NameSpace & Scheduling in Kubernetes

NameSpaces

In Kubernetes, *namespaces* provide a mechanism for isolating groups of resources within a single cluster. Names of resources need to be unique within a namespace, but not across namespaces. Namespace-based scoping is applicable only for namespaced objects *(e.g. Deployments, Services, etc.)* and not for cluster-wide objects *(e.g. StorageClass, Nodes, PersistentVolumes, etc.)*.

Let's check default NameSpaces

```
root@master-node:~# kubectl get namespaces
NAME              STATUS   AGE
default           Active   13d
kube-node-lease   Active   13d
kube-public       Active   13d
kube-system       Active   13d
root@master-node:~#
```

```
root@master-node:~# kubectl get pods -n kube-system
NAME                                     READY   STATUS    RESTARTS         AGE
coredns-55cb58b774-h2xxl                 1/1     Running   1 (4m39s ago)    3d
coredns-55cb58b774-rkdnc                 1/1     Running   2 (4m39s ago)    3d20h
etcd-master-node.com                     1/1     Running   6 (4m39s ago)    13d
kube-apiserver-master-node.com           1/1     Running   6 (4m39s ago)    13d
kube-controller-manager-master-node.com  1/1     Running   6 (4m39s ago)    13d
kube-proxy-9s99k                         1/1     Running   7 (4m37s ago)    13d
kube-proxy-cj6kv                         1/1     Running   6 (4m39s ago)    13d
kube-proxy-xgwwc                         1/1     Running   7 (4m37s ago)    13d
kube-scheduler-master-node.com           1/1     Running   6 (4m39s ago)    13d
weave-net-hhfds                          2/2     Running   13 (4m39s ago)   13d
weave-net-pjkvm                          2/2     Running   16 (4m37s ago)   13d
weave-net-wqhbh                          2/2     Running   15 (4m37s ago)   13d
root@master-node:~#
```

Cmd to check current namespace in which we working right now

```
# kubectl config get-contexts
```

```
root@master-node:~#  kubectl config get-contexts
CURRENT   NAME                          CLUSTER      AUTHINFO           NAMESPACE
*         kubernetes-admin@kubernetes   kubernetes   kubernetes-admin
root@master-node:~#
```

In NAMESPACE section it is showing blank it means right now we working in default NS.

Cmd to change NS

kubectl config set-context $(kubectl config current-context) --namespace kube-system

```
root@master-node:~# kubectl config set-context $(kubectl config current-context) --namesp
ace kube-system
Context "kubernetes-admin@kubernetes" modified.
root@master-node:~#
root@master-node:~# kubectl config get-contexts
CURRENT   NAME                          CLUSTER      AUTHINFO           NAMESPACE
*         kubernetes-admin@kubernetes   kubernetes   kubernetes-admin   kube-system
root@master-node:~#
```

Let's create a NS

kubectl create ns prod

```
root@master-node:~# kubectl create ns prod
namespace/prod created
root@master-node:~#
root@master-node:~# kubectl get ns
NAME              STATUS   AGE
default           Active   13d
kube-node-lease   Active   13d
kube-public       Active   13d
kube-system       Active   13d
prod              Active   7s
root@master-node:~#
```

```
root@master-node:~# kubectl get pods
No resources found in default namespace.
root@master-node:~#
root@master-node:~# kubectl run webserver --image httpd -n prod
pod/webserver created
root@master-node:~#
root@master-node:~# kubectl get pods
No resources found in default namespace.
root@master-node:~#
root@master-node:~# kubectl get pods -n prod
NAME        READY   STATUS    RESTARTS   AGE
webserver   1/1     Running   0          10s
root@master-node:~#
```

As we can see in this image we create a pod in prod NS. And get pod info we need to specify the NS name bcoz right now we are working in default NS.

We can define NS in file. Let's create a pod using file method

vim pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
        name: mypod
        namespace: prod
        labels:
                app: wordpress
                env: prod
spec:
        containers:
                - name: test
                  image: nginx
                  ports:
                        - containerPort: 80
```

```
root@master-node:~# kubectl apply -f pod.yml
pod/mypod created
root@master-node:~# kubectl get pods -n prod
NAME    READY   STATUS    RESTARTS   AGE
mypod   1/1     Running   0          21s
root@master-node:~#
```

```
root@master-node:~# kubectl get pods -n prod
NAME    READY   STATUS    RESTARTS   AGE
mypod   1/1     Running   0          4m42s
root@master-node:~#
root@master-node:~# kubectl delete pod mypod -n prod
pod "mypod" deleted
root@master-node:~#
```

Scheduling

Kubernetes scheduling is a process that assigns Pods to available Nodes. When you create a Pod, the Kubernetes Scheduler decides which Node should run that Pod.

How Scheduler work

Pod is Created

When you run `kubectl apply -f pod.yaml`, an unscheduled Pod is created.
Its nodeName field remains empty.

Scheduler Evaluates Nodes

The scheduler checks available Nodes based on resources (CPU, memory, storage, taints, affinity rules).

Best Node is Chosen

The scheduler uses Filtering and Scoring algorithms.
After selecting the best Node, the Pod is bound to it.

Pod is Deployed on the Node

The Kubelet pulls and starts the Pod on the assigned Node.

What if the scheduler is not working to schedule pods to best fit node.

In this kind of situation we can define manual node where pods will be created.

Let's perform this task.

vim pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
        name: mypod
        labels:
              app: wordpress
              env: prod
spec:
        containers:
                - name: test
                  image: nginx
                  ports:
                      - containerPort: 80
        nodeName: worker2-node.com
```

```
root@master-node:~# kubectl apply -f pod.yml
pod/mypod created
root@master-node:~# kubectl get pods -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP          NODE              NOMINATED NODE
  READINESS GATES
mypod   1/1     Running   0          10s   10.32.0.2   worker2-node.com   <none>
  <none>
root@master-node:~# vim pod.yml
root@master-node:~# root@master-node:~#
```

kubectl describe pods

```
                          node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason   Age   From     Message
  ----    ------   ----  ----     -------
  Normal  Pulling  87s   kubelet  Pulling image "nginx"
  Normal  Pulled   87s   kubelet  Successfully pulled image "nginx" in 131ms (131ms inclu
ding waiting). Image size: 72188133 bytes.
  Normal  Created  87s   kubelet  Created container test
  Normal  Started  87s   kubelet  Started container test
```

Here we can see that there is no event logs from scheduler. Bcoz we bypass it.

Otherwise by default is look like this.

```
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  4s    default-scheduler  Successfully assigned default/mypod to work
er2-node.com
  Normal  Pulling    3s    kubelet            Pulling image "nginx"
  Normal  Pulled     3s    kubelet            Successfully pulled image "nginx" in 108ms
(108ms including waiting). Image size: 72188133 bytes.
  Normal  Created    3s    kubelet            Created container test
  Normal  Started    3s    kubelet            Started container test
```

Taint & Tolerations

In Kubernetes, **Taints & Tolerations** are used to ensure that only specific Pods can be scheduled on certain Nodes.

- **Taint (Applied to Nodes)** → Restricts Nodes so that only allowed Pods can be scheduled on them.
- **Toleration (Applied to Pods)** → Allows Pods to run on tainted Nodes.

Taint → Restricts a Node.
Toleration → Allows a Pod to run on a tainted Node.

Scenario: Scheduling Pods on High CPU & High RAM Nodes

You have 2 Nodes:

- node2 has **high RAM**.
- node1 has **high CPU**.

Objective

- High CPU usage Pods should only run on **node1.**
- High RAM usage Pods should only run on **node2**.

To achieve this, we will use **Taints & Tolerations**.

```
# kubectl describe worker2-node.com
```

```
CreationTimestamp:  Mon, 10 Feb 2025 18:37:18 +0000
Taints:             <none>
Unschedulable:      false
Lease:
```

As we can see at this point no taints added to node1 & 2

Before adding tolerations in the pod YAML file, let's create 10 pods using the deployment method and check how many pods are created on each node.

```
# kubectl apply -f deployment.yaml
```

```
root@master-node:~# kubectl apply -f deployment.yaml
deployment.apps/mywebsite created
root@master-node:~#
root@master-node:~# kubectl get pods -o wide
NAME                         READY   STATUS    RESTARTS   AGE   IP          NODE                NOMI
NATED NODE   READINESS GATES
mywebsite-fb9b5bd94-5b7tx    1/1     Running   0          11s   10.46.0.4   worker1-node.com    <non
e>           <none>
mywebsite-fb9b5bd94-62f2c    1/1     Running   0          11s   10.32.0.5   worker2-node.com    <non
e>           <none>
mywebsite-fb9b5bd94-7mvvs    1/1     Running   0          11s   10.32.0.4   worker2-node.com    <non
e>           <none>
mywebsite-fb9b5bd94-995g8    1/1     Running   0          11s   10.46.0.1   worker1-node.com    <non
e>           <none>
mywebsite-fb9b5bd94-dl6nd    1/1     Running   0          11s   10.32.0.6   worker2-node.com    <non
e>           <none>
mywebsite-fb9b5bd94-n6stq    1/1     Running   0          11s   10.46.0.5   worker1-node.com    <non
e>           <none>
mywebsite-fb9b5bd94-ndp8p    1/1     Running   0          11s   10.46.0.3   worker1-node.com    <non
e>           <none>
mywebsite-fb9b5bd94-tddxd    1/1     Running   0          11s   10.46.0.2   worker1-node.com    <non
e>           <none>
mywebsite-fb9b5bd94-tg52j    1/1     Running   0          11s   10.32.0.3   worker2-node.com    <non
```

As we can pods are created on both nodes.

Step 1: Apply Taints to Nodes

First, apply **taints** on nodes to restrict scheduling to only matching Pods.

```
kubectl taint nodes worker2-node.com key=value:effect
```

Here we 3 type of effects

1. NoSchedule
2. PreferNoShedule
3. NoExecute

Apply Taint on High CPU Node (node2)

```
kubectl taint nodes worker1-node.com type=high-cpu:NoSchedule
```

Now, only Pods with a matching **toleration** will be scheduled on node1.

Let's check .

```
kubectl apply -f deployment.yaml

kubectl get pods -o wide
```

```
root@master-node:~# kubectl get pods -o wide
NAME                         READY   STATUS    RESTARTS   AGE   IP           NODE               NOM
INATED NODE   READINESS GATES
mywebsite-fb9b5bd94-28rbs    1/1     Running   0          23s   10.32.0.3    worker2-node.com   <no
ne>           <none>
mywebsite-fb9b5bd94-4rgtb    1/1     Running   0          23s   10.32.0.6    worker2-node.com   <no
ne>           <none>
mywebsite-fb9b5bd94-67z2q    1/1     Running   0          23s   10.32.0.7    worker2-node.com   <no
ne>           <none>
mywebsite-fb9b5bd94-7l544    1/1     Running   0          23s   10.32.0.8    worker2-node.com   <no
ne>           <none>
mywebsite-fb9b5bd94-997ns    1/1     Running   0          23s   10.32.0.11   worker2-node.com   <no
ne>           <none>
mywebsite-fb9b5bd94-b557c    1/1     Running   0          23s   10.32.0.5    worker2-node.com   <no
ne>           <none>
mywebsite-fb9b5bd94-b7w7j    1/1     Running   0          23s   10.32.0.10   worker2-node.com   <no
ne>           <none>
mywebsite-fb9b5bd94-kztcp    1/1     Running   0          23s   10.32.0.9    worker2-node.com   <no
ne>           <none>
mywebsite-fb9b5bd94-pw889    1/1     Running   0          23s   10.32.0.2    worker2-node.com   <no
ne>           <none>
mywebsite-fb9b5bd94-wbgb5    1/1     Running   0          23s   10.32.0.4    worker2-node.com   <no
ne>           <none>
```

Here, we can see that all the pods are created only on **worker-node2**.

Apply Taint on High RAM Nodes (node2)

```
kubectl taint nodes worker2-node.com type=high-ram:NoSchedule
```

Now, only Pods with a **high-ram** toleration will be scheduled on node1.

Step 2: Add Tolerations to Pods

Next, we add **tolerations** in the Pod YAML files to allow them to be scheduled on the correct nodes.

Pod for High CPU Usage (Will Deploy on node1)

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: mywebsite
spec:
 selector:
  matchLabels:
   app: webserver
 replicas: 10
 template:
  metadata:
   name: xyz
   labels:
    app: webserver
  spec:
   containers:
    - name: web
      image: nginx
   tolerations:
    - key: "type"
      operator: "Equal"
      value: "high-cpu"
      effect: "NoSchedule"
```

This Pod will only be scheduled on node1 because it has the **high-cpu** toleration.

```
# kubectl apply -f deployment.yaml
```

```
root@master-node:~# kubectl get pods -o wide
NAME                         READY   STATUS    RESTARTS   AGE   IP          NODE              NOMINATED
 NODE   READINESS GATES
mywebsite-66b86c97d6-4xdhz   1/1     Running   0          3s    10.32.0.2   worker2-node.com   <none>
        <none>
mywebsite-66b86c97d6-5mnx7   1/1     Running   0          3s    10.32.0.4   worker2-node.com   <none>
        <none>
mywebsite-66b86c97d6-8wx7z   1/1     Running   0          3s    10.46.0.2   worker1-node.com   <none>
        <none>
mywebsite-66b86c97d6-j9kjw   1/1     Running   0          3s    10.32.0.3   worker2-node.com   <none>
        <none>
mywebsite-66b86c97d6-wjgcc   1/1     Running   0          3s    10.46.0.1   worker1-node.com   <none>
        <none>
root@master-node:~#
```

According to this, all pods should have been created on **worker-node1**, but as we can see, they were created on both **node1 and node2**.
Let's understand why this happened.

The reason for this behavior is Kubernetes' scheduler behavior. Let's understand:

Effect of Taint:

- You applied a **high-cpu** taint on **worker-node1**.
- **Worker-node2** has no taint, meaning pods can be scheduled there without any restrictions.

Effect of Tolerations:

- You added **tolerations** in the deployment file, allowing pods to be scheduled on **worker-node1** as well.
- However, **tolerations only provide permission**; they do not guarantee that the pod will be scheduled on that node.

Scheduler's Decision:

- The **Kubernetes scheduler** considers both **worker-node1** and **worker-node2** and makes a decision based on **resource availability, load balancing, and affinity rules**.
- Since **worker-node2** has no taint, the scheduler will first consider it if resources are available.
- **Worker-node1** has a taint, but the pods can tolerate it, so some pods get scheduled there.

End Result:

- **2 pods** are scheduled on **worker-node1**.
- **3 pods** are scheduled on **worker-node2**.

So the solution is **NodeSelector**

NodeSelector

NodeSelector can be a solution, but it is a **hard constraint**, meaning pods will only be scheduled on nodes that match the specified label.

If you want the pods to be scheduled **only and only on worker-node1**, then you can use **nodeSelector**.

Let's do a practical implementation of **nodeSelector** step by step and schedule pods on **worker-node1**.

Add a lebal on **worker-node1**.

Remove taint from worker1-node.com

```
# kubectl taint nodes worker1-node.com type=high-cpu:NoSchedule-

# kubectl label nodes worker1-node.com app=webserver
```

```
root@master-node:~# kubectl get nodes worker1-node.com --show-labels
NAME               STATUS   ROLES    AGE   VERSION   LABELS
worker1-node.com   Ready    <none>   14d   v1.30.9   app=webserver,beta.kubernetes.io/arc
h=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,kubernetes.io/hostname=worke
r1-node.com,kubernetes.io/os=linux
root@master-node:~#
```

```
vim deployment.yaml
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mywebsite
spec:
  selector:
    matchLabels:
      app: webserver
  replicas: 5
  template:
    metadata:
      name: xyz
      labels:
        app: webserver
    spec:
      containers:
        - name: web
          image: nginx
      nodeSelector:
           app: webserver
```

kubectl get deployments.apps

```
root@master-node:~# kubectl get deployments.apps
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
mywebsite   5/5     5            5           10s
root@master-node:~# kubectl get pods -o wide
NAME                         READY   STATUS    RESTARTS   AGE   IP          NODE
      NOMINATED NODE   READINESS GATES
mywebsite-85677875cc-2dmwb   1/1     Running   0          46s   10.46.0.3   worker1-node.
com   <none>           <none>
mywebsite-85677875cc-5tp8c   1/1     Running   0          46s   10.46.0.2   worker1-node.
com   <none>           <none>
mywebsite-85677875cc-gmwvt   1/1     Running   0          46s   10.46.0.1   worker1-node.
com   <none>           <none>
mywebsite-85677875cc-gq9lt   1/1     Running   0          46s   10.46.0.4   worker1-node.
com   <none>           <none>
mywebsite-85677875cc-h9t42   1/1     Running   0          46s   10.46.0.5   worker1-node.
com   <none>           <none>
root@master-node:~#
```

All pods are runing on worker-node1

Daemon-set

What is a DaemonSet in Kubernetes?

A **DaemonSet** in Kubernetes ensures that a **copy of a pod runs on every node** (or on a selected set of nodes). It is mainly used for **node-specific tasks** such as monitoring, logging, or networking.

Key Features of DaemonSet:

Ensures that a pod is running on **every node** in the cluster.
Automatically adds the pod to **newly added nodes** in the cluster.
Removes the pod when a node is **removed from the cluster**.
Can be restricted to **specific nodes** using **nodeSelector, nodeAffinity, or tolerations**.

Use Cases of DaemonSet:

1. **Log Collection** – Running Fluentd, Filebeat, or another logging agent on every node.
2. **Monitoring** – Deploying Prometheus Node Exporter or any monitoring agent.
3. **Networking** – Running CNI plugins like Calico, Flannel, or Cilium.
4. **Security Agents** – Running security monitoring tools like Falco on all nodes.

```
# vim daemon-set.yaml

apiVersion: apps/v1

kind: DaemonSet

metadata:

 name: logcollector

spec:

 selector:
```

```
  matchLabels:
    app: logserver
  template:
    metadata:
      name: kuchbhi
      labels:
        app: logserver
    spec:
      containers:
        - name: lkjsdf
          image: nginx
```

```
#  kubectl apply -f daemon-set.yaml
```

```
root@master-node:~# kubectl apply -f daemon-set.yaml
daemonset.apps/logcollector created
root@master-node:~#
root@master-node:~# kubectl get daemonsets.apps
NAME           DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
logcollector   2         2         2       2            2           <none>          6s
root@master-node:~#
root@master-node:~# kubectl get pods -o wide
NAME                 READY   STATUS    RESTARTS   AGE   IP          NODE                NO
MINATED NODE   READINESS GATES
logcollector-9p4gh   1/1     Running   0          21s   10.46.0.1   worker1-node.com    <n
one>           <none>
logcollector-h5nxk   1/1     Running   0          21s   10.32.0.2   worker2-node.com    <n
one>           <none>
root@master-node:~#
```

Create multi-container in a single pod

```
vim podx.yml

apiVersion: v1

kind: Pod

metadata:
```

```
    name: multicontainer
    labels:
      app: multiapp
spec:
    containers:
      - name: abc
        image: nginx

      - name: xyz
        image: httpd
```

```
kubectl apply -f podx.yml
```

```
root@master-node:~# kubectl get pods
NAME                 READY   STATUS             RESTARTS      AGE
logcollector-9p4gh   1/1     Running            0             46m
logcollector-h5nxk   1/1     Running            0             46m
multicontainer       1/2     CrashLoopBackOff   1 (4s ago)    8s
```

Resource Quotas

When several users or teams share a cluster with a fixed number of nodes, there is a concern that one team could use more than its fair share of resources.

Resource quotas are a tool for administrators to address this concern.

A resource quota, defined by a `ResourceQuota` object, provides constraints that limit aggregate resource consumption per namespace. It can limit the quantity of objects that can be created in a namespace by type, as well as the total amount of compute resources that may be consumed by resources in that namespace.

Resource quotas work like this:

- Different teams work in different namespaces. This can be enforced with RBAC.

- The administrator creates one ResourceQuota for each namespace.
- Users create resources (pods, services, etc.) in the namespace, and the quota system tracks usage to ensure it does not exceed hard resource limits defined in a ResourceQuota.
- If creating or updating a resource violates a quota constraint, the request will fail with HTTP status code `403 FORBIDDEN` with a message explaining the constraint that would have been violated.
- If quotas are enabled in a namespace for compute resources like `cpu` and `memory`, users must specify requests or limits for those values; otherwise, the quota system may reject pod creation. Hint: Use the `LimitRanger` admission controller to force defaults for pods that make no compute resource requirements.

Assign Memory Resources to Containers & Pods

vim pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
        name: mypod
        labels:
              app: wordpress
              env: prod
spec:
        containers:
                - name: test
                  image: nginx
                  resources:
                   requests:
                     memory: "100Mi"
                   limits:
                     memory: "200Mi"
```