**Agenda: Deep Dive into Pods and Containers**

* Resource Limits
* Limit Ranges
* Configure Memory and CPU Quotas for a Namespace
* Manual and Auto Scaling
* Termination Grace Period
* Init Containers
* Working with Probes

**Understanding Resource Limits**

* When you specify a [Pod](https://kubernetes.io/docs/concepts/workloads/pods/), you can optionally specify how much of each resource a [container](https://kubernetes.io/docs/concepts/containers/) needs.
* The most common resources to specify are CPU and memory (RAM);
* When you specify the resource **request** for containers in a Pod, the **kube-scheduler** uses this information to decide which node to place the Pod on.
* When you specify a resource **limit** for a container, the **kubelet** enforces those limits so that the running container is not allowed to use more of that resource than the limit you set.

**CPU Units**

* The unit suffix **m** stands for **millicore**/**millicpu** (“thousandth of a core”).
* 1 CPU unit is equivalent to **1 physical CPU core**, or **1 virtual core = 1000 millicore**
* Likewise **2000m** would be two full cores, which can also be specified as **2** or **2.0**

**Memory Units:**

When looking at the metrics, you get the memory in typical **Mi** notation, meaning **mebibytes**

apiVersion: v1

kind: Pod

metadata:

 name: nginx-pod

 labels:

   app: web

spec:

 containers:

  - name: nginx-con

    image: nginx

    ports:

      - containerPort: 80

    resources:

**requests:**

        memory: 50Mi

        cpu: 50m #or 0.05 or 5%

**limits:**

        memory: 100Mi

        cpu: 100m #or 0.1 or 10%

**In example above:** Resources object specifies that the container process needs 50/1000 of a core (5%) and is allowed to use at most 100/1000 of a core (10%).

**Demo:** Change Request Memory to 8Gi and Limit Memory to 10Gi and note that Pod remains in **Pending** state.

**Quality of Service (QoS)** in Kubernetes refers to the categorization of Pods based on how the resources (CPU and memory) are requested and limited. QoS ensures that Kubernetes manages Pods more effectively during resource contention or overcommitment scenarios, meaning when the cluster is under heavy load or low on resources.

* **Guaranteed** (requests and limits values are same): Pods have the **highest priority** and are the last to be evicted when a node is under resource pressure.
* **Burstable** (requests are less than limits): Pods are next in line; they can use more resources than requested if available, but they may be evicted if resource pressure arises.
* **BestEffort** (requests and limits are not specified): Pods are the most flexible but also the least guaranteed. They are the **first to be evicted** and only get leftover resources.

Limit Ranges

By default, containers run with unbounded compute resources on a Kubernetes cluster. With resource quotas, cluster administrators can restrict resource consumption and creation on a namespace basis.

Within a namespace, a Pod or Container can consume as much CPU and memory as defined by the **namespace's resource quota**. There is a concern that one Pod or Container could monopolize all available resources. **A LimitRange is a policy to constrain resource allocations (to Pods or Containers) in a namespace.**

**Default Limits and resources for POD not specifying the configuration for its containers:**

To establish default limits you create the LimitRange object in the namespace you want them to apply to.

Here’s an example:

apiVersion: v1

kind: LimitRange

metadata:

  name: default-limit

namespace: n1

spec:

  limits:

  - **default**: #limits default

      memory: 100Mi

      cpu: 100m

**defaultRequest**: #requests default

      memory: 50Mi

      cpu: 50m

**max**:

      memory: 512Mi

      cpu: 500m

**min**:

      memory: 50Mi

      cpu: 50m

**type: Container**

---

apiVersion: v1

kind: Pod

metadata:

 name: nginx-pod

 labels:

   app: web

spec:

 containers:

  - name: nginx-con

    image: nginx

    ports:

      - containerPort: 80

The **default** key under **limits** represents the default limits for each resource.

The **defaultRequest** key is for resource requests.

Note:

* If a LimitRange is **activated in a namespace** for compute resources like cpu and memory, users must specify requests or limits for those values. Otherwise, the system may reject Pod creation.
* LimitRange validations occurs only at Pod Admission stage, not on Running Pods.

Configure Resource Quotas for a Namespace

ResourceQuota object can be used to set quotas for the **total amount** memory and CPU that can be used by all Pods running in a [namespace](https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces). Also it can restrict the number of objects, of a particular like, that can be created in a namedpace.

* The administrator creates one ResourceQuota for each namespace.
* 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.

apiVersion: v1

kind: ResourceQuota

metadata:

  name: mem-cpu-demo

  namespace: n1

spec:

**hard**:

    requests.cpu: "1"

    requests.memory: 10Gi

    limits.cpu: "2"

    limits.memory: 20Gi

    persistentvolumeclaims: "1"

    services.loadbalancers: "2"

    services.nodeports: "0"

* For every Pod in the namespace, each container must have a memory request, memory limit, cpu request, and cpu limit.
* The memory request total for all Pods in that namespace must not exceed 1 GiB.
* The memory limit total for all Pods in that namespace must not exceed 2 GiB.
* The CPU request total for all Pods in that namespace must not exceed 1 cpu.
* The CPU limit total for all Pods in that namespace must not exceed 2 cpu.

kubectl get **resourcequota** mem-cpu-demo --namespace=n1 --output=yaml

The output shows the quota along with how much of the quota has been used. You can see that the memory and CPU requests and limits for your Pod do not exceed the quota.

status:

hard:

limits.cpu: "2"

limits.memory: 2Gi

requests.cpu: "1"

requests.memory: 1Gi

persistentvolumeclaims: "1"

services.loadbalancers: "2"

services.nodeports: "0"

**used:**

**limits.cpu: 800m**

**limits.memory: 800Mi**

**requests.cpu: 400m**

**requests.memory: 600Mi**

persistentvolumeclaims: "0"

services.loadbalancers: "0"

services.nodeports: "0"

**Monitoring Cluster for Memory and CPU**

**Metrics Server offers:**

* A single deployment that works on most clusters (see [Requirements](https://github.com/kubernetes-sigs/metrics-server#requirements))
* Fast autoscaling, collecting metrics every 15 seconds.
* Resource efficiency, using 1 mili core of CPU and 2 MB of memory for each node in a cluster.
* Scalable support up to 5,000 node clusters

**To install the Metrics Server:**

**For Docker Desktop:**

**1. Download and save the below file locally:**

https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml

#Add the below line to metrics server's container args, around line 135

- --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname

**- --kubelet-insecure-tls=true**

2. Deploy the Metric Server

kubectl apply -f <https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml>

3. Verify that the Metric Server is started successfully

kubectl get pods -n kube-system

**Note: If you get error, use the YAML provided in the below (its older version 4.5 but works)**

<https://stackoverflow.com/questions/72239288/kubernetes-metrics-server-not-starting-up-locally>

**For MiniKube:**

minikube addons enable metrics-server

**To Verify if metric server has started**

kubectl get pods -n kube-sysem

**# To get the memory and CPU usage.**

kubectl top pods

kubectl top nodes

kubectl top pods --sort-by=cpu

kubectl top pods --sort-by=memory

#We can look at our system pods, CPU and Memory

kubectl top pods --all-namespaces

Note: The **kubectl top** command doesn’t actually collect any **metrics** itself. It queries the Metrics API for the metrics and presents them to you. In most clusters, especially those provided by cloud services, the Metrics API will already be installed.

Horizontal Scaling

**Scale pods manually**

kubectl **scale --replicas=5** deployment/hellowebapp-deployment

## Autoscale Pods

Kubernetes supports **horizontal pod autoscaling** to adjust the number of pods in a deployment depending on CPU utilization or other select metrics. The **Metrics Server** is used to provide resource utilization to Kubernetes.

To use the autoscaler, your pods must have CPU requests and limits defined.

apiVersion: apps/v1

kind: Deployment

metadata:

  name: nginx-deployment

spec:

  replicas: 5

  selector:

    matchLabels:

      app: nginx-app

  template:

    metadata:

      labels:

        app: nginx-app

    spec:

      containers:

      - name:  nginx-con

        image: nginx

        ports:

        - containerPort: 80

        imagePullPolicy: Always

        resources:

          limits:

            cpu: 500m

          requests:

            cpu: 200m

## Create Horizontal Pod Autoscaler (HPA)

Now that the server is running, we will create the autoscaler using kubectl autoscale.

The following example uses the [kubectl autoscale](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#autoscale) command to autoscale the number of pods in the deployment. If CPU utilization exceeds 80%, the autoscaler increases the pods up to a maximum of 10 instances:

kubectl **autoscale** deployment/nginx-deployment --cpu-percent=80 --min=3 --max=10

OR

apiVersion: autoscaling/v1

kind: **HorizontalPodAutoscaler**

metadata:

  name: nginx-hpa

spec:

  scaleTargetRef:

    apiVersion: apps/v1

    kind: Deployment

    name: nginx-deployment

  minReplicas: 3

  maxReplicas: 10

  targetCPUUtilizationPercentage: 80

To see the status of the autoscaler, use the kubectl get hpa command as follows:

$ **kubectl get hpa**

NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE

nginx-hpa deployment/nginx-deployment 10% / 80% 3 10 3 2m

In above sample 10% is Current CPU Usage and 80% is Desired CPU usage.

After a few minutes, with minimal load on the app, the number of pod replicas decreases automatically to three.

In new terminal, execute the following (repeat the same for **atleast two pods**)

* kubectl exec nginx-deployment-857485c896-gfplb -- sh

### **# dd if=/dev/urandom of=/dev/null**

* + dd: A Unix command used to copy and convert data.
  + if=/dev/urandom: Specifies the input file (if) as /dev/urandom, which generates random data.
  + of=/dev/null: Specifies the output file (of) as /dev/null, a special file that discards anything written to it.

In two more terminal window note the o/p of each of the following commands

* kubectl get pods --watch
* kubectl get hpa --watch

### **Algorithm details**

From the most basic perspective, the HorizontalPodAutoscaler controller operates on the ratio between desired metric value and current metric value:

**desiredReplicas** = ceil[currentReplicas \* ( currentMetricValue / desiredMetricValue )]

For example, if the current metric value is 200m, and the desired value is 100m, the number of replicas will be doubled, since 200.0 / 100.0 == 2.0 If the current value is instead 50m, you'll halve the number of replicas, since 50.0 / 100.0 == 0.5. The control plane skips any scaling action if the ratio is sufficiently close to 1.0 (within a globally-configurable tolerance, 0.1 by default).

**Manually Scalling of Nodes:**

**Scale Nodes Manually in Azure AKS Cluster:**

**CLI Command to scale**

az aks **scale** --name democluster --resource-group DemoRG **--node-count** **2**

**Azure Portal:**

Portal 🡪 DssDemoCluster Node pools 🡪 Scale node pool 🡪 Provide the number of nodes required.

**Cluster Autoscalar**

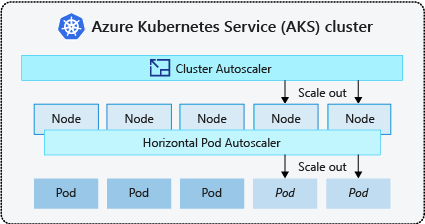
Cluster Autoscaler is a tool that automatically adjusts the size of the Kubernetes cluster when one of the following conditions is true:

* It increases the size of the cluster when there are pods that failed to run in the cluster due to insufficient resources.
* It decreases the size of the cluster when there are nodes in the cluster that have been underutilized for an extended period of time and their pods can be placed on other existing nodes.

More: <https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/README.md>

**Azure:**

The cluster autoscaler on Azure dynamically scales Kubernetes worker nodes. It runs as a deployment in your cluster.



**During Cluster Creation:**

az aks create \ --resource-group <ResourceGroupName> \ --name <AKSClusterName> \ --enable-cluster-autoscaler \ --min-count <MinNodeCount> \ --max-count <MaxNodeCount>

**For an Existing Cluster:**

az aks update \ --resource-group <ResourceGroupName> \ --name <AKSClusterName> \ --enable-cluster-autoscaler \ --min-count <MinNodeCount> \ --max-count <MaxNodeCount>

**To Monitor Cluster Autoscalar:**

kubectl logs -f deployment/cluster-autoscaler -n kube-system

**AWS:**

1. Create a IAM Policy that grants the necessary permissions to the cluster autoscalar

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"autoscaling:DescribeAutoScalingGroups",

"autoscaling:DescribeAutoScalingInstances",

"autoscaling:DescribeTags",

"autoscaling:SetDesiredCapacity",

"autoscaling:TerminateInstanceInAutoScalingGroup"

],

"Resource": "\*"

}

]

}

1. Create this policy using the AWS CLI:

aws iam create-policy \

--policy-name AmazonEKSClusterAutoscalerPolicy \

--policy-document <file://cluster-autoscaler-policy.json>

1. Attach the IAM Policy to the Worker Node IAM Role:

aws iam attach-role-policy \

--role-name <NodeInstanceRoleName> \ # Find the IAM role associated with your worker nodes

--policy-arn arn:aws:iam::<AccountID>:policy/AmazonEKSClusterAutoscalerPolicy

1. Install the Cluster Autoscaler in the EKS Cluster:

wget <https://raw.githubusercontent.com/kubernetes/autoscaler/master/cluster-autoscaler/cloudprovider/aws/examples/cluster-autoscaler-one-asg.yaml>

1. Update the manifest file with your EKS cluster name and other settings.
2. Apply the manifest to deploy the autoscaler:

kubectl apply -f cluster-autoscaler-one-asg.yaml

1. Annotate the Cluster Autoscaler Deployment:

kubectl annotate deployment cluster-autoscaler \

-n kube-system \

cluster-autoscaler.kubernetes.io/safe-to-evict="false"

1. Configure Auto-Discovery: Edit the Cluster Autoscaler deployment to enable auto-discovery of Auto Scaling Groups (ASGs). Ensure the following command line arguments are present:

- --cloud-provider=aws

- --nodes=<MinNodes>:<MaxNodes>:<ASGName>

- --auto-discover-node-group-tag=kubernetes.io/cluster/<ClusterName>

1. Check if the Cluster Autoscaler is running correctly by viewing the logs:

kubectl logs -f deployment/cluster-autoscaler -n kube-system

**Termination Grace Period**

**How it works:**

* When a Pod is being terminated (e.g., due to scaling down, deployment update, or node shutdown), Kubernetes sends a SIGTERM signal to all containers in the Pod.
* The containers should start the shutdown process (e.g., stopping services, releasing resources).
* Kubernetes waits for the duration of terminationGracePeriodSeconds.
* If the containers have not exited within the grace period, Kubernetes will send a SIGKILL signal to forcefully terminate them.

Default value = 30

apiVersion: v1

kind: Pod

metadata:

  name: init-container-demo

  labels:

    purpose: demo

spec:

**terminationGracePeriodSeconds**: 30

  containers:

  - name: nginx-container

    image: nginx

**OR**

kubectl delete pod **--grace-period=<seconds>**

**Force Deletion** - Immediately deletes records in API and etcd

kubectl delete pod --grace-period=0 **--force=True**

Note: Can only be set to 0 when --force is true (force deletion).

**Init Containers**

**About Init Containers**

* Runs before main application container.
* Init containers can contain utilities or setup scripts not present in an app image.
* Always Run to completion.
* Can have more than one per Pod.
* Each init container must complete successfully before the next one starts.
* All init containers must run to successful completion for the Pod to start.
* If a Pod's init container fails, the kubelet repeatedly restarts that init container until it succeeds. However, if the Pod has a **restartPolicy** of Never, and an init container fails during startup of that Pod, Kubernetes treats the overall Pod as failed.

**myapp.yaml**

apiVersion: v1

kind: Pod

metadata:

  name: myapp-pod

  labels:

    app: myapp

spec:

**containers**:

  - name: myapp-container

    image: nginx

**initContainers**:

  - name: init-myservice1

    image: ubuntu

    command: ['sh', '-c', "apt update; apt install dnsutils -y; until nslookup myservice1.default.svc.cluster.local; do echo waiting for myservice1; sleep 2; done"]

  - name: init-myservice2

    image: ubuntu

    command: ['sh', '-c', " apt update; apt install dnsutils -y; until nslookup myservice2.default.svc.cluster.local; do echo waiting for myservice2; sleep 2; done"]

Execute the following Commands:

kubectl apply -f myapp.yaml

kubectl get all

Note: Status shows **Init=0/2** and **Ready=0/1**

kubectl describe pods myapp-pod

Note: Look at how the init containers are created before the nginx pod.

To see logs for the init containers in this Pod, run:

kubectl logs myapp-pod -c init-myservice1 # Inspect the first init container

kubectl logs myapp-pod -c init-myservice2 # Inspect the first init container

Note: At this point, those init containers will be waiting to discover Service named **myservice1**.

Here's a configuration you can use to make those Services appear:

**service.yaml**

apiVersion: v1

kind: Service

metadata:

  name: myservice1

spec:

  ports:

  - protocol: TCP

    port: 80

    targetPort: 9376

---

apiVersion: v1

kind: Service

metadata:

  name: myservice2

spec:

  ports:

  - protocol: TCP

    port: 81

    targetPort: 9376

kubectl apply -f service.yaml

kubectl get pods

Note: That Main container is now created.

**Use InitContainer to override index.html with shared emptyDir volume**

apiVersion: apps/v1

kind: DaemonSet

metadata:

  name: nginx

  labels:

    kubernetes.courselabs.co: daemonsets

spec:

  selector:

    matchLabels:

      app: nginx

  template:

    metadata:

      labels:

        app: nginx

    spec:

      initContainers:

        - name: init-html

          image: ubuntu

          command: ['sh', '-c', "echo '<!DOCTYPE html><html><body><h1>I was written by an init container</h1></body></html>' > /html/index.html"]

          volumeMounts:

            - name: html

              mountPath: /html

      containers:

        - image: nginx:1.18-alpine

          name: nginx

          volumeMounts:

            - name: html

              mountPath: /usr/share/nginx/html

            - name: logs

              mountPath: /var/log/nginx

      volumes:

        - name: html

          emptyDir: {}

        - name: logs

          hostPath:

            path: /volumes/nginx-logs

            type: DirectoryOrCreate

**Example2: Use InitContainer to override index.html with shared emptyDir volume:**

apiVersion: apps/v1

kind: Deployment

metadata:

  name: nginx

spec:

  selector:

    matchLabels:

      app: nginx

  template:

    metadata:

      labels:

        app: nginx

    spec:

**initContainers**:

        - name: init-html

          image: ubuntu

          command: ['sh', '-c', "**sleep 5;** echo '<!DOCTYPE html><html><body><h1>I was written by an init container</h1></body></html>' > **/html**/index.html"]

          volumeMounts:

            - name: **html**

              mountPath: /html

**containers**:

        - image: nginx:1.18-alpine

          name: **nginx**

          volumeMounts:

            - name: **html**

              mountPath: /usr/share/nginx/html

**volumes:**

        - name: **html**

          emptyDir: {}

Node Selector and Affinity

1. List the [nodes](https://kubernetes.io/docs/concepts/architecture/nodes/) in your cluster, along with their labels:

**kubectl get nodes --show-labels**

1. Choose one of your nodes, and add a label to it:

**kubectl label nodes <your-node-name> disktype=ssd**

1. Create pod that gets scheduled to your chosen node

apiVersion: v1

kind: Pod

metadata:

  name: nginx

  labels:

    env: test

spec:

  nodeSelector:

**disktype: ssd**

  containers:

  - name: nginx

    image: nginx

    imagePullPolicy: IfNotPresent

If the Pod doesn’t find an node with label disktyp: ssd, it remains in **pending** state.

**Affinity and anti-affinity**

* 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

**Node affinity:**

There are two types of node affinity:

* **requiredDuringSchedulingIgnoredDuringExecution**: (Hard Constraints) The scheduler **can't schedule** the Pod unless the rule is met. Once the Pod is running, the system will **not** take any action if the label is removed from the node.
* **preferredDuringSchedulingIgnoredDuringExecution**: (Soft Constraints) The scheduler tries to find a node that meets the rule. If a matching node is not available, the scheduler **still schedules** the Pod.

apiVersion: v1

kind: Pod

metadata:

  name: with-node-affinity

spec:

  affinity:

    nodeAffinity:

**requiredDuringSchedulingIgnoredDuringExecution**:

        nodeSelectorTerms:

        - matchExpressions:

          - key: disktype

            operator: In

            values:

            - ssd

**preferredDuringSchedulingIgnoredDuringExecution**:

      - weight: 1

        preference:

          matchExpressions:

          - key: disktype

            operator: In

            values:

            - ssd

  containers:

  - name: nginx-con

    image: nginx

You can use the operator field to specify a logical operator for Kubernetes to use when interpreting the rules. You can use In, NotIn, Exists, DoesNotExist, Gt and Lt.

Pod affinity / anti-affinity

Inter-pod affinity and anti-affinity allow you to constrain which nodes your Pods can be scheduled based on the **labels of Pods already running** on that node, instead of the node labels.

You could use **requiredDuringSchedulingIgnoredDuringExecution** affinity to tell the scheduler to co-locate Pods of two services in the same cloud provider zone because they communicate with each other a lot. Similarly, you could use **preferredDuringSchedulingIgnoredDuringExecution** anti-affinity to spread Pods from a service across multiple cloud provider zones.

|  |  |
| --- | --- |
| apiVersion: v1  kind: Pod  metadata:    name: with-pod-affinity  spec:    affinity:  **podAffinity**:        requiredDuringSchedulingIgnoredDuringExecution:          topologyKey: **kubernetes.io/hostname**        - labelSelector:            matchExpressions:            - key: app              operator: In              values:              - frontend  **podAntiAffinity**:        preferredDuringSchedulingIgnoredDuringExecution:        - weight: 100          podAffinityTerm:  **labelSelector**:              matchExpressions:              - key: app                operator: In                values:                - backend  **topologyKey**: **kubernetes.io/hostname**    containers:    - name: with-pod-affinity      image: registry.k8s.io/pause:2.0 |  |

* **labelSelector**: This is used to select other Pods based on labels. In this case, it selects Pods that have the label **app=frontend**.
* **topologyKey**: It tells Kubernetes how to group or spread Pods based on certain attributes of the cluster's infrastructure. Nodes that have a label with this key and identical values are considered to be in the same topology.

**Example**:

* + **Node-Level Distribution**: If you want to make sure certain Pods are not scheduled on the same node, you can use **kubernetes.io/hostname** as the topologyKey.
  + **Zone-Level Distribution**: If you want to distribute Pods across different availability zones, you can use **failure-domain.beta.kubernetes.io/zone** as the topologyKey. This ensures that Pods are spread across different zones to increase resilience.

The **affinity** **rule** says that the scheduler can **only** **schedule** a Pod onto a node if the node is in the same host as one or more existing Pods with the label **app=frontend.**

The **anti-affinity** rule says that the scheduler should try to **avoid** **scheduling** the Pod onto a node that is in the same host as one or more Pods with the label **app=backend.**

Taints and Tolerations

Taints and tolerations are a mechanism that allows you to ensure that pods are not placed on inappropriate nodes.

**Taints are added to nodes, while tolerations are defined in the pod specification.**

When you taint a node, it will **repel** all the pods except those that have a toleration for that taint.

A taint can produce three possible effects:

1. **NoSchedule**: The Kubernetes scheduler will only allow scheduling pods that have tolerations for the tainted nodes.
2. **PreferNoSchedule**: The Kubernetes scheduler will try to avoid scheduling pods that don’t have tolerations for the tainted nodes.
3. **NoExecute**: Kubernetes will **evict** the running pods from the nodes if the pods don’t have tolerations for the tainted nodes.

**To get nodes and their taints:**

kubectl get nodes -o=custom-columns=NodeName:.metadata.name,TaintKey:.spec.taints[\*].key,TaintValue:.spec.taints[\*].value,TaintEffect:.spec.taints[\*].effect

**To Add a taint to a node:**

kubectl **taint** nodes frontend-node app=frontend:NoSchedule environment=development:NoSchedule

**To Remove taint from a node:**

kubectl taint nodes frontend-node environment**-**

**To overwrite the taint:**

kubectl taint nodes frontend-node environment=development:NoExecute **--overwrite=true**

**For example, imagine you taint a node like this**

kubectl taint nodes node1 key1=value1:NoSchedule

kubectl taint nodes node1 key1=value1:NoExecute

kubectl taint nodes node1 key2=value2:NoSchedule

**And a pod has two tolerations:**

**tolerations**:

- **key**: "key1"

**operator**: "Equal"

**value**: "value1"

**effect**: "NoSchedule"

- **key**: "key1"

**operator**: "Equal"

**value**: "value1"

**effect**: "NoExecute"

In this case, the pod **will no**t be able to schedule onto the node, because there is no toleration matching the third taint. But it will be **able to continue** running if it is already running on the node when the taint is added, because the third taint is the only one of the three that is not tolerated by the pod.

**Example:**

**Set the taints:**

kubectl **taint** nodes aks-default-38464568-vmss000005 **app=frontend:NoSchedule**

kubectl **taint** nodes aks-default-38464568-vmss000006 **app=backend:NoSchedule**

**pod.yaml**

apiVersion: v1

kind: Pod

metadata:

  name: mynginx

  labels:

    env: test

spec:

  containers:

  - name: nginx

    image: nginx

    imagePullPolicy: IfNotPresent

**tolerations**:

  - key: "app"

    operator: "Equal"

    value: "frontend"

    effect: "NoSchedule"

kubectl apply -f pod.yaml

#Note that Pod is scheduled on Node with label app=Frontend.

kubectl get pod -owide

**Working with Probes**

A Probe is a diagnostic performed **periodically** by the Kubelet on a container.

**Pod Health Probes**

* A Pod is considered **Ready** when all containers are Ready.
* We can add additional intelligence to our Pod’s state and health.

**Types of Probes:**

1. **Liveness** **Probes**: To know when to restart a container. It continues to be checked at a regular interval of time.
2. **Readiness** **Probes**: To know when a container is ready to start accepting traffic. The pod will not become available to the cluster until a test is met and returns a successful exit code.
3. **Startup** **Probes**: To know when a container application has started.
4. **Liveness Probes:** To know when to restart a container. Used when an application is running, but unable to make progress. Restarting a container in such a state can help to make the application more available despite bugs.

apiVersion: v1

kind: Pod

metadata:

  labels:

    test: liveness

  name: mynginx

spec:

  containers:

  - name: mynginx-con

    image: nginx

    args:

    - /bin/sh

    - -c

    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600

**livenessProbe:**

**exec:**

        command:

        - cat

        - /tmp/healthy

      initialDelaySeconds: 5 #Default 0

      periodSeconds: 5 #Default 10

      timeoutSeconds: 5 #Default 1

      successThreshold: 1 #Default 1

      failureThreshold: 3 #Default 3

In the configuration file, you can see that the Pod has a single Container. The **periodSeconds** field specifies that the kubelet should perform a liveness probe every 5 seconds. The **initialDelaySeconds** field tells the kubelet that it should wait 5 seconds before performing the first probe. To perform a probe, the kubelet executes the command cat /tmp/healthy in the target container. If the command succeeds, it returns 0, and the kubelet considers the container to be alive and healthy. If the command returns a non-zero value, the kubelet kills the container and restarts it after 3 consequtive failures.

**kubectl apply -f liveness.yaml**

**Note:**

**kubectl describe pod mynginx**

Within 30 seconds, view the Pod events, the output indicates that no liveness probes have failed yet:

After 35 seconds, view the Pod events again, the output indicates that liveness probes have failed and the containers have been killed and recreated.

## Define a liveness with HTTP request

**httpGet Options:**

1. host – Host name to connect to
2. scheme – Protocol type HTTP or HTTPS
3. path – Path to access on the HTTP/HTTPS server
4. httpHeaders– Custom headers to set in the request
5. port – Port number to access on the container

apiVersion: v1

kind: Pod

metadata:

  labels:

    test: liveness

  name: mynginx

spec:

  containers:

  - name: liveness

    image: nginx

ports:

    - name: liveness-port

containerPort: 80

    args:

    - /server

**livenessProbe:**

**httpGet:**

        path: /healthz

        port: liveness-port

        httpHeaders:

        - name: Custom-Header

          value: Awesome

      initialDelaySeconds: 3

      periodSeconds: 3

      successThreshold: 1

      failureThreshold: 3

Note: Any status code greater than or equal to 200 and less than 400 indicates success. Any other code indicates failure.

1. **Readiness Probes:**

To know when a container is **ready to start accepting traffic**. One use of this signal is to control which Pods are used as backends for Services. When a Pod is not ready or on failure, it is removed from Service load balancers. **Prevents users from seeing errors**.

pod.yaml

apiVersion: v1

kind: Pod

metadata:

  labels:

    test: liveness

  name: mynginx

spec:

  containers:

  - name: mynginx-con

    image: nginx

**readinessProbe**:

      exec:

        command:

        - cat

        - /tmp/healthy1

      initialDelaySeconds: 5

      periodSeconds: 15

* kubectl apply -f pod.yaml
* kubectl get pods
* kubectl describe pod myngix
* kubectl exec -it mynginx -- /bin/bash
* # touch /tmp/healthy
* # exit

Wait at least five seconds, then check the pods again. Once the probe runs again the container should show available quickly.

* kubectl get pods mynginx --watch

A screen shot of a black screen

Description automatically generated

1. **Startup Probes:**

To know when a container **application has started**. If such a probe is configured, it **disables liveness and readiness checks until it succeeds**, making sure those probes don't interfere with the application startup. This can be used to adopt liveness checks on slow starting containers, **avoiding them getting killed by the kubelet before they are up and running.**

**startupProbe:**

      httpGet:

        path: /healthz

        port: liveness-port

      failureThreshold: 30

      periodSeconds: 10

Thanks to the startup probe, the application will have a **maximum** of **5 minutes (30 \* 10 = 300s)** to finish its startup. Once the startup probe has succeeded **once**, the **liveness** probe takes over to provide a fast response to container deadlocks. If the startup probe never succeeds, the container is killed after 300s and subject to the pod's **restartPolicy**.