Learning Objectives

By the end of this chapter, you should be able to:

- Learn how kube-scheduler schedules Pod placement.
- Use Labels to manage Pod scheduling.
- Configure taints and tolerations.
- Use **podAffinity** and **podAntiAffinity**.
- Understand how to run multiple schedulers.

SCHEDULING

kube-scheduler

The larger and more diverse a Kubernetes deployment becomes, the more administration of scheduling can be important. The kube-scheduler determines which nodes will run a Pod, using a topology-aware algorithm.

Users can set the priority of a pod, which will allow preemption of lower priority pods. The eviction of lower priority pods would then allow the higher priority pod to be scheduled.

The scheduler tracks the set of nodes in your cluster, filters them based on a set of predicates, then uses priority functions to determine on which node each Pod should be scheduled. The Pod specification as part of a request is sent to the kubelet on the node for creation.

The default scheduling decision can be affected through the use of Labels on nodes or Pods. Labels of podAffinity, taints, and pod bindings allow for configuration from the Pod or the node perspective. Some, like tolerations, allow a Pod to work with a node, even when the node has a taint that would otherwise preclude a Pod being scheduled.

Not all labels are drastic. Affinity settings may encourage a Pod to be deployed on a node, but would deploy the Pod elsewhere if the node was not available. Sometimes, documentation may use the term *require*, but practice shows the setting to be more of a request. As beta features, expect the specifics to change. Some settings will evict Pods from a node should the required condition no longer be true, such as **requiredDuringScheduling**, **RequiredDuringExecution**.

Other options, like a custom scheduler, need to be programmed and deployed into your Kubernetes cluster.

Predicates

The scheduler goes through a set of filters, or predicates, to find available nodes, then ranks each node using priority functions. The node with the highest rank is selected to run the Pod.

```
predicatesOrdering = []string{CheckNodeConditionPred, GeneralPred,
HostNamePred, PodFitsHostPortsPred,MatchNodeSelectorPred,
PodFitsResourcesPred, NoDiskConflictPred,PodToleratesNodeTaintsPred,
PodToleratesNodeNoExecuteTaintsPred,CheckNodeLabelPresencePred,
checkServiceAffinityPred,MaxEBSVolumeCountPred,
MaxGCEPDVolumeCountPred,MaxAzureDiskVolumeCountPred,
CheckVolumeBindingPred,NoVolumeZoneConflictPred,
CheckNodeMemoryPressurePred,CheckNodeDiskPressurePred,
MatchInterPodAffinityPred}
```

The predicates, such as **PodFitsHost** or **NoDiskConflict**, are evaluated in a particular and configurable order. In this way, a node has the least amount of checks for new Pod deployment, which can be useful to exclude a node from unnecessary checks if the node is not in the proper condition.

For example, there is a filter called **HostNamePred**, which is also known as **HostName**, which filters out nodes that do not match the node name specified in the pod specification. Another predicate is **PodFitsResources** to make sure that the available CPU and memory can fit the resources required by the Pod.

The scheduler can be updated by passing a configuration of **kind: Policy**, which can order predicates, give special weights to priorities, and even **hardPodAffinitySymmetricWeight**, which deploys Pods such that if we set Pod A to run with Pod B, then Pod B should automatically be run with Pod A.

Priorities

Priorities are functions used to weight resources. Unless Pod and node affinity has been configured to the SelectorSpreadPriority setting, which ranks nodes based on the number of existing running pods, they will select the node with the least amount of Pods. This is a basic way to spread Pods across the cluster.

Other priorities can be used for particular cluster needs. The **ImageLocalityPriorityMap** favors nodes which already have downloaded container images. The total sum of image size is compared with the largest having the highest priority, but does not check the image about to be used.

Currently, there are more than ten included priorities, which range from checking the existence of a label to choosing a node with the most requested CPU and memory usage. You can view a list of priorities at **master/pkg/scheduler/algorithm/priorities**.

A stable feature as of v1.14 allows the setting of a **PriorityClass** and assigning pods via the use of **PriorityClassName** settings. This allows users to preempt, or evict, lower priority pods so that their higher priority pods can be scheduled. The kube-scheduler determines a node where the pending pod could run if one or more existing pods were evicted. If a node is found, the low priority pod(s) are evicted and the higher priority pod is scheduled. The use of a Pod Disruption Budget (PDB) is a way to limit the number of pods preemption evicts to ensure enough pods remain running. The scheduler will remove pods even if the PDB is violated if no other options

are available.

Scheduling Policies

The default scheduler contains a number of predicates and priorities; however, these can be changed via a scheduler policy file.

A short version is shown below:

```
"kind" : "Policy",
"apiVersion" : "v1",
"predicates" : [
        {"name" : "MatchNodeSelector", "order": 6},
        {"name" : "PodFitsHostPorts", "order": 2},
        {"name" : "PodFitsResources", "order": 3},
        {"name" : "NoDiskConflict", "order": 4},
        {"name" : "PodToleratesNodeTaints", "order": 5},
        {"name" : "PodFitsHost", "order": 1}
        ],
"priorities" : [
        {"name" : "LeastRequestedPriority", "weight" : 1},
        {"name" : "BalancedResourceAllocation", "weight" : 1},
        {"name" : "ServiceSpreadingPriority", "weight" : 2},
        {"name" : "EqualPriority", "weight" : 1}
        ],
"hardPodAffinitySymmetricWeight" : 10
}
```

Typically, you will configure a scheduler with this policy using the **--policy-config-file** parameter and define a name for this scheduler using the **--scheduler-name** parameter. You will then have two schedulers running and will be able to specify which scheduler to use in the pod specification.

With multiple schedulers, there could be conflict in the Pod allocation. Each Pod should declare which scheduler should be used. But, if separate schedulers determine that a node is eligible because of available resources and both attempt to deploy, causing the resource to no longer be available, a conflict would occur. The current solution is for the local kubelet to return the Pods to the scheduler for reassignment. Eventually, one Pod will succeed and the other will be scheduled elsewhere.

Pod Specification

Most scheduling decisions can be made as part of the Pod specification. A pod specification contains several fields that inform scheduling, namely:

- **nodeName**

- **nodeSelector**
- **affinity**
- **schedulerName**
- **tolerations**

Fields in a Pod Specification

- nodeName and nodeSelector

The **nodeName** and **nodeSelector** options allow a Pod to be assigned to a single node or a group of nodes with particular labels.

- affinity and anti-affinity

Affinity and anti-affinity can be used to require or prefer which node is used by the scheduler. If using a preference instead, a matching node is chosen first, but other nodes would be used if no match is present.

- taints and tolerations

The use of taints allows a node to be labeled such that Pods would not be scheduled for some reason, such as the master node after initialization. A toleration allows a Pod to ignore the taint and be scheduled assuming other requirements are met.

- schedulerName

Should none of the options above meet the needs of the cluster, there is also the ability to deploy a custom scheduler. Each Pod could then include a **schedulerName** to choose which schedule to use.

Specifying the Node Label

The **nodeSelector** field in a pod specification provides a straightforward way to target a node or a set of nodes, using one or more key-value pairs.

```
spec:
  containers:
  - name: redis
    image: redis
  nodeSelector:
    net: fast
```

Setting the **nodeSelector** tells the scheduler to place the pod on a node that matches the labels. All listed selectors must be met, but the node could have more labels. In the example above, any node with a key of **net** set to **fast** would be a candidate for scheduling. Remember that labels are administrator-created tags, with no tie to actual resources. This node could have a slow network.

The pod would remain Pending until a node is found with the matching labels.

The use of affinity/anti-affinity should be able to express every feature as nodeSelector.

Pod Affinity Rules

Pods which may communicate a lot or share data may operate best if co-located, which would be a form of affinity. For greater fault tolerance, you may want Pods to be as separate as possible, which would be anti-affinity. These settings are used by the scheduler based on the labels of Pods that are already running. As a result, the scheduler must interrogate each node and track the labels of running Pods. Clusters larger than several hundred nodes may see significant performance loss. Pod affinity rules use **In**, **NotIn**, **Exists**, and **DoesNotExist** operators.

Pod Affinity Rules

- requiredDuringSchedulingIgnoredDuringExecution

The use of **requiredDuringSchedulingIgnoredDuringExecution** means that the Pod will not be scheduled on a node unless the following operator is true. If the operator changes to become false in the future, the Pod will continue to run. This could be seen as a hard rule.

- preferredDuringSchedulingIgnoredDuringExecution

Similarly, **preferredDuringSchedulingIgnoredDuringExecution** will choose a node with the desired setting before those without. If no properly-labeled nodes are available, the Pod will execute anyway. This is more of a soft setting, which declares a preference instead of a requirement.

- podAffinity

With the use of **podAffinity**, the scheduler will try to schedule Pods together.

- podAntiAffinity

The use of **podAntiAffinity** would cause the scheduler to keep Pods on different nodes.

- topologyKey

  The **topologyKey** allows a general grouping of Pod deployments. Affinity (or the inverse anti-affinity) will try to run on nodes with the declared topology key and running Pods with a particular label. The **topologyKey** could be any legal key, with some important considerations.

  - If using **requiredDuringScheduling** and the admission controller **LimitPodHardAntiAffinityTopology** setting, the **topologyKey** must be set to **kubernetes.io/hostname**.
  - If using **PreferredDuringScheduling**, an empty **topologyKey** is assumed to be all, or the combination of **kubernetes.io/hostname**, **topology.kubernetes.io/zone** and **topology.kubernetes.io/region**.

podAffinity Example

An example of **affinity** and **podAffinity** settings can be seen below. This also requires a particular label to be matched when the Pod starts, but not required if the label is later removed.

```
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - S1
        topologyKey: topology.kubernetes.io/zone
```

Inside the declared topology zone, the Pod can be scheduled on a node running a Pod with a key label of **security** and a value of **S1**. If this requirement is not met, the Pod will remain in a **Pending** state.

podAntiAffinity Example

With **podAntiAffinity**, we can prefer to avoid nodes with a particular label. In this case, the scheduler will prefer to avoid a node with a key set to **security** and value of **S2**.

```
podAntiAffinity:
  preferredDuringSchedulingIgnoredDuringExecution:
  - weight: 100
    podAffinityTerm:
      labelSelector:
        matchExpressions:
        - key: security
          operator: In
          values:
          - S2
    topologyKey: kubernetes.io/hostname
```

In a large, varied environment, there may be multiple situations to be avoided. As a preference, this setting tries to avoid certain labels, but will still schedule the Pod on some node. As the Pod will still run, we can provide a weight to a particular rule. The weights can be declared as a value from 1 to 100. The scheduler then tries to choose, or avoid the node with the greatest combined value.

Node Affinity Rules

Where Pod affinity/anti-affinity has to do with other Pods, the use of **nodeAffinity** allows Pod scheduling based on node labels. This is similar and will some day replace the use of the **nodeSelector** setting. The scheduler will not look at other Pods on the system, but the labels of the nodes. This should have much less performance impact on the cluster, even with a large

number of nodes.

- Uses **In**, **NotIn**, **Exists**, **DoesNotExist** operators
- **requiredDuringSchedulingIgnoredDuringExecution**
- **preferredDuringSchedulingIgnoredDuringExecution**
- Planned for future: **requiredDuringSchedulingRequiredDuringExecution**.

Until **nodeSelector** has been fully deprecated, both the selector and required labels must be met for a Pod to be scheduled.

Node Affinity Example

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/colo-tx-name
            operator: In
            values:
            - tx-aus
            - tx-dal
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: disk-speed
            operator: In
            values:
            - fast
            - quick
```

The first **nodeAffinity** rule requires a node with a key of **kubernetes.io/colo-tx-name** which has one of two possible values: **tx-aus** or **tx-dal**.

The second rule gives extra weight to nodes with a key of **disk-speed** with a value of **fast** or **quick**. The Pod will be scheduled on some node - in any case, this just prefers a particular label.

Taints

A node with a particular taint will repel Pods without tolerations for that taint. A taint is expressed as **key=value:effect**. The key and the value are created by the administrator.

The key and value used can be any legal string, and this allows flexibility to prevent Pods from running on nodes based off of any need. If a Pod does not have an existing toleration, the scheduler will not consider the tainted node.

Ways to Handle Pod Scheduling

- NoSchedule

  The scheduler will not schedule a Pod on this node, unless the Pod has this toleration. Existing Pods continue to run, regardless of toleration.

- PreferNoSchedule

  The scheduler will avoid using this node, unless there are no untainted nodes for the Pods toleration. Existing Pods are unaffected.

- NoExecute

  This taint will cause existing Pods to be evacuated and no future Pods scheduled. Should an existing Pod have a toleration, it will continue to run. If the Pod **tolerationSeconds** is set, they will remain for that many seconds, then be evicted. Certain node issues will cause the kubelet to add 300 second tolerations to avoid unnecessary evictions.

If a node has multiple taints, the scheduler ignores those with matching tolerations. The remaining unignored taints have their typical effect.

The use of **TaintBasedEvictions** is still an alpha feature. The kubelet uses taints to rate-limit evictions when the node has problems.

Tolerations

Setting tolerations on a node are used to schedule Pods on tainted nodes. This provides an easy way to avoid Pods using the node. Only those with a particular toleration would be scheduled.

An operator can be included in a Pod specification, defaulting to **Equal** if not declared. The use of the operator **Equal** requires a value to match. The **Exists** operator should not be specified. If an empty key uses the **Exists** operator, it will tolerate every taint. If there is no effect, but a key and operator are declared, all effects are matched with the declared key.

```
tolerations:
- key: "server"
  operator: "Equal"
  value: "ap-east"
  effect: "NoExecute"
  tolerationSeconds: 3600
```

In the above example, the Pod will remain on the server with a key of **server** and a value of **ap-east** for 3600 seconds after the node has been tainted with **NoExecute**. When the time runs out, the Pod will be evicted.

Custom Scheduler

If the default scheduling mechanisms (affinity, taints, policies) are not flexible enough for your needs, you can write your own scheduler. The programming of a custom scheduler is outside the scope of this course, but you may want to start with the existing scheduler code, which can be found in the Scheduler repository on GitHub.

If a Pod specification does not declare which scheduler to use, the standard scheduler is used by default. If the Pod declares a scheduler, and that container is not running, the Pod would remain in a **Pending** state forever.

The end result of the scheduling process is that a pod gets a binding that specifies which node it should run on. A binding is a Kubernetes API primitive in the **api/v1** group. Technically, without any scheduler running, you could still schedule a pod on a node, by specifying a binding for that pod.

You can also run multiple schedulers simultaneously.

You can view the scheduler and other information with:

```
$ kubectl get events
```

ß