Learn How to Assign Pods to Nodes in Kubernetes Using nodeSelector and Affinity Features

As we've already discussed in our earlier tutorials, Kubernetes administrators normally don't need to choose a node to which their Pods should be scheduled. Instead, the selection of the appropriate node(s) onto which to schedule their Pods is handled by the Kubernetes scheduler. Automatic node selection prevents users from selecting unhealthy nodes or nodes with a shortage of resources.

Kubernetes scheduler ensures that the right node is selected by checking the node's capacity for CPU and RAM and comparing it to the Pod's resource requests. The scheduler makes sure that, for each of these resource types, the sum of all resource requests by the Pods' containers is less than the capacity of the node. This mechanism ensures that Pods end up on nodes with spare resources.

However, there are some scenarios when you want your Pods to end up on specific nodes. For example:

- You want your Pod(s) to end up on a machine with the SSD attached to it.
- You want to co-locate Pods on a particular machine(s) from the same availability zone.
- You want to co-locate a Pod from one Service with a Pod from another service on the same node because these Services strongly depend on each other. For example, you may want to place a web server on the same node as the in-memory cache store like Memcached (see the example below).

These scenarios are addressed by a number of primitives in Kubernetes:

- `nodeSelector` — This is a simple Pod scheduling feature that allows scheduling a Pod onto a node whose labels match the `nodeSelector` labels specified by the user.
- Node Affinity — This is the enhanced version of the `nodeSelector` introduced in Kubernetes 1.4 in beta. It offers a more expressive syntax for fine-grained control of how Pods are scheduled to specific nodes.
- Inter-Pod Affinity — This feature addresses the third scenario above. Inter-Pod affinity allows co-location by scheduling Pods onto nodes that already have specific Pods running.

In addition to the mentioned scenarios, you may want to prevent Pods to be scheduled on specific nodes or be co-located on nodes with specific Pods. This feature is the opposite to node and Pod affinity and is known as node/Pod anti-affinity. Anti-affinity can be implemented in Kubernetes using:

- `spec.affinity.podAntiAffinity` field of the Pod spec. There, we can specify a list of labels which will be compared with the labels of Pods running on the node. If the labels match, the Pod won't be placed alongside with the Pod having this label and running on the node.
- Taints and tolerations, which ensure that a given Pod does not end up on the inappropriate node. If the taint is applied to a node, only those Pods that have tolerations for this taint can be scheduled onto that node.

In what follows, we will describe how to use these concepts to control Pod scheduling in your Kubernetes cluster. In this tutorial, we'll focus on such concepts as `nodeSelector`, node affinity/anti-affinity, and inter-Pod affinity/anti-affinity, and we will illustrate how they work using some practical examples. In a subsequent tutorial, we'll describe taints and tolerations in more detail. Let's begin our discussion with the `nodeSelector` feature.

nodeSelector

As we've already mentioned, `nodeSelector` is the early Kubernetes feature designed for manual Pod scheduling. The basic idea behind the `nodeSelector` is to allow a Pod to be scheduled only on those nodes that have label(s) identical to the label(s) defined in the `nodeSelector`. The latter are key-value pairs that can be specified inside the `PodSpec`.

Applying `nodeSelector` to the Pod involves several steps. We first need to assign a label to some node that will be later used by the `nodeSelector` . Let's find what nodes exist in your cluster. To get the name of these nodes, you can run:

```
kubectl get nodes --show-labelsNAME        STATUS
ROLES                      AGE       VERSION   LABELShost01
Ready     controlplane,etcd,worker   61d       v1.10.5   beta.
kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io
/hostname=host01,node-role.kubernetes.io/controlplane=true,node-role.
kubernetes.io/etcd=true,node-role.kubernetes.io/worker=truehost02
Ready     etcd,worker                61d       v1.10.5   beta.
kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io
/hostname=host02,node-role.kubernetes.io/etcd=true,node-role.kubernetes.
io/worker=truehost03      Ready     etcd,worker
61d       v1.10.5   beta.kubernetes.io/arch=amd64,beta.kubernetes.io
/os=linux,kubernetes.io/hostname=host03,node-role.kubernetes.io
/etcd=true,node-role.kubernetes.io/worker=true
```

As you see, we have three nodes in the cluster: `host01`, `host02`, and `host03`.

Next, select a node to which you want to add a label. For example, let's say we want to add a new label with the key `disktype` and value `ssd` to the `host02` node, which is a node with the SSD storage. To do so, run:

```
kubectl label nodes host02 disktype=ssd
node “host02” labeled
```

As you noticed, the command above follows the format `kubectl label nodes <node-name> <label-key>=<label-value>` .

Finally, let's verify that the new label was added by running:

```
kubectl get nodes --show-labelsNAME        STATUS
ROLES                      AGE       VERSION   LABELShost01
Ready     controlplane,etcd,worker   61d       v1.10.5   beta.
kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io
/hostname=host01,node-role.kubernetes.io/controlplane=true,node-role.
kubernetes.io/etcd=true,node-role.kubernetes.io/worker=truehost02
Ready     etcd,worker                61d       v1.10.5   beta.
kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io
/hostname=host02, disktype=ssd,node-role.kubernetes.io/etcd=true,node-
role.kubernetes.io/worker=truehost03      Ready     etcd,
worker                61d       v1.10.5   beta.kubernetes.io/arch=amd64,
beta.kubernetes.io/os=linux,kubernetes.io/hostname=host03,node-role.
kubernetes.io/etcd=true,node-role.kubernetes.io/worker=true
```

As you see, the `host02` now has a new label `disktype=ssd`. To see all labels attached to the node, you can also run:

```
kubectl describe node "host02"Name:
host02Roles:              nodeLabels:             beta.kubernetes.io
/arch=amd64,
                    beta.kubernetes.io/os=linux,
                    kubernetes.io/hostname=host02,
                    disktype=ssd,
                    node-role.kubernetes.io/etcd=true,
                    node-role.kubernetes.io/worker=true
```

Along with the `disktype=ssd` label we've just added, you can see other labels such as `beta.kubernetes.io/arch` or `kubernetes.io/hostname` . These are all default labels attached to Kubernetes nodes. Some of them define the node's architecture and OS or the name of the host:

- `kubernetes.io/hostname`
- `failure-domain.beta.kubernetes.io/zone`
- `failure-domain.beta.kubernetes.io/region`
- `beta.kubernetes.io/instance-type`
- `beta.kubernetes.io/os`
- `beta.kubernetes.io/arch`

Note: values of these labels may vary depending on the environment (e.g., OS), cloud provider, etc.

In order to assign a Pod to the node with the label we just added, you need to specify a `nodeSelector` field in the `PodSpec`. You can have a manifest that looks something like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: httpd
  labels:
    env: prod
spec:
  containers:
  - name: httpd
    image: httpd
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

We add the `spec.nodeSelector` field to the `PodSpec` with the label `disktype:ssd` identical to the node's label.

Save this spec to the `test-pod.yaml` and run:

```
kubectl create -f test-pod.yaml
```

Upon running this command, your `httpd` Pod will be scheduled on the node with the `disktype=ssd` label (`host02` node in our case). You can verify that by running `kubectl get pods -o wide` and looking at the "NODE" to which the Pod was assigned.

```
NAME                            READY       STATUS  RESTARTS
IP            NODEpod-test-657c7bccfd-n4jc8       2/2       Running
0      172.17.0.7     host03httpd                          2/2
Running     0      172.17.0.21    host02........
```

Note: `nodeSelector` is a very simple feature for Pod scheduling that has limitations. After the introduction of node affinity in Kubernetes 1.2 as alpha, Kubernetes users have a more flexible and useful mechanism for Pod scheduling that can do more than `nodeSelector`. Therefore, in the near future, `nodeSelectors` will be deprecated, although they still work as usual in the latest Kubernetes releases.

Note on Node Isolation

In many cases, you will use `nodeSelector` to run your Pods on nodes with certain security parameters, isolation, or access control credentials. In this case, you want to avoid situations when the malicious user who compromised the node sets labels that attract your Pods onto that node.

How do you prevent such a situation? Kubernetes has a `NodeRestriction` admission plugin that prevents kubelets from setting or modifying labels with a `node-restriction.kubernetes.io/` prefix. There are two prerequisites for using this prefix to secure your nodes:

- Make sure you use the Node authorizer and have enabled the NodeRestriction admission plugin.
- Add labels under `node-restriction.kubernetes.io/` prefix to your Node objects, and use those labels in your node selectors. For example, `test.com.node-restriction.kubernetes.io/fips=true` or `test.com.node-restriction.kubernetes.io/pci-dss=true`.

Node and Pod Affinity and Anti-Affinity

As we've mentioned earlier, `nodeSelector` is the simplest Pod scheduling constraint in Kubernetes. In Kubernetes 1.2, a new node and Pod affinity feature was added as alpha and graduated to beta in Kubernetes 1.6. The affinity greatly expands the `nodeSelector` functionality introducing the following improvements:

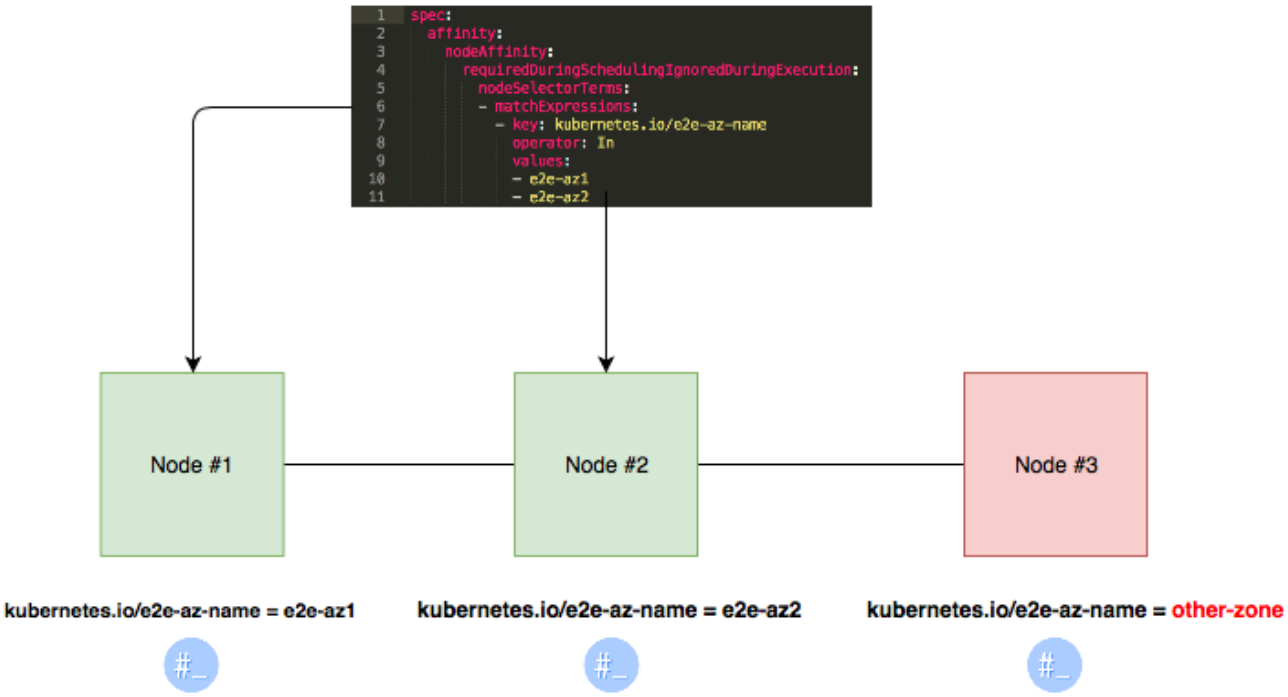
1. Affinity language is more expressive (more logical operators to control how Pods are scheduled).
2. Users can now "soft" scheduling rules. If the "soft" rule is not met, the scheduler can still schedule a Pod onto a specific node.
3. New affinity feature supports Pod co-location. Users can constraint a Pod against the label of another Pod running on a specific node rather than against node labels. With this feature, users can control what Pods end up on the same node and which one don't. This feature is called "inter-Pod affinity/anti-affinity."

Essentially, there are two types of affinity in Kubernetes: *node* affinity and *Pod* affinity. Similarly to `nodeSelector`, node affinity attracts a Pod to certain nodes, whereas the Pod affinity attracts a Pod to certain Pods. In addition to that, Kubernetes supports Pod anti-affinity, which repels a Pod from other Pods. Users can also implement node anti-affinity using logical operators. In what follows, we discuss node and Pod affinity/anti-affinity in more detail.

Node Affinity

Node affinity allows scheduling Pods to specific nodes. There are a number of use cases for node affinity, including the following:

- Spreading Pods across different availability zones to improve resilience and availability of applications in the cluster (see the image below).
- Allocating nodes for memory-intensive Pods. In this case, you can have a few nodes dedicated to less compute-intensive Pods and one or two nodes with enough CPU and RAM dedicated to memory-intensive Pods. This way you prevent undesired Pods from consuming resources dedicated to other Pods.



One of the best features of the current affinity implementation in Kubernetes is the support for “hard” and “soft” node affinity.

With “hard” affinity, users can set a precise rule that should be met in order for a Pod to be scheduled on a node. For example, using “hard” affinity you can tell the scheduler to run the Pod only on the node that has SSD storage attached to it.

As the name suggests, “soft” affinity is less strict. Using “soft” affinity, you can ask the scheduler to try to run the set of Pod in availability zone XYZ, but if it’s impossible, allow some of these Pods to run in the other Availability Zone.

The image above demonstrates the “hard” node affinity. As you see in the manifest, it is defined by the `requiredDuringSchedulingIgnoredDuringExecution` field of the `PodSpec`. In its turn, “soft” node affinity is defined by the `preferredDuringSchedulingIgnoredDuringExecution` field.

Note: `IgnoredDuringExecution` part in both names means that if labels on a node will be changed after the Pod matching these labels is scheduled, it will still continue to run on that node. There is still an unimplemented `requiredDuringSchedulingRequiredDuringExecution` feature that allows evicting Pods from nodes that no longer satisfy the Pod’s node affinity requirements.

To get the feel of how node affinity works, let’s look at this example:

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/e2e-az-name
            operator: In
            values:
            - e2e-az1
            - e2e-az2
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: custom-key
            operator: In
            values:
            - custom-value
  containers:
  - name: with-node-affinity
    image: k8s.gcr.io/pause:2.0
```

As you see, the node affinity is specified under `nodeAffinity` field of the affinity in the `PodSpec`. We define “hard” and “soft” affinity rules in the same `PodSpec`.

The “hard” affinity rule tells the scheduler to place the Pod only onto a node with the label whose key is `kubernetes.io/e2e-az-name` and the value is either `e2e-az1` or `e2e-az2` .

The “soft” rule says that among the nodes that meet “hard” criteria, we prefer the Pod to be placed on the nodes that have a label with the key `custom-key` and value `custom-value`. Because this rule is “soft,” if there are no such nodes, the Pod will still be scheduled if the “hard” rule is met.

In the example above, we used the `In` operator in `matchExpressions`. This operator matches the key’s value if it includes the “expression” specified. Other supported operators include `NotIn` , `Exists` , `DoesNotExist` , `Gt` , `Lt` By the way, you can use `NotIn` and `DoesNotExist` to achieve node anti-affinity behavior.

Also, the “soft” affinity rule includes the weight field that takes values in the range from 1 to 100. This value is used to calculate the priority of nodes to take Pods for scheduling. To learn more about this “weight” parameter, consult the official documentation.

Inter-Pod Affinity/Anti-Affinity

With Inter-Pod affinity/anti-affinity, you can define whether a given Pod should or should not be scheduled onto a particular node based on labels of other Pods already running on that node.

There are a number of use cases for Pod Affinity, including the following:

- Co-locate the Pods from a particular service or Job in the same availability zone.
- Co-locate the Pods from two services dependent on each other on one node to reduce network latency between them. Co-location might mean same nodes and/or same availability zone.

In its turn, Pod anti-affinity is typically used for the following use cases:

- Spread the Pods of a service across nodes and/or availability zones to reduce correlated failures. For example, we may want to prevent data Pods of some database (e.g., Elasticsearch) to live on the same node to avoid the single point of failure.
- Give a Pod “exclusive” access to a node to guarantee resource isolation.
- Don’t schedule the Pods of a particular service on the same nodes as Pods of another service that may interfere with the performance of the Pods of the first service.

The Pod affinity/anti-affinity may be formalized as follows. “this Pod should or should not run in an X node if that X node is already running one or more pods that meet rule Y”.

Y is a `LabelSelector` of Pods running on that node.

X may be a node, rack, CSP region, CSP zone, etc. Users can express it using a `topologyKey` .

Similarly to node affinity, Pod affinity and anti-affinity support “hard” and “soft” rules. Inter-pod affinity is specified under the field `affinity.podAffinity` of the `PodSpec`. And inter-pod anti-affinity is specified under `affinity.podAntiAffinity` in the `PodSpec`. Take a look at this manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - S1
        topologyKey: failure-domain.beta.kubernetes.io/zone
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: security
              operator: In
              values:
              - S2
          topologyKey: kubernetes.io/hostname
  containers:
  - name: with-pod-affinity
    image: k8s.gcr.io/pause:2.0
```

In the manifest above, we defined one Pod affinity rule and one Pod anti-affinity rule. Specifically, we used the “hard” `requiredDuringSchedulingIgnoredDuringExecution` for `podAffinity` and “soft” `preferredDuringSchedulingIgnoredDuringExecution` for `podAntiAffinity` .

The Pod affinity rule tells that our Pod can be scheduled to a node if two conditions are met:

- The node has a label with the key `failure-domain.beta.kubernetes.io/zone` and some value V.

- There is at least one node with key `failure-domain.beta.kubernetes.io/zone` and value V with already-running Pod that has a label with key the `security` and value `S1`

In its turn, the Pod anti-affinity rule tells that we prefer not to schedule Pods to nodes that have a key `kubernetes.io/hostname` granted they have Pods with the label `security:S2` running on them.

Node: Pod anti-affinity requires nodes to be consistently labeled. This means that every node in the cluster must have an appropriate label matching `topologyKey` . If some nodes are missing the specified `topologyKey` label, this may lead to unintended behavior.

Tutorial

In what follows, we'll demonstrate how to configure Pod affinity anti-affinity for Pods and Deployments in your cluster. In order to test the examples presented below, the following prerequisites are required:

- A running Kubernetes cluster. See Supergiant documentation for more information about deploying a Kubernetes cluster with Supergiant. As an alternative, you can install a single-node Kubernetes cluster on a local system using Minikube.
- A **kubectl** command line tool installed and configured to communicate with the cluster. See how to install **kubectl** here.

Configuring Pod Anti-Affinity

To implement Pod anti-affinity, we need at least two Pods. The first Pod is a Pod with labels against which we define the anti-affinity rule. The second one is the Pod with the Pod anti-affinity rule used to prevent co-location of the first and the second Pod.

Let's first define the first Pod against which we'll later create the anti-affinity rule:

```
apiVersion: v1
kind: Pod
metadata:
  name: s1
  labels:
    security: s1
spec:
  containers:
  - name: bear
    image: supergiantkir/animals:bear
```

As you see, we have assigned a label `security:s1` to this Pod. It will be referred to in the anti-affinity rule later. Now, save this spec to `pod-test1.yaml` and run the following command:

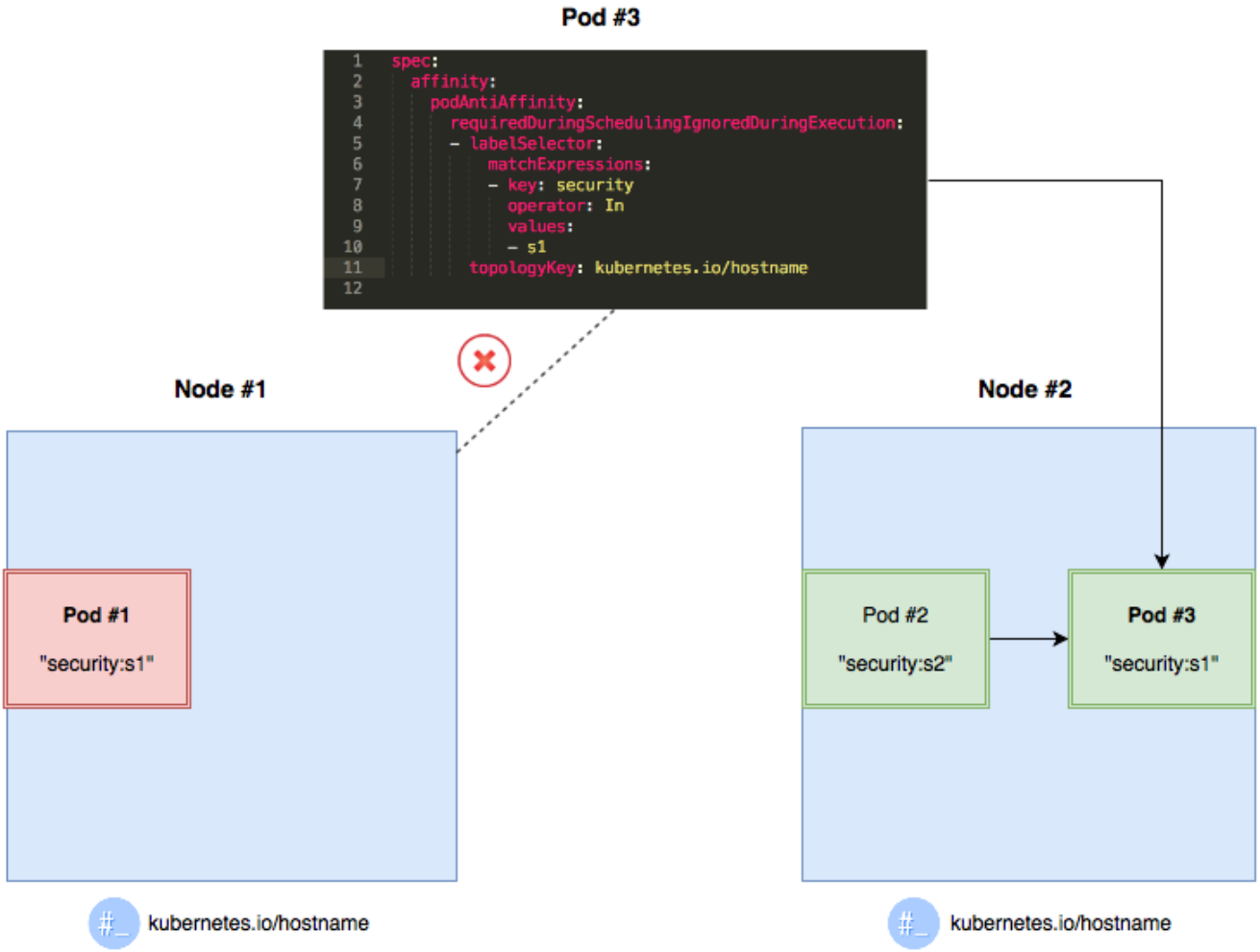
```
kubectl create -f pod-test1.yamlpod "s1" created
```

Next, let's create the second Pod with the Pod anti-affinity rule:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-s2
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
```

```
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - s1
        topologyKey: kubernetes.io/hostname
  containers:
  - name: pod-antiaffinity
    image: supergiantkir/animals:hare
```

As you see, we specified the `labelSelector` “security:s1” under `spec.affinity.podAntiAffinity` . That’s why the second Pod won’t be scheduled to any node that has a Pod with a label `security:s1` running on it (see the image below).



Let’s verify this by creating the second Pod. Save this spec to `anti-affinity-pod.yaml` and run the following command:

```
kubectl create -f anti-affinity-pod.yamlpod “pod-s2” created
```

If you now run `kubectl get pods` , you’ll see that the second Pod is in the Pending state.

```
kubectl get podsNAME                                  READY     STATUS
RESTARTS   AGEpod-s2                                0/1       Pending
0          5m
```

You can verify why this has happened by running `kubectl describe pod pod-s2` :

```
Events:Type     Reason            Age               From
Message----     ------            ----              ----
-------Warning  FailedScheduling  2s (x22 over 5m)  default-scheduler  0
/1 nodes are available: 1 MatchInterPodAffinity, 1
PodAntiAffinityRulesNotMatch.
```

In the events section above, you can see that the Pod was not scheduled due to `PodAntiAffinityRulesNotMatch` exception. That's because the only node we have on Minikube has the first Pod that does not match the anti-affinity rules of the `pod-s2`. Therefore, it was not scheduled.

Configuring Pod Affinity

You can use the same spec with a slight modification to implement Pod Affinity. The first Pod should look identical to the first Pod from the previous example.

The second Pod, however, should configure `podAffinity` field instead of the `podAntiAffinity` under the `spec.affinity` . For example:

```
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - s1
```

Now, when deployed, the second Pod will be scheduled to the node where the first Pod runs because the `podAffinity` `labelSelector` matches the labels of the first Pod against which the affinity rule is checked.

Using Affinity with High-Level Collections (Note: you need a multi-node cluster for this example)

In the examples above, we defined Pod affinity and anti-affinity rules for individual Pods. However, Kubernetes allows specifying these rules for a set of Pods managed by high-level abstractions such as Deployments or `StatefulSets`. In this case, Kubernetes will manage affinity or anti-affinity for a set of Pods created by the Deployment or a `StatefulSet`. Let's demonstrate how this works:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: memcached
```

```
spec:
  selector:
    matchLabels:
      app: store
  replicas: 3
  template:
    metadata:
      labels:
        app: store
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - store
            topologyKey: "kubernetes.io/hostname"
      containers:
      - name: memcached
        image: memcached:1.5.12
```

The manifest above defines a Deployment for the Memcached in-memory key-value store used to store small chunks of data from results of database calls, API calls, or page rendering. We're planning to launch 3 replicas of the Memcached in our cluster. To prevent them from ending up on the same node, we use the `podAntiAffinity` rule with a `labelSelector` that matches the label `app:store` of the Deployment's PodSpec. This will allow us to evenly spread the Memcached Pods across the cluster.

Save this spec to the `memcached.yaml` and run the following command:

```
kubectl create -f memcached.yaml
```

Next, we define the Deployment running three replicas of Apache HTTPD web server. We want our servers to be co-located with Memcached, so that they could access the stores faster avoiding high network latency. To achieve this, we are using a set of Pod affinity and anti-affinity rules.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
spec:
  selector:
    matchLabels:
      app: web-store
  replicas: 3
  template:
    metadata:
      labels:
        app: web-store
```

```
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - web-store
            topologyKey: "kubernetes.io/hostname"
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - store
            topologyKey: "kubernetes.io/hostname"
      containers:
      - name: web-app
        image: nginx:1.12-alpine
```

`podAntiAffinity` rule of this spec says that we don't want the replicas of the Apache HTTPD server to run on the same node. We use the web server's Pod label `app:web-store` to achieve this.

In its turn, the `podAffinity` rule says that we want our web server replicas to be placed on the same nodes where the Memcached Pods are running.

Now, let's save this spec to `httpd.yaml` and run:

```
kubectl create -f httpd.yaml
```

If you now check the Pod running on each node of your cluster, you'll probably see that each web server instance is co-located on the same Node as the Memcached instance. This way we achieved tight co-location between in-memory store and web servers.

Conclusion

That's it! In this tutorial, you learned how to use nodeSelector, node, and Pod affinity and anti-affinity rules to control how Pods are scheduled onto nodes. Pod scheduling rules are very useful for a number of use cases such as spreading your application instances evenly across availability zones, controlling resource usage, managing dedicated nodes and enabling tightly coupled applications and services. In a subsequent tutorial, we'll go a little more in-depth and study how you can combine node and Pod affinity with taints and tolerations used to repel Pods from specific nodes. Stay tuned to our blog to learn more!