**Labels:**

1. Labels are the mechanism you use to organize the Kubernetes objects
2. Labels are key value paired without any predefined meaning that can be attach to objects
3. Labels are similar to tag in AWS and GIT
4. Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system
5. Labels can be attached to objects at creation time and subsequently added and modified at any time.
6. You are free to choose labels as you need it to refers an environment which is used for dev or Testing or production, refer a product group like Deployment A, Deployment B .
7. Valid label value:

• must be 63 characters or less (can be empty),

• unless empty, must begin and end with an alphanumeric character ([a-z0-9A-Z]),

• could contain dashes (-), underscores (\_), dots (.), and alphanumeric between.

1. Apply label on pod via imperative method a. Kubectl label pod{object} object\_name env=dev

**Example of Declarative :**

apiVersion: v1

kind: Pod

metadata:

name: label-demo

labels:

environment: production

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

**Command:**

a. Kubectl get pods{object} --show-labels

b. Kubectl get pods{object} -l env=dev

c. Kubectl get pods{object} -l env!=dev

**Note:** there is 3 way to delete an object

a. From yaml file

b. Kubectl delete object object\_name

c. Kubectl delete object -l env=dev

**Selectors:**

1. Unlike names and UIDs, labels do not provide uniqueness. In general, we expect many objects to carry the same label(s).
2. Via a label selector, the client/user can identify a set of objects. The label selector is the core grouping primitive in Kubernetes.
3. The API currently supports two types of selectors

a. equality-based (=, !=)

b. Set-based (in, notin, exists)

**Command of set-based:** Kubectl get pods{object} -l 'env in (dev,test,qa)'

1. You can constrain a Pod so that it can only run on particular set of node(s). There are several ways to do this and the recommended approaches all use label selectors to facilitate the selection.
2. Generally such constraints are unnecessary, as the scheduler will automatically do a reasonable placement (for example, spreading your Pods across nodes so as not place Pods on a node with insufficient free resources).
3. However, there are some circumstances where you may want to control which node the Pod deploys to, for example, to ensure that a Pod ends up on a node with an SSD attached to it, or to co-locate Pods from two different services that communicate a lot into the same availability zone.
4. You can use any of the following methods to choose where Kubernetes schedules specific Pods: a. Node-Selector b. Affinity and Anti-Affinity c. Nodename.

**NodeSelector:**

1. nodeSelector is the simplest recommended form of node selection constraint.
2. You can add the nodeSelector field to your Pod specification and specify the node labels you want the target node to have. Kubernetes only schedules the Pod onto nodes that have each of the labels you specify.

**Example:**

apiVersion: v1

kind: Pod

metadata:

name: nginx

spec:

containers:

- name: nginx

image: nginx

nodeSelector:

nodename: minikubenod

**Affinity and anti-affinity**

nodeSelector is the simplest way to constrain Pods to nodes with specific labels. Affinity and anti-affinity expands the types of constraints you can define. Some of the benefits of affinity and anti-affinity include:

* The affinity/anti-affinity language is more expressive. nodeSelector only selects nodes with all the specified labels. Affinity/anti-affinity gives you more control over the selection logic.
* You can indicate that a rule is soft or preferred, so that the scheduler still schedules the Pod even if it can't find a matching node.
* You can constrain a Pod using labels on other Pods running on the node (or other topological domain), instead of just node labels, which allows you to define rules for which Pods can be co-located on a node.
* The affinity feature consists of two types of affinity:
  + Node affinity functions like the nodeSelector field but is more expressive and allows you to specify soft rules.
  + Inter-pod affinity/anti-affinity allows you to constrain Pods against labels on other Pods
* There are two types of node affinity:
  + **requiredDuringSchedulingIgnoredDuringExecution**: The scheduler can't schedule the Pod unless the rule is met. This functions like nodeSelector, but with a more expressive syntax.
  + **preferredDuringSchedulingIgnoredDuringExecution**: The scheduler tries to find a node that meets the rule. If a matching node is not available, the scheduler still schedules the Pod