VMs on my local machine, so all the nodes are in the same availability zone, so to speak. But if the nodes were in different zones, all I’d need to do to run the frontend pods in the same zone as the backend pod would be to change the topologyKey property to failure-domain.beta.kubernetes.io/zone.

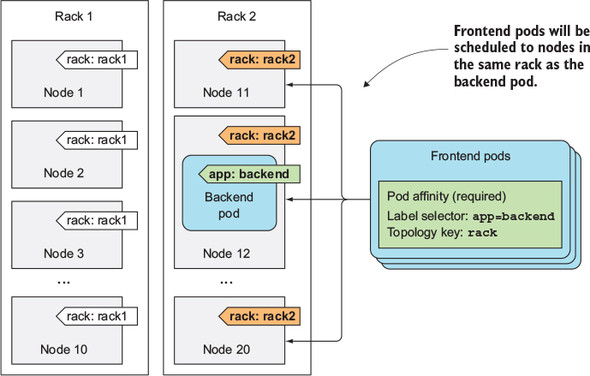
**Co-locating pods in the same geographical region**

To allow the pods to be deployed in the same region instead of the same zone (cloud providers usually have datacenters located in different geographical regions and split into multiple availability zones in each region), the topologyKey would be set to failure-domain.beta.kubernetes.io/region.

**Understanding how topologyKey works**

The way topologyKey works is simple. The three keys we’ve mentioned so far aren’t special. If you want, you can easily use your own topologyKey, such as rack, to have the pods scheduled to the same server rack. The only prerequisite is to add a rack label to your nodes. This scenario is shown in [figure 16.5](https://learning.oreilly.com/library/view/kubernetes-in-action/9781617293726/kindle_split_028.html#ch16fig05).

**Figure 16.5. The topologyKey in podAffinity determines the scope of where the pod should be scheduled to.**



For example, if you had 20 nodes, with 10 in each rack, you’d label the first ten as rack=rack1 and the others as rack=rack2. Then, when defining a pod’s podAffinity, you’d set the toplogyKey to rack.

When the Scheduler is deciding where to deploy a pod, it checks the pod’s pod-Affinityconfig, finds the pods that match the label selector, and looks up the nodes they’re running on. Specifically, it looks up the nodes’ label whose key matches the topologyKey field specified in podAffinity. Then it selects all the nodes whose label matches the values of the pods it found earlier. In [figure 16.5](https://learning.oreilly.com/library/view/kubernetes-in-action/9781617293726/kindle_split_028.html#ch16fig05), the label selector matched the backend pod, which runs on Node 12. The value of the rack label on that node equals rack2, so when scheduling a frontend pod, the Scheduler will only select among the nodes that have the rack=rack2label.

|  |
| --- |
|  |

**Note**

By default, the label selector only matches pods in the same namespace as the pod that’s being scheduled. But you can also select pods from other namespaces by adding a namespacesfield at the same level as label-Selector.

|  |
| --- |
|  |

**16.3.3. Expressing pod affinity preferences instead of hard requirements**

Earlier, when we talked about node affinity, you saw that nodeAffinity can be used to express a hard requirement, which means a pod is only scheduled to nodes that match the node affinity rules. It can also be used to specify node preferences, to instruct the Scheduler to schedule the pod to certain nodes, while allowing it to schedule it anywhere else if those nodes can’t fit the pod for any reason.

The same also applies to podAffinity. You can tell the Scheduler you’d prefer to have your frontend pods scheduled onto the same node as your backend pod, but if that’s not possible, you’re okay with them being scheduled elsewhere. An example of a Deployment using the preferredDuringSchedulingIgnoredDuringExecution pod affinity rule is shown in the next listing.

**Listing 16.16. Pod affinity preference**

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: frontend

spec:

replicas: 5

template:

...

spec:

affinity:

podAffinity:

preferredDuringSchedulingIgnoredDuringExecution: ***1***

- weight: 80 ***2***

podAffinityTerm: ***2***

topologyKey: kubernetes.io/hostname ***2***

labelSelector: ***2***

matchLabels: ***2***

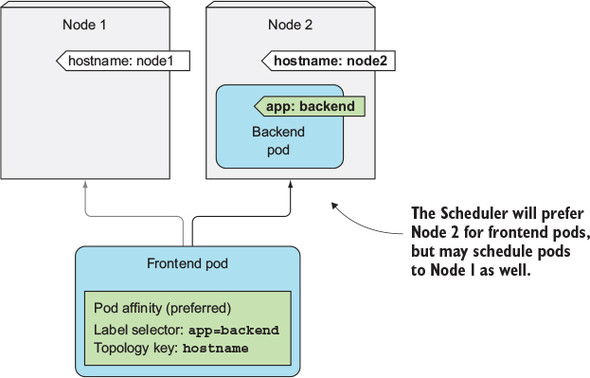
app: backend ***2***

containers: ...

* ***1*** Preferred instead of Required
* ***2*** **A weight and a podAffinity term is specified as in the previous example**

As in nodeAffinity preference rules, you need to define a weight for each rule. You also need to specify the topologyKey and labelSelector, as in the hard-requirement podAffinity rules. [Figure 16.6](https://learning.oreilly.com/library/view/kubernetes-in-action/9781617293726/kindle_split_028.html#ch16fig06) shows this scenario.

**Figure 16.6. Pod affinity can be used to make the Scheduler prefer nodes where pods with a certain label are running.**



Deploying this pod, as with your nodeAffinity example, deploys four pods on the same node as the backend pod, and one pod on the other node (see the following listing).

**Listing 16.17. Pods deployed with podAffinity preferences**

**$ kubectl get po -o wide**

NAME READY STATUS RESTARTS AGE IP NODE

backend-257820-ssrgj 1/1 Running 0 1h 10.47.0.9 node2.k8s

frontend-941083-3mff9 1/1 Running 0 8m 10.44.0.4 node1.k8s

frontend-941083-7fp7d 1/1 Running 0 8m 10.47.0.6 node2.k8s

frontend-941083-cq23b 1/1 Running 0 8m 10.47.0.1 node2.k8s

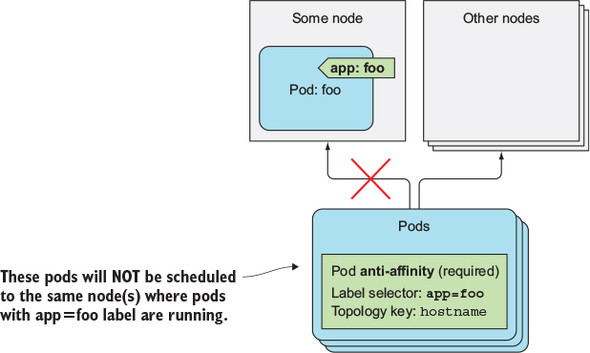
frontend-941083-m70sw 1/1 Running 0 8m 10.47.0.5 node2.k8s

frontend-941083-wsjv8 1/1 Running 0 8m 10.47.0.4 node2.k8s

**16.3.4. Scheduling pods away from each other with pod anti-affinity**

You’ve seen how to tell the Scheduler to co-locate pods, but sometimes you may want the exact opposite. You may want to keep pods away from each other. This is called pod anti-affinity. It’s specified the same way as pod affinity, except that you use the podAntiAffinityproperty instead of podAffinity, which results in the Scheduler never choosing nodes where pods matching the podAntiAffinity’s label selector are running, as shown in [figure 16.7](https://learning.oreilly.com/library/view/kubernetes-in-action/9781617293726/kindle_split_028.html#ch16fig07).

**Figure 16.7. Using pod anti-affinity to keep pods away from nodes that run pods with a certain label.**



An example of why you’d want to use pod anti-affinity is when two sets of pods interfere with each other’s performance if they run on the same node. In that case, you want to tell the Scheduler to never schedule those pods on the same node. Another example would be to force the Scheduler to spread pods of the same group across different availability zones or regions, so that a failure of a whole zone (or region) never brings the service down completely.

**Using anti-affinity to spread apart pods of the same Deployment**

Let’s see how to force your frontend pods to be scheduled to different nodes. The following listing shows how the pods’ anti-affinity is configured.

**Listing 16.18. Pods with anti-affinity: frontend-podantiaffinity-host.yaml**

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: frontend

spec:

replicas: 5

template:

metadata:

labels: ***1***

app: frontend ***1***

spec:

affinity:

podAntiAffinity: ***2***

requiredDuringSchedulingIgnoredDuringExecution: ***2***

- topologyKey: kubernetes.io/hostname ***3***

labelSelector: ***3***

matchLabels: ***3***

app: frontend ***3***

containers: ...

* ***1*** **The frontend pods have the app=frontend label.**
* ***2*** **Defining hard-requirements for pod anti-affinity**
* ***3*** **A frontend pod must not be scheduled to the same machine as a pod with app=frontend label.**

This time, you’re defining podAntiAffinity instead of podAffinity, and you’re making the labelSelector match the same pods that the Deployment creates. Let’s see what happens when you create this Deployment. The pods created by it are shown in the following listing.

**Listing 16.19. Pods created by the Deployment**

**$ kubectl get po -l app=frontend -o wide**

NAME READY STATUS RESTARTS AGE IP NODE

frontend-286632-0lffz 0/1 Pending 0 1m <none>

frontend-286632-2rkcz 1/1 Running 0 1m 10.47.0.1 node2.k8s

frontend-286632-4nwhp 0/1 Pending 0 1m <none>

frontend-286632-h4686 0/1 Pending 0 1m <none>

frontend-286632-st222 1/1 Running 0 1m 10.44.0.4 node1.k8s

As you can see, only two pods were scheduled—one to node1, the other to node2. The three remaining pods are all Pending, because the Scheduler isn’t allowed to schedule them to the same nodes.

**Using preferential pod anti-affinity**

In this case, you probably should have specified a soft requirement instead (using the preferredDuringSchedulingIgnoredDuringExecution property). After all, it’s not such a big problem if two frontend pods run on the same node. But in scenarios where that’s a problem, using requiredDuringScheduling is appropriate.

As with pod affinity, the topologyKey property determines the scope of where the pod shouldn’t be deployed to. You can use it to ensure pods aren’t deployed to the same rack, availability zone, region, or any custom scope you create using custom node labels.

**16.4. SUMMARY**

In this chapter, we looked at how to ensure pods aren’t scheduled to certain nodes or are only scheduled to specific nodes, either because of the node’s labels or because of the pods running on them.

You learned that

* If you add a taint to a node, pods won’t be scheduled to that node unless they tolerate that taint.
* Three types of taints exist: NoSchedule completely prevents scheduling, Prefer-NoSchedule isn’t as strict, and NoExecute even evicts existing pods from a node.
* The NoExecute taint is also used to specify how long the Control Plane should wait before rescheduling the pod when the node it runs on becomes unreachable or unready.
* Node affinity allows you to specify which nodes a pod should be scheduled to. It can be used to specify a hard requirement or to only express a node preference.
* Pod affinity is used to make the Scheduler deploy pods to the same node where another pod is running (based on the pod’s labels).
* Pod affinity’s topologyKey specifies how close the pod should be deployed to the other pod (onto the same node or onto a node in the same rack, availability zone, or availability region).
* Pod anti-affinity can be used to keep certain pods away from each other.
* Both pod affinity and anti-affinity, like node affinity, can either specify hard requirements or preferences.

In the next chapter, you’ll learn about best practices for developing apps and how to make them run smoothly in a Kubernetes environment.