Caltech | Center for Technology & Management Education

Post Graduate Program in DevOps

Source: https://kubernetes.io/docs/

simplilearn

DevOps

Caltech | Center for Technology & Management Education

CKA - Certified Kubernetes Administrator

simplilearn

Workloads

Learning Objectives

By the end of this lesson, you will be able to:

- Define Workloads
- Describe Containers and how to manage them
- Present an overview of Health Monitoring, Pod Topology, and Static Pods
- Discuss ReplicaSets, application configuration
- Define Self-Healing Pods

Overview of Workloads

Overview

A Workload is an application running on Kubernetes.

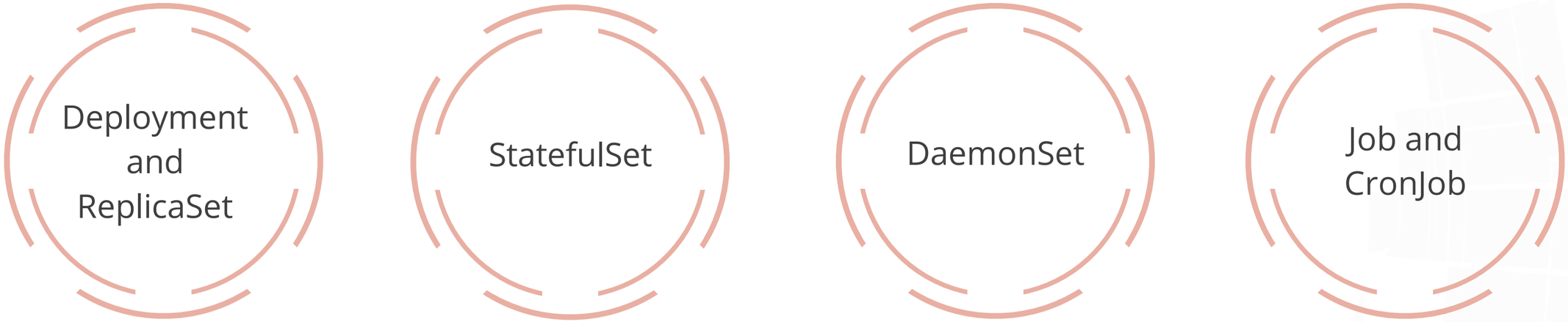
Whether the Workload has a single component or several that work together, it is run inside a set of Pods.

Once a Pod is running in the cluster, then a critical fault on the Node where that Pod is running means that all the Pods on that Node fail.

Kubernetes treats that level of failure as final; it creates a new Pod to recover, even if the prior Node later becomes healthy.

Overview

Kubernetes provides several built-in Workload resources:



Third-party Workload resources that provide additional behaviors are found in the wider Kubernetes ecosystem.

Deployment

What Is Deployment?

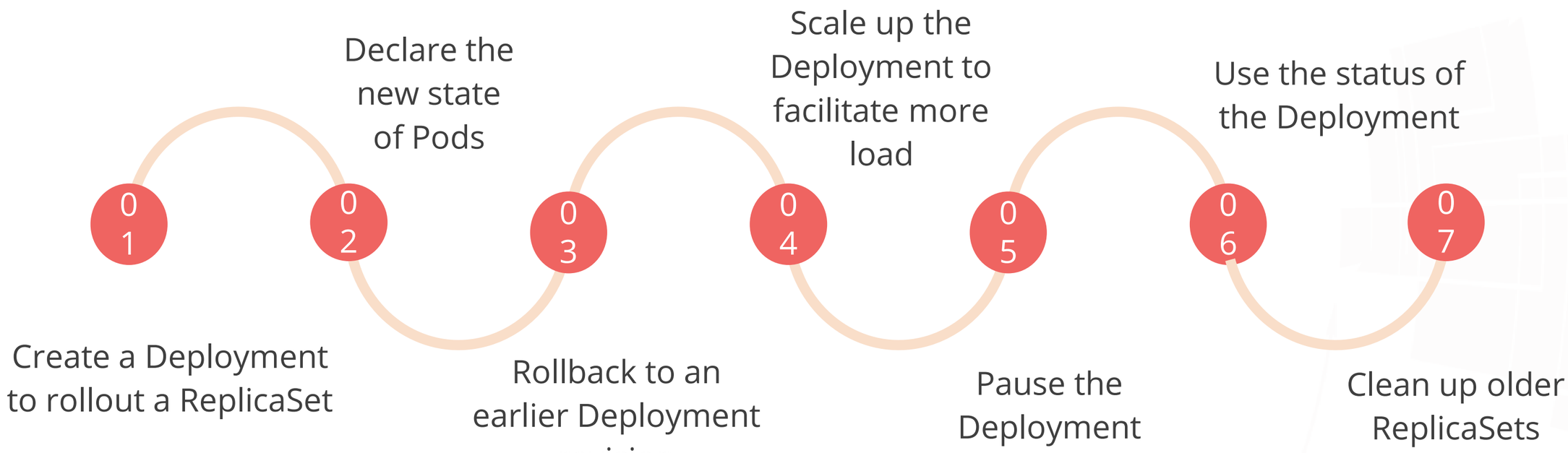
A Deployment provides declarative updates for Pods and ReplicaSets.

A desired state can be defined in a Deployment, and the Deployment controller changes the actual state to the desired state at a controlled rate.

Deployments can be defined to create new ReplicaSets or to remove existing Deployments and adopt all their resources with new Deployments.

Use Cases

The following are typical use cases for Deployments:



Assisted Practice

Understanding the Working of Deployments

Duration: 10 mins

Problem Statement:

Learn to work with Deployments.

Assisted Practice: Guidelines

Steps to demonstrate Deployment in Kubernetes:

1. Update to a new version of the code.
2. Check the status of the Deployment.

Understanding Pods

Introduction

Pods are the smallest deployable units of computing that can be created and managed in Kubernetes.

A Pod is a group of one or more Containers with shared storage and network resources and a specification for how to run the Containers.

The Pod's contents are always co-located and co-scheduled, as well as run in a shared context.

The Pod models an application-specific **logical host**. It contains one or more application Containers that are relatively tightly coupled.

Using Pods

Pods do not need to be created directly, not even singleton Pods. Instead, use Workload resources such as Deploymentor Job. If the Pods need to track state, consider the StatefulSet resource.

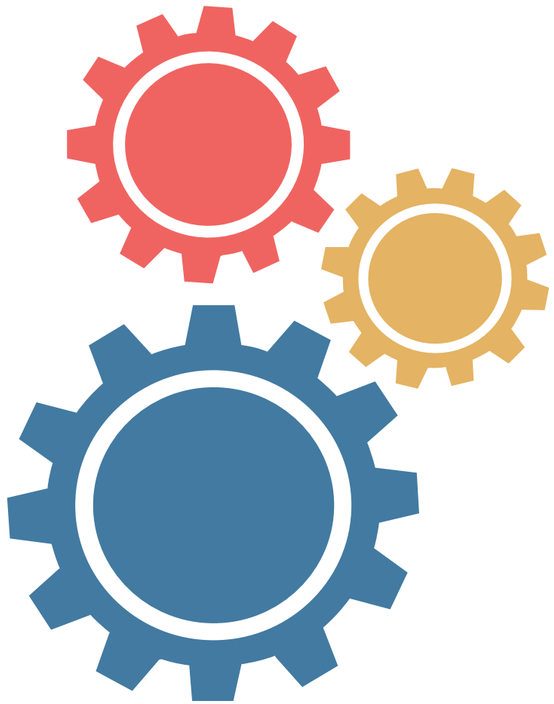
Pods in a Kubernetes cluster are used in two main ways:

Pods and Controllers

When a Pod gets created, the new Pod is scheduled to run on a Node in your cluster.

A controller for the resource handles replication and rollout and automatic healing in case of Pod failure. Example:

Deployment



Statefulset

Daemonset

Pod Templates

These are specifications that help us to create Pods. They are included in Workload resources such as Deployments, Jobs, and DaemonSets.

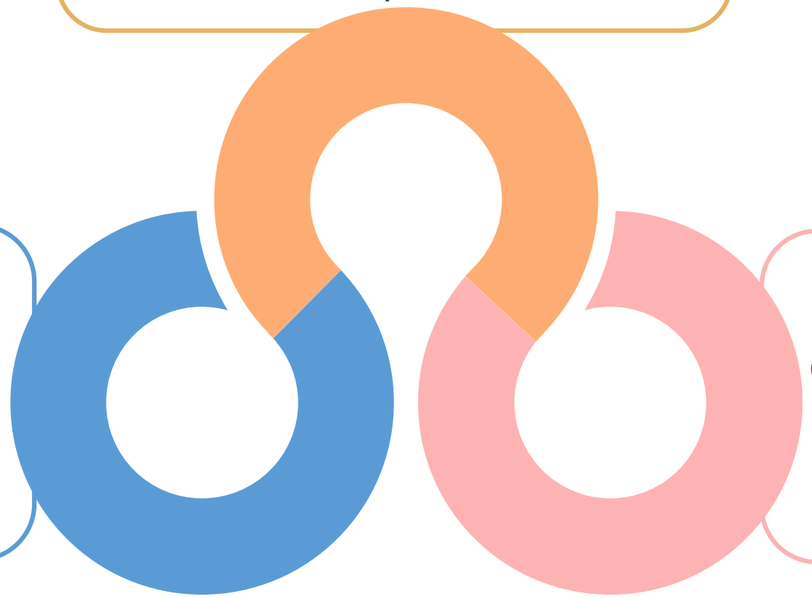
This sample is a manifest for a basic Job with a template that starts a single Container:

Demo

```
apiVersion: batch/v1
Kind:  Job
Metadata:
   name:  hello
   spec:
     ktemplate:
       #This is the pod template
       spec:
        containers:
  - name: hello
     protocol: TCP
     image: busybox
     command: [‘sh’, ‘-c’, ‘echo “Hello, kubernetes!” && sleep 3600’]
  restartPolicy: OnFilure
       # The pod template ends here
```

Pods and Controllers

Each Workload resource implements its own rules for handling changes to the Pod template.



Modifying the Pod template or switching to a new Pod template has no direct effect on the Pods that already exist.

The kubelet does not directly observe or manage any of the details around Pod templates and updates.

Pod Update and Replacement

Pod update operations like patch and replace have some limitations:

The metadata about a Pod is immutable.

If the metadata.deletionTimestamp is set, no new entry can be added to the metadata.finalizers list.

Pod updates may not change fields other than spec.containers[*].image, spec.initContainers[*].image, spec.activeDeadlineSeconds or spec.tolerations.

Two types of updates are allowed while updating the spec.activeDeadlineSeconds field.

Resource Sharing and Communication

Pods enable data sharing and communication among their constituent Containers.

All Containers in the Pod can access the shared volumes, allowing those Containers to share data.

Inside a Pod (and only then) can the Containers, that belong to the Pod, communicate with one another using localhost.

Storage in Pods

Pod networking: each Pod has a unique IP address for each address family.

Privileged Mode for Containers

Any Container in a Pod can enable Privileged Mode using the privileged flag on the security context of the container spec.

Useful for Containers that want to use operating system administrative capabilities such as manipulating the network stack or accessing hardware devices.

Assisted Practice

Understanding Pods

Duration: 5 mins

Problem Statement:

Comprehend the working of Pods.

Assisted Practice: Guidelines

Steps to demonstrate Pods in Kubernetes:

1. Create a Deployment to create a Pod.
2. Create a Deployment file named singlepod.yaml ***Vi singlepod.yaml***.
3. Type ***kubectl apply -f singlepod.yaml.***
4. Type ***kubectl get pods -n twitter***.

Pod Lifecycle

Pod Lifecycle

Pods follow a defined Life Cycle, starting in the Pending phase, moving through Running* and then through either the Succeeded or Failed phase, depending on whether any Container in the Pod terminated in failure.

Within a Pod, Kubernetes tracks different container states and determines what action to take to make the Pod healthy again.

In the Kubernetes API, Pods have both a specification and an actual status.

**If at least one of its primary Containers starts OK*

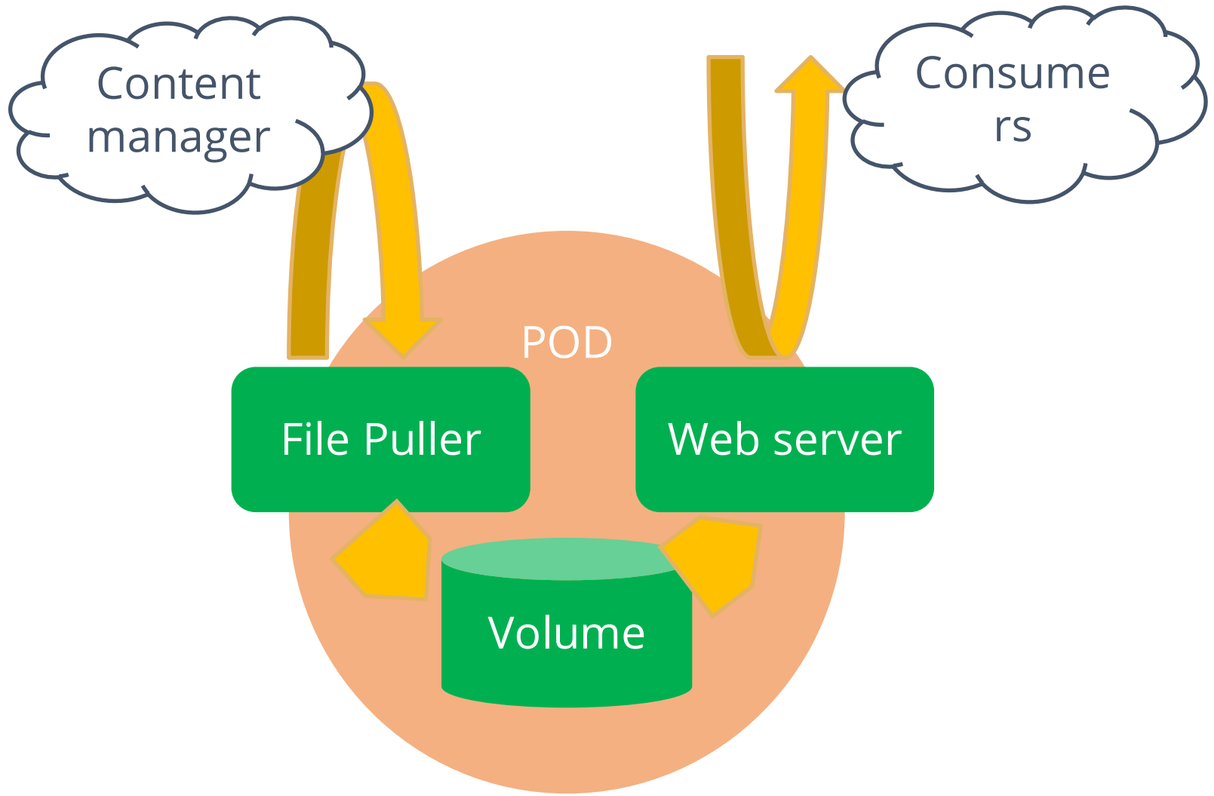
Pods are Ephemeral

Pods are created, assigned a unique ID (UID), and scheduled to Nodes where they remain until termination (according to restart policy) or deletion.

- If a Node dies, the Pods scheduled to that Node are scheduled for deletion after a timeout period.
- Kubernetes uses a higher-level abstraction called controller, which handles the work of managing the relatively disposable Pod instances.
- A given Pod (as defined by a UID) is never **rescheduled** to a different Node.
- A Pod can be replaced by a new, near-identical Pod, with even the same name if desired, but with a different UID.

Pods and Volumes

A multi-container Pod that contains a file puller and a web server that uses a persistent volume for shared storage between the Containers.



Phase of a Pod

This is a simple, high-level summary of where the Pod is in its Life cycle. A Pod's status field is a `PodStatus` object, which has a `phase` field.

Possible values for `phase`:

Value	Description
Pending	The Pod has been accepted by the Kubernetes cluster, but one or more of the Containers has not been set up and made ready to run.
Running	The Pod has been bound to a Node, and all the Containers have been created.
Succeeded	All Containers in the Pod have terminated successfully, and will not be restarted.

Pod Phase

If a Node dies or is disconnected from the rest of the cluster, Kubernetes applies a policy for setting the phase of all Pods on the lost Node to failed.

Possible values for phase:

Value	Description
Failed	All Containers in the Pod have terminated, and at least one Container has terminated in failure.
Unknown	This phase typically occurs due to an error in communicating with the Node where the Pod should be running.

Container States

Kubernetes tracks the state of each Container inside a Pod. Each state has a specific meaning:

02

Waiting

If a container is not in either the running or terminated state, it is waiting.

01

Terminated

A Container in the terminated state began execution and then either ran to completion or failed for some reason.

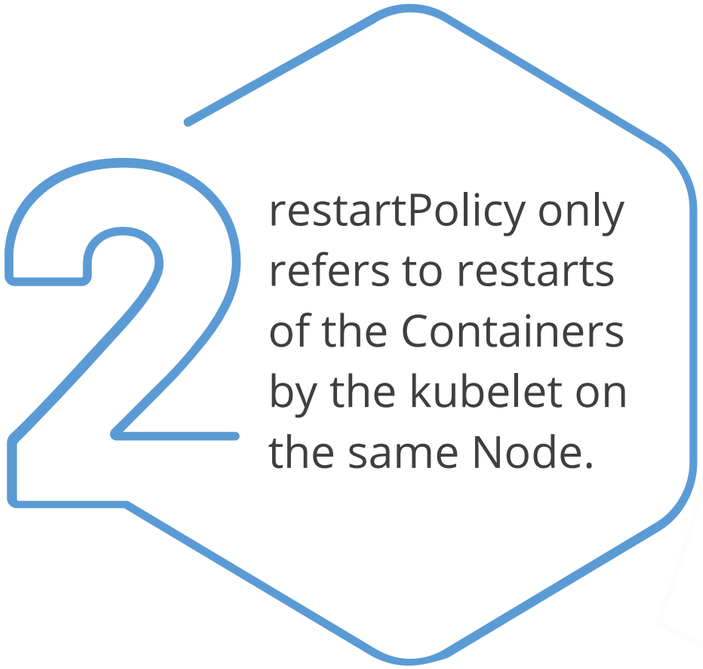
03

Running

The Running status indicates that a Container is executing without issues.

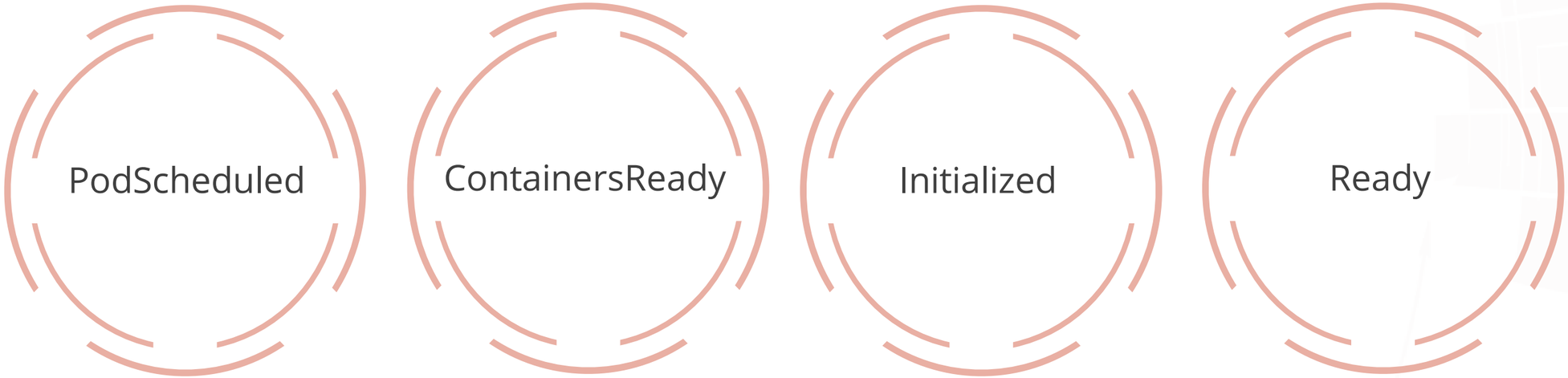
Container Restart Policy

The spec of a Pod has a restartPolicy field with possible values Always, OnFailure, and Never.



Pod Conditions

A Pod has a **PodStatus**, which has an array of **PodConditions** through which the Pod has passed or not. Following are the PodConditions:



Pod Conditions

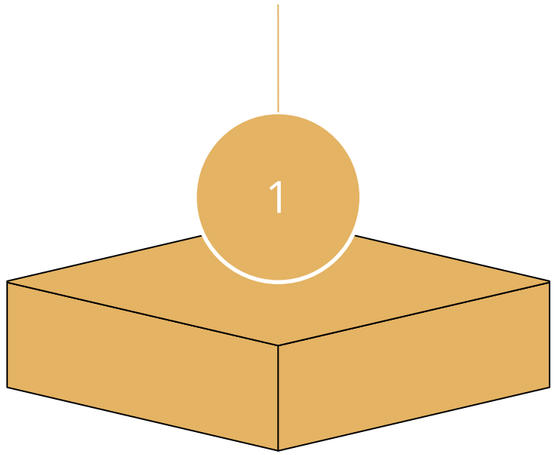
Field Name	Description
type	Name of this Pod condition
status	Indicates whether that condition is applicable, with possible values **True**, **False**, or **Unknown**
lastProbeTime	Timestamp of when the Pod condition was last probed
lastTransitionTime	Timestamp for when the Pod last transitioned from one status to another
reason	Machine-readable, UpperCamelCase text indicating the reason for the condition's last transition
message	Human-readable message indicating details about the last status transition

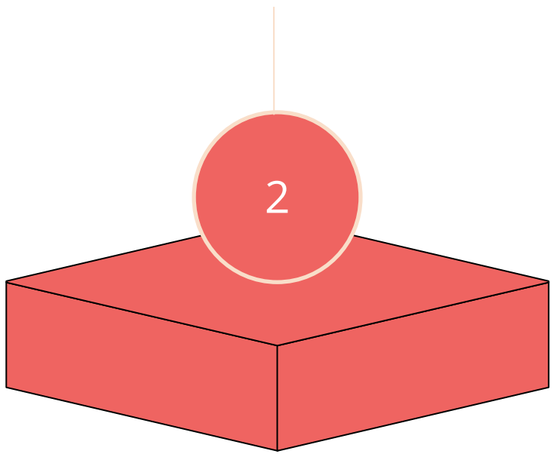
Pod Readiness

An application can inject extra feedback or signals into PodStatus: **Pod readiness**.

To use this, set readinessGates in the Pod's spec to specify a list of additional conditions that the kubelet evaluates for Pod readiness.

Readiness gates are determined by the current state of status.condition fields for the Pod.





Pod Readiness

Here is an example:

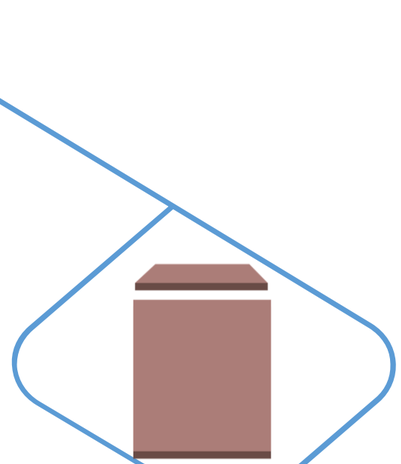
Demo

```
Kind:  pod
   ...
 spec:
    readinessGates:
      -  conditionType:  " www.example.com/feature-1"
 status:
    conditions: /
      - type:: Prefix                # a built in podCondition
        status:
        lastProbeTime:
        LastTransitionTime: 2018-01-01T00:00:00Z
        - tyoe: 'www.example.com/feature-1"     # an extra podCondition
        status: "False"
        lastProbeTime: null
        lastTransitionTime: 2018-01-01T00:00:00Z
    containerStatuses:
      - containerID:  docker://abcd...
        Ready: true
 ...
```

Status for Pod Readiness

In case of a Pod that uses custom conditions, it is evaluated to be ready only when both the statements given below apply:

All Containers in the Pod are ready.



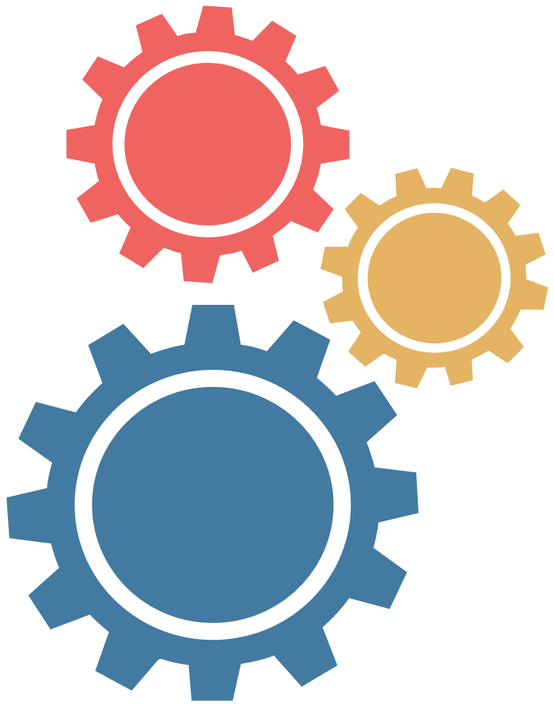
All conditions specified in readinessGates are True.

Container Probes

A Probe is a diagnostic performed periodically by the kubelet on a Container.

There are three types of handlers:

ExecAction



TCPsocketAction

HTTPGetAction

Container Probes

Each probe has one of three results:

Success
The container passed the diagnostic.

02

Unknown
The diagnostic failed, so no action should be taken.

01

03

Failure
The container failed the diagnostic.

Container Probes

The kubelet can optionally perform and react to three kinds of probes on running Containers:

livenessProbe

Indicates whether the Container is running

readinessProbe

Indicates whether the Container is ready to respond to requests

startupProbe

Indicates whether the application within the Container is started

Termination of Pods

The design aim is to be able to request deletion, know when processes terminate, and to ensure that deletes eventually complete.

If the kubelet or the Container runtime's management service is restarted while waiting for processes to terminate, the cluster retries from the start, including the full original grace period.

Forced Pod Termination

1. Forced deletions can be potentially disruptive for some Workloads and their Pods.

1. Kubectl delete command supports the **--grace-period=<seconds>** option to override the default value.

1. Control plane cleans up the terminated Pods when the number of Pods exceeds the configured threshold.

1. When a forced deletion is performed, the API server does not wait for confirmation from the kubelet that the Pod has been terminated on the Node on which it was running.

Assisted Practice

Understanding Pod Lifecycle

Duration: 10 mins

Problem Statement:

Learn about the Pod Lifecycle in Kubernetes.

Assisted Practice: Guidelines

Steps to demonstrate Pod Lifecycle in Kubernetes:

1. Get Pod status.
2. Type ***kubectl describe pod twitter***.

Working on Pod Allocation

Introduction

There are several ways to allocate Pods, and all the recommended approaches use label selectors to facilitate the selection.

Constraints are unnecessary as the Scheduler will automatically do a reasonable placement. However, there are some circumstances where you may want to control the Node to which the Pod deploys.

nodeSelector

nodeSelector is the simplest recommended form of Node selection constraint.

Example of how to use nodeSelector:

Step Zero: Prerequisites

This example assumes that you have a basic understanding of Kubernetes Pods and that you have set up a Kubernetes cluster.

Step One: Attach label to Node

Run **kubectl get nodes** command to get the names of your cluster Nodes.

nodeSelector

Step Two: Add a nodeSelector field to the Pod configuration.

For example:

Demo

```
apiVersion: v1
Kind:  pod
Metadata:
   name:  nginx
   Labels:
     env: test
spec:
     containers:
     -  name: nginx
        image: nginx
```

nodeSelector

Step three: Add a nodeSelector

Demo

```
apiVersion: v1
Kind:  pod
Metadata:
   name:  nginx
   Labels:
     env: test
spec:
     containers:
     -  name: nginx
        image: nginx
        imagePullPolicy: iFNotPresent
     nodeSelector:
        disktypr: ssd
```

Then run kubectl apply -f https://k8s.io/examples/pods/pod-nginx.yaml. The Pod will get scheduled on the Node that is attached to the label.

Node Isolation or Restriction

Node Isolation is used to ensure that only specific Pods run on Nodes with certain isolation, security, or regulatory properties.

To make use of that label prefix for Node isolation:

Use Nodeauthorizer and enable NodeRestriction and mission plugin

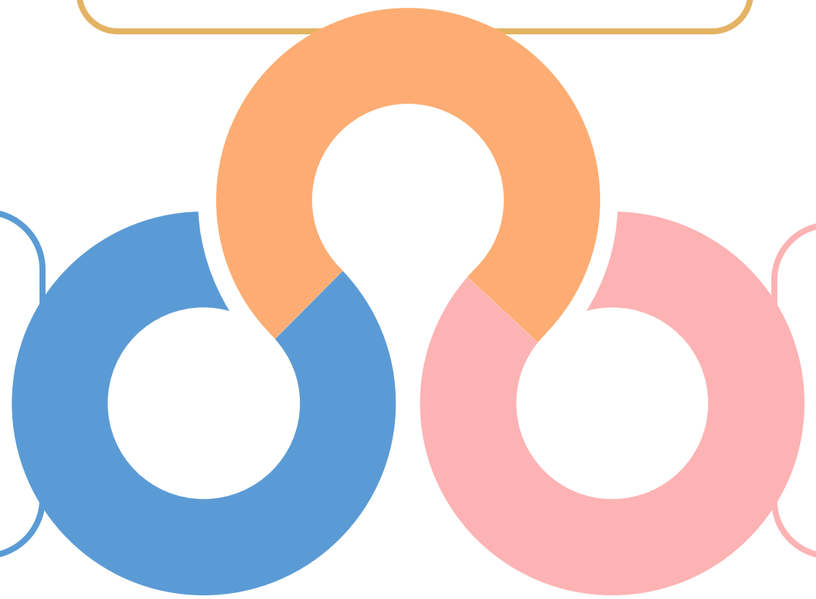
Add labels under the node-restriction.kubernetes.io/ prefix to your Node objects and use those labels in your Node selectors

Affinity and Anti-affinity

The Affinity or Anti-affinity feature expands the types of constraints that can be expressed.

The key enhancements are:

The Affinity or Anti-affinity language is more expressive.



Constrain against labels on other Pods running on the Node.

Indicate that the rule is **soft** or **preference** rather than a hard requirement.

Node Affinity

Node Affinity is conceptually similar to nodeSelector; it allows to constrain the nodes that the Pod is eligible to be scheduled on, based on the labels on the Node.

There are two types of Node Affinity:

- requiredDuringSchedulingIgnoredDuringExecution
- referredDuringSchedulingIgnoredDuringExecution

Node Affinity

Demo

```
    apiVersion: v1
    Kind:  Pod
    Metadata:
       name:  with-node-affinity
       spec:
         affinity:
         nodeAffinity:
          requiredDuringSchudulingIgnoredDuringExecution:
         nodeSelectorTerms:
          -  matchExpressions:
           - key: kubernetes.io/e2e-az-name
              operator: in
                 values:
                   - e2e-az1
                   - e2e-az2
                     preferedDuringSchedulingDuringException:
         - weight: 1
            preference:
              matchExpressions:
          key: another-node-label-key
operator: In
               values:
                  - another-node-label-value
                containers:
                -  Name: with-node-affinity
image: k8s.gcr.io/pause:2.0
```

Example of a Pod that uses Node Affinity:

This Node Affinity rule says that the Pod can only be placed on a Node with a label whose key is kubernetes.io/e2e-az-name and whose value is either e2e-az1 or e2e-az2.

Node Affinity

The operator is being used in the example. The new Node Affinity syntax supports the following operators:

```
NotIn, Exists, DoesNotExist, Gt, Lt.
```

Both nodeSelector and nodeAffinity must be satisfied for the Pod to be scheduled on a candidate Node.

If multiple matchExpressions associated with nodeSelectorTerms are specified, then the Pod can be scheduled on a Node only if allmatchExpressions are satisfied.

Feature State: Kubernetes V1.20 [Beta]

When configuring multiple scheduling profiles, associate a profile with a Node Affinity that is useful if a profile only applies to a specific set of Nodes. Add an addedAffinity to the arguments of the NodeAffinity plugin in the Scheduler configuration. For example:

Demo

```
apiVersion: kubescheduler.config.k8s.io/v1beta1
Kind:  KubeSchedulerConfiguration

profiles:
   - scheduleeName:  default-scheduler
   - schedulerNmae; foo-scheduler
     pluginConfig:
     - name:  NodeAffinity
      args:
     naddedAffinity:
      requiredDuringSchedulerIgnoredDuringxExecution:
       nodeSelectorTerms:
       - MatchExpressions: in
          - key:  scheduler-profile
                operator:  In
     values:
        - foo
```

Feature State: Kubernetes V1.20 [Beta]

There are existing Kubernetes concepts that allow exposing a single service. This can be achieved through an Ingress by specifying a default backend with no rules:

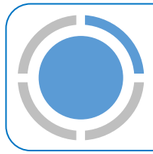
Demo

```
apiVersion: networking.k8s.10/v1
Kind:  Ingress
Metadata:
   name:  test-ingress
   spec:
     defaultBackend
     service:
     name: test
     port: k8s.exaple.com
        number: 80
```

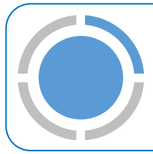
Inter-Pod Affinity and Anti-affinity

Inter-Pod affinity and Anti-affinity allow constraining the Nodes the Pod is eligible to be scheduled based on labels on Pods that are already running on the Node rather than on labels on the Nodes.

There two types of Pod Affinity and Anti-affinity:



requiredDuringSchedulingIgnoredDuringExecution



preferredDuringSchedulingIgnoredDuringExecution

Inter-Pod Affinity and Anti-Affinity

Demo

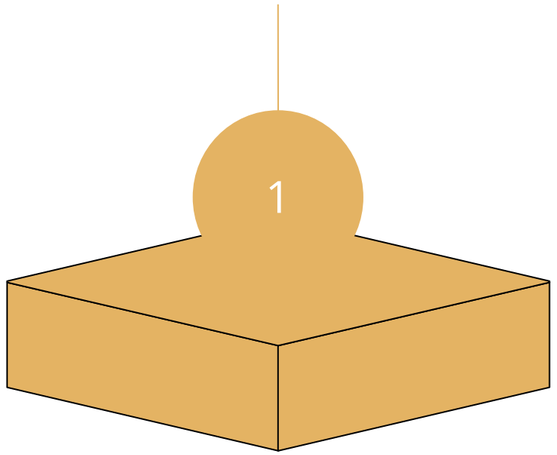
```
apiVersion: v1
Kind:  pod
Metadata:
   name:  with-pod-affinity
   spec:
     affinity:
     nodeAffinity:
      requiredDuringSchudulingIgnoredDuringExecution:
       - labelSelector:
           matchExpressions:
           - key: security
             operator: in
             values:
               - 51
         topologyKey : topology.kebernetes.io/zone
     podAntiAffinity:
               preferedDuringSchedulingIgnoredException:
     - weight: 100
        podAffinityTerm:
          labelSelector
         matchExpressions:
     - key: security
       operator: In
           values:
              - 52
             topologyKey: topology.kubernetes.io/zone
      containers:
             -  Name: with-pod-affinity
image: k8s.gcr.io/pause:2.0
```

An example of a Pod that uses Pod Affinity.

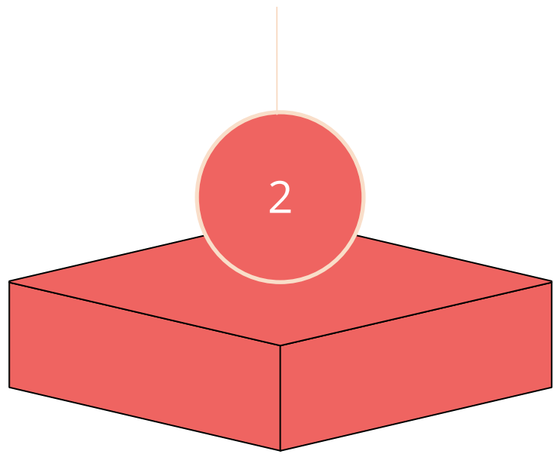
The Affinity on this Pod defines one Pod Affinity rule and one Pod Anti-affinity rule.

Inter-Pod Affinity and Anti-affinity

The Pod Affinity rule says that the Pod can be scheduled on a Node only if the Node is in the same zone as at least one already-running Pod that has a label with key **security** and value. **S1**.



The Pod Anti-affinity rule says that the Pod should not be scheduled on a Node if the Node is in the same zone as a Pod with label having key **security** and value **S2**.



Constraints on TopologyKey

For performance and security reasons, there are some constraints on topologyKey:

1. For Pod Affinity, empty topologyKey is not allowed in both.
2. For Pod Anti-affinity also, empty topologyKey is not allowed in both.
3. For requiredDuringSchedulingIgnoredDuringExecution Pod Anti-affinity, the admission controller LimitpodHardAntiAffinityTopology was introduced to limit topologyKey to kubernetes.io/hostname.
4. Except for the above cases, the topologyKey can be any legal label key.

Feature State: Kubernetes V1.21 [Alpha]

The Affinity term is applied to the union of the namespaces selected by the namespaceSelector and the ones listed in the namespaces field.

An empty namespaceSelector ({}) matches all namespaces while a null or empty namespaces list and null namespaceSelector means **this pod's namespace**.

Interpod Affinity and Anti-affinity can be more useful when they are used with higher level collections such as ReplicaSets, StatefulSets, Deployments, etc.

One can easily configure that a set of Workloads should be co-located in the same defined Topology, for example, the same node.

Feature State: Kubernetes V1.21 [Alpha]

Demo

```
apiVersion: v1
Kind:  Deployment
Metadata:
   name:  redis-cache
   spec:
     selector:
     matchlabels:
      app: store
    replicas: 3
    template:
     metadata:
      labels:
         app: store
    spec:
      affinity:
        podAntiAffinity:
      requiredDuringSchudulingIgnoredDuringExecution:
       - labelSelector:
           matchExpressions:
           - key: app
             operator: in
             values:
               - store
         topologyKey : "kubernetes.io/hostname"
      containers:
             -  Name: with-pod-affinity
               image: redis: 3.2- alpine
```

yaml snippet of a simple redis Deployment with three Replicas and selector label app=store

Feature State: Kubernetes V1.21 [Alpha]

yaml snippet of the webserver deployment has podAntiAffinity and podAffinity configured:

```
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:
          requiredDuringSchudulingIgnoredDuringExecution:
           - labelSelector:
               matchExpressions:
               - key: app
```

```
operator: in
                  values:
                    - web-store
              topologyKey : “kubernetes.io/hostname”
         podAntiAffinity:
           requiredDuringSchudulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                - key: app
                  operator: in
                  values:
                    - store
                topologyKey : “kubernetes.io/hostname”
           containers:
                  -  Name: web-app
                    image: nginx: 1.16- alpine
```

Feature State: Kubernetes V1.21 [Alpha]

To create two Deployments, the three Node cluster should be:

node-1	node-2	node-3
webserver-1	webserver-2	webserver-3
cache-1	cache-2	cache-3

As expected, all the three replicas of the web server are automatically colocated with the cache.

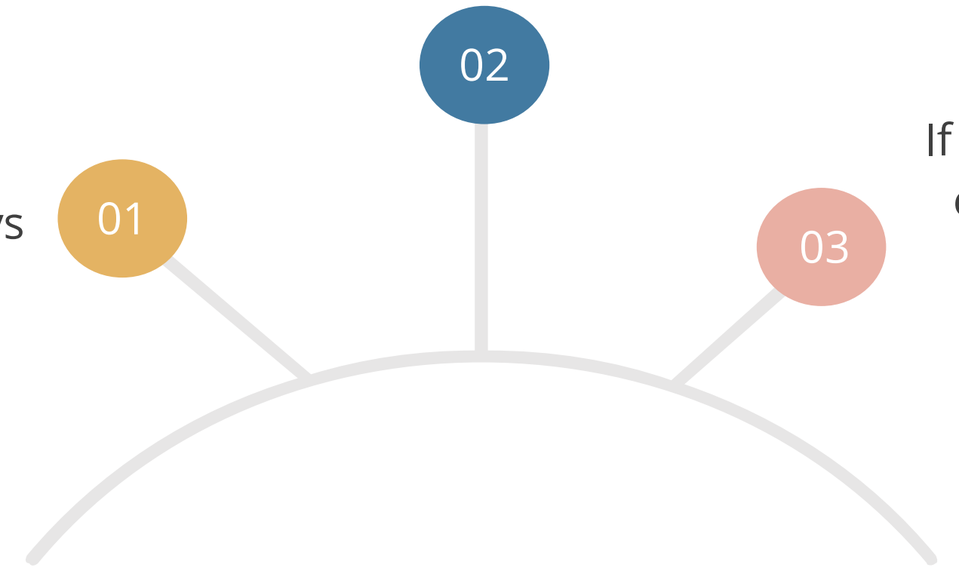
NodeName

NodeName is the simplest form of Node selection constraint, but owing to its limitations it is typically not used. Some of the limitations of using NodeName to select Nodes are:

01 Node names in cloud environments are not always predictable or stable.

02 If the named Node does not exist, the Pod will not run or might be automatically deleted.

03 If the named Node does not have enough resources, the Pod will fail with a reason indicated.



NodeName

Example of a Pod config file using the nodeName field:

Demo

```
apiVersion: v1
Kind:  pod
Metadata:
   name:  nginx
   spec:
     containers:
     - name: nginx
      image: nginx
     nodename: kube-01
```

This Pod will run on the Node kube-01.

Assisted Practice

Understanding the Working of Pod Allocation

Duration: 5 mins

Problem Statement:

Learn about allocation of Pods in Kubernetes.

Assisted Practice: Guidelines

Steps to demonstrate Pod Allocation in Kubernetes:

1. Add Node selector to the Pod configuration.
2. Open the Pod yaml file ***vi singlepod.yaml***.
3. Save the file.
4. Type ***kubectl apply -f singlepod.yaml***.

Init Containers

Introduction

Init Containers are specialized Containers that run before app Containers in a Pod.

Init Containers can contain utilities or setup scripts not present in an app image.

Init Containers

A Pod can have multiple Containers running apps within it. It can also have one or more Init Containers that are run before the app Containers are started.

Differences between regular and Init Containers:

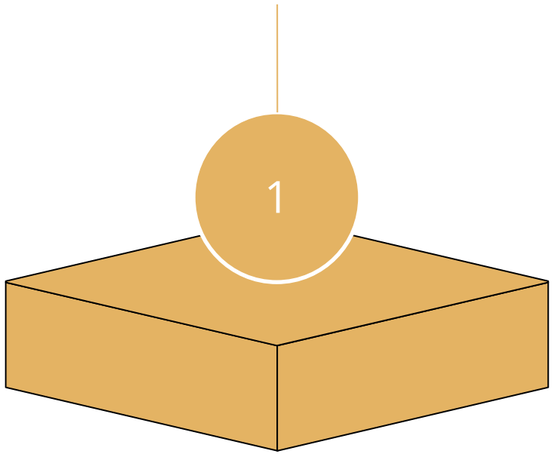
Init Containers support all the fields and features of app Containers, including resource limits, volumes, and security settings.

Init Containers do not support lifecycle, livenessProbe, readinessProbe, or startupProbe because they must run to completion before the Pod can be ready.

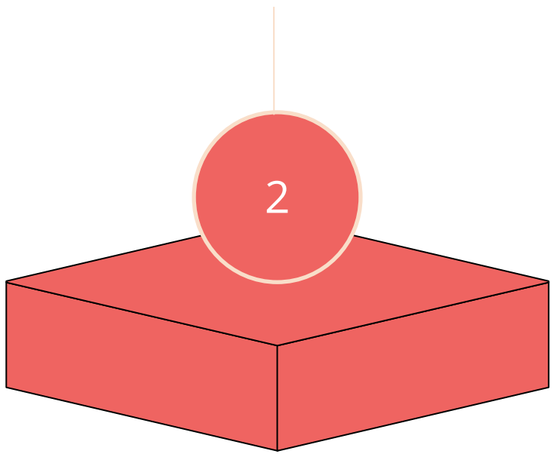
Init Containers

Advantages of Init Containers for start-up related code:

Init Containers can contain utilities or custom code for setup that are not present in an app image.



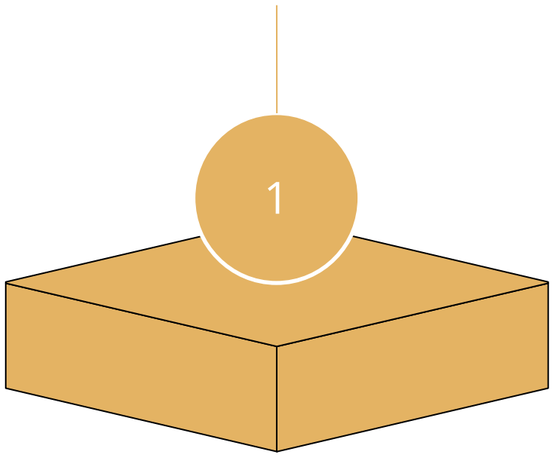
The application image builder and Deployer roles can work independently without the need to jointly build a single app image.



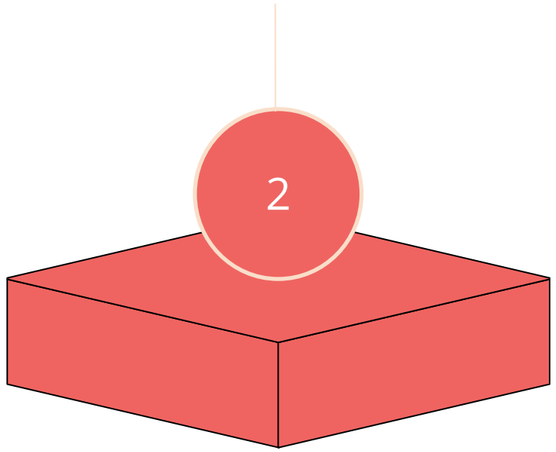
Init Containers

Init Containers offer certain advantages for start-up related code:

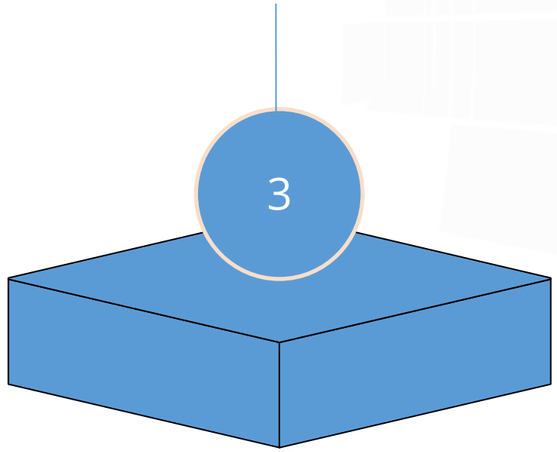
init Containers offer a mechanism to block or delay app container startup until a set of preconditions are met.



Init Containers can run with a different view of the filesystem than app Containers in the same Pod.



Storage Class lets administrators assign **classes** of storage-to-map service quality levels.



Init Containers in Use

This example defines a simple Pod that has two Init Containers.

Demo

```
    apiVersion: v1
    Kind:  pod
    Metadata:
       name:  myapp-pod
       labels:
         app: myapp
       spec:
         containers:
         - name: myapp-container
           image: busybox:1.28
           command: ['sh','-c','echo The app os running! 66 sleep3600
        initContainers:
           - name: myapp-container
           image: busybox:1.28
           command: ['sh','-c',"intil nslookup
myservice.$(cat/var/run/secrets/kubernetes.io/seviceaccount/namespace).svc.c
luster.local;do echo waiting for myservice;seep 2;done"]
         - name: myapp-container
           image: busybox:1.28
           command: ['sh','-c',"until nslookup
mydb.4(cat/var/run/secrets/kubernetes.io/serviceaccount/namespace.svc.cluste
r.local;do echo waiting for mydb;seep2;done"]
```

Init Containers in Use

Start this Pod by running:

Demo

```
kubectl apply -f myapp.yaml

The output is similar to this:

pod/myapp-pod created

And check on its status with:

kubectl get -f myapp.yaml

The output is similar to this:

NAME         READY     STATUS     RESTARTS   AGE
myapp-pod    0/1       Init:0/2   0          6m

or for more details:

kubectl describe -f myapp.yaml
```

Init Containers in Use

Here's a configuration that can be used to make Services appear:

Demo

```
    apiVersion: v1
    Kind:  service
    Metadata:
       name:  myservice
      spec:
         ports:
         -  protocol: TCP
            port: 80
            targetPort: 9376
apiVersion: v1
    Kind:  service
    Metadata:
       name:  mydb
      spec:
         ports:
         -  protocol: TCP
            port: 80
            targetPort: 9377
```

Init Containers in Use

Demo

```
To create the mydb and myservice services:

kubectl apply -f services.yaml

The output is similar to this:

service/myservice created
service/mydb created

See the init containers complete, and that the myapp-pod pod
moves into the Running state:

kubectl get -f myapp.yaml

The output is similar to this:

NAME         READY     STATUS    RESTARTS   AGE
myapp-pod    1/1       Running   0          9m
```

Assisted Practice

Understanding Init Containers

Duration: 5 mins

Problem Statement:

Learn about the working of Init Containers in Kubernetes.

Assisted Practice: Guidelines

Steps to demonstrate Init Container in Kubernetes:

1. Add init Container entries to the pod configuration.
2. Create a pod yaml file.
3. Save the file.
4. Type ***kubectl apply -f singlepod.yaml***.

Managing Container Resources

Introduction

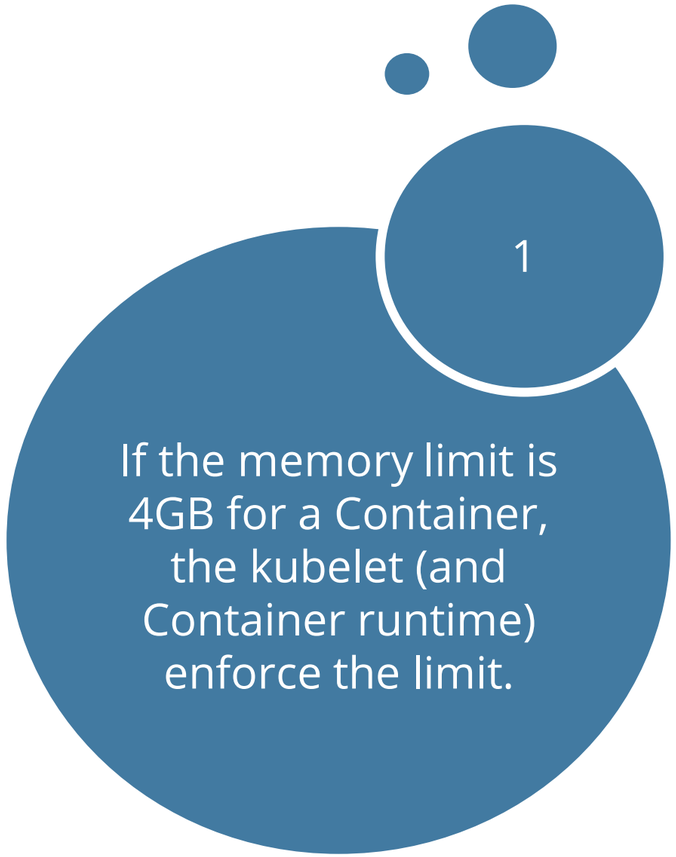
The most common resources to specify in a Pod are CPU and memory (RAM); there are others.

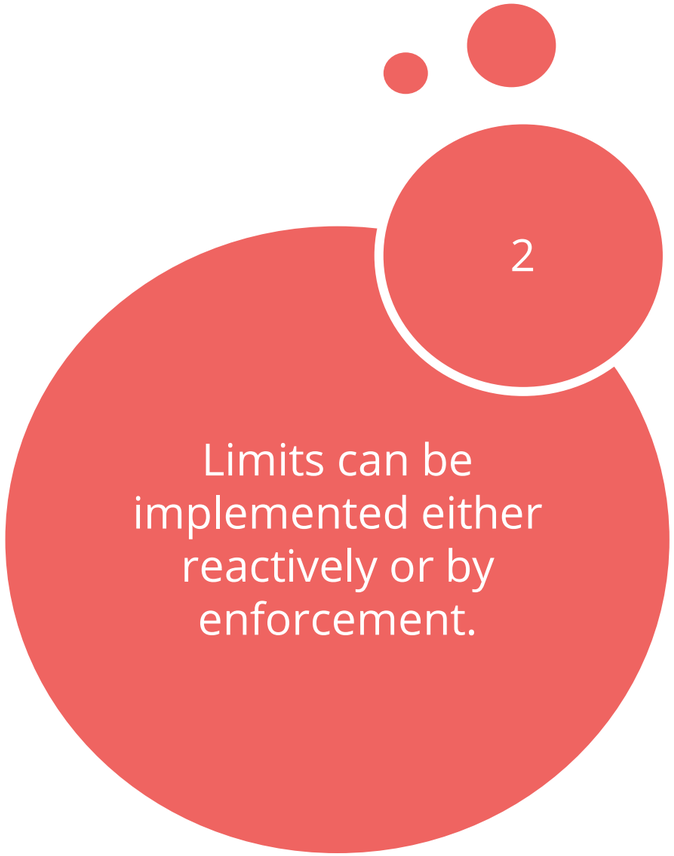
When the resource request is specified for Containers in a Pod, the Scheduler uses this information to decide on which Node to place the Pod.

When the resource limit is specified 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 set.

Requests and Limits

If the Node where a Pod is running has enough resource available, it is possible (and allowed) for a Container to use more resources than requested.

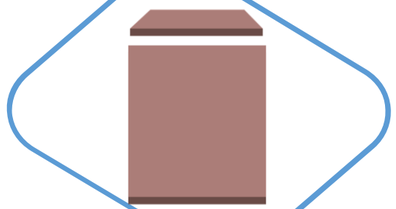




Resource Types

CPU and memory are both Resource types.

CPU represents compute processing and is specified in units of Kubernetes CPU.

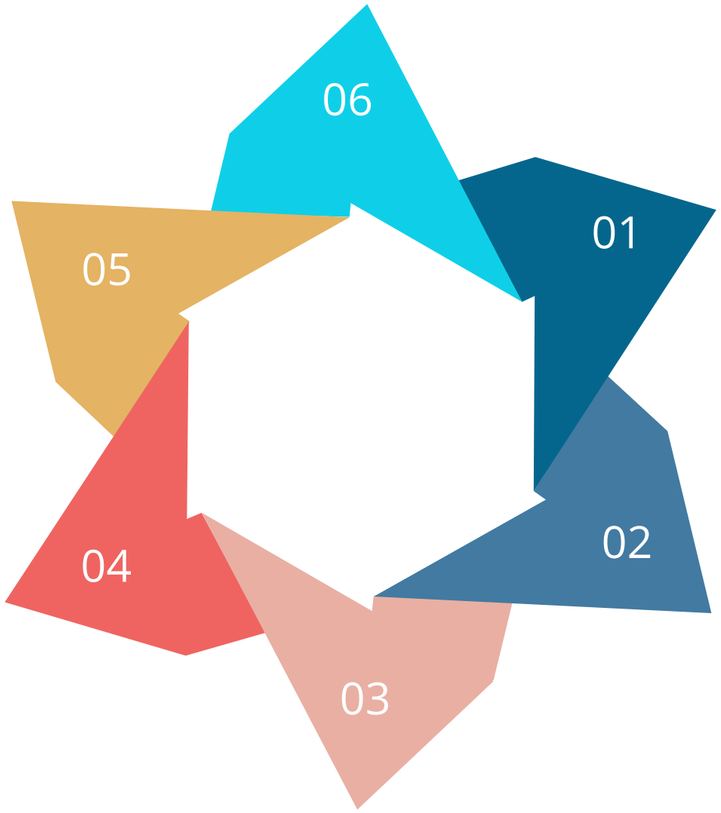


CPU and memory are collectively referred to as Compute Resources or Resources.

Resource Requests and Limits of Pod and Container

Each Container of a Pod can specify one or more of the following:

spec.containers[].resources.limits.cpu



spec.containers[].resources.requests.hugepages-<size>

spec.containers[].resources.limits.memory

spec.containers[].resources.requests.memory

spec.containers[].resources.limits.hugepages-<size>

spec.containers[].resources.requests.cpu

Resource Units in Kubernetes

In Kubernetes, one CPU is equivalent to 1 vCPU/Core for cloud providers and 1 hyperthread on bare-metal Intel processors.

Meaning of CPU

Memory can be expressed as a plain integer or as a fixed-point number, using one of these suffixes: E, P, T, G, M, K.

Meaning of Memory

Resource Units in Kubernetes

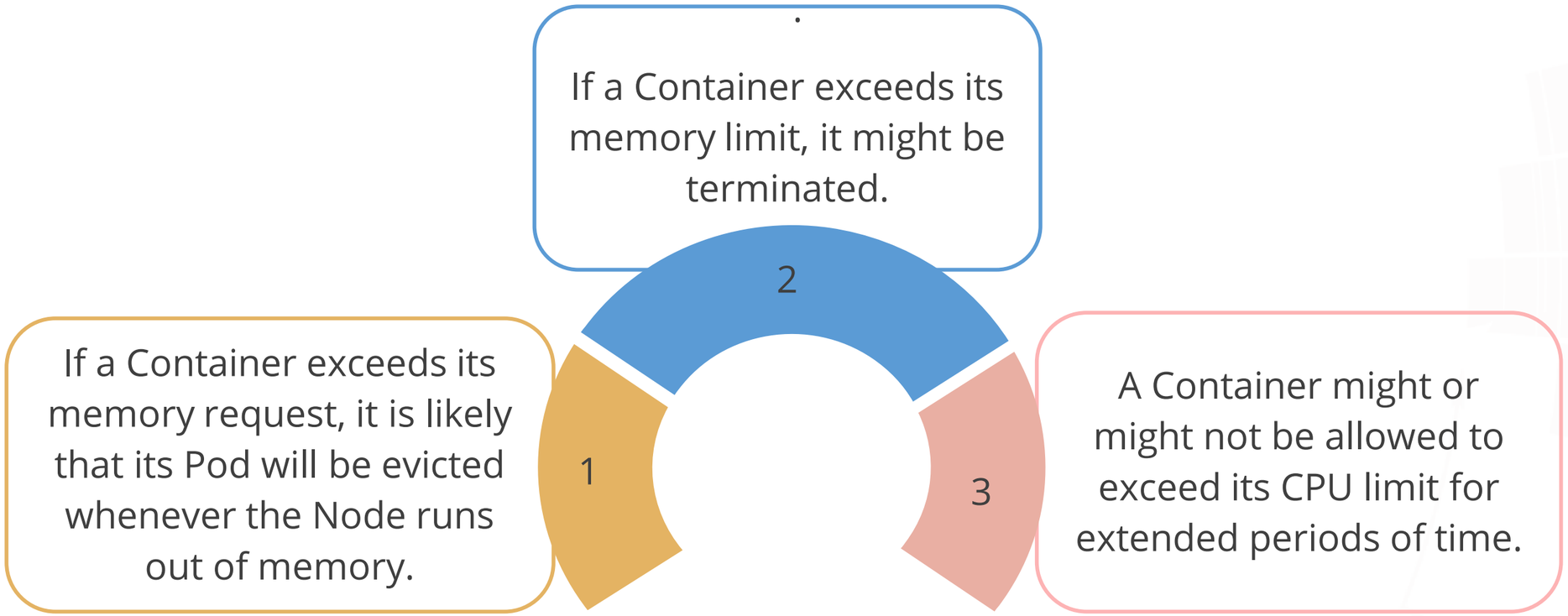
The following Pod has two Containers. Each container has a request of 0.25 cpu and 64MiB (226 bytes) of memory. Each container has a limit of 0.5 cpu and 128MiB of memory.

Demo

```
apiVersion: v1
Kind:  pod
Metadata:
   name:  Frontend
  spec:
    containers:
     -  name: app
        image: images.my company.example/app;V4
      resources:
            requests:
              memory: “64M1”
              cpu: ‘250M”
            limits:
              memory: “128M1”
              cpu: “500m”
     -  name: log-aggregator
        image: images.my company.example/log-aggregator;v6
        resources:
            requests:
              memory: “64M1”
              cpu: ‘250M”
            limits:
              memory: “128M1”
              cpu: “500m”
```

How Pods with Resource Limits are Run

When the kubelet starts a Container of a Pod, it passes the CPU and memory limits to the Container runtime.



Local Ephemeral Storage

Nodes have Local Ephemeral Storage, backed by locally-attached writeable devices or, sometimes, by RAM.

Kubernetes supports two ways to configure Local Ephemeral Storage on a node:

In this configuration, place all different kinds of ephemeral local data (emptyDir volumes, writeable layers, container images, logs) in one filesystem.

Single File System

Use this filesystem for other data (for example: system logs not related to Kubernetes); it can even be the root filesystem.

Two File Systems

Local Ephemeral Storage

Ephemeral storage can be used for managing Local Ephemeral Storage.

Each Container of a Pod can specify one or more of the following:

spec.containers[].resources.limits.ephemeral-storage

spec.containers[].resources.requests.ephemeral-storage

Local Ephemeral Storage

In the following example, the Pod has two Containers:

Demo

```
apiVersion: v1
Kind:  pod
Metadata:
   name:  Frontend
  spec:
    containers:
     -  name: app
        image: images.my company.example/app;V4
      resources:
            requests:
              ephemeral-storage: “2Gi”
            limits:
              ephemeral-storage: “4Gi”
              cpu: “500m”
     -  name: log-aggregator
        image: images.my company.example/log-aggregator;v6
        resources:
            requests:
              ephemeral-storage: “2Gi”
            limits:
              ephemeral-storage: “4Gi”
```

Local Ephemeral Storage

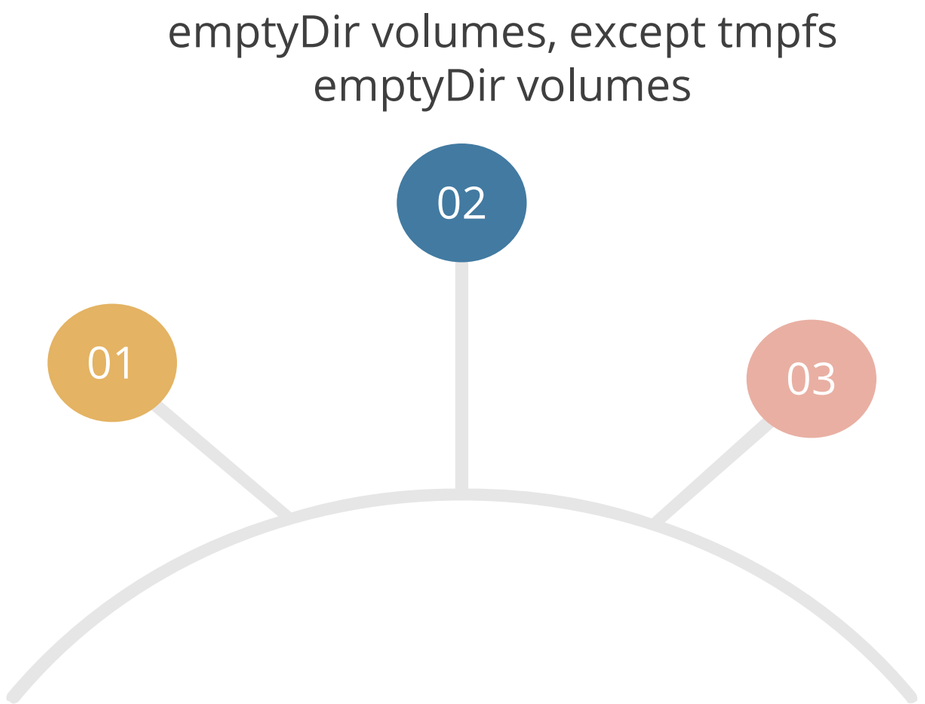
When a Pod is created, the Kubernetes Scheduler selects a Node on which the Pod will run.

The Scheduler ensures that the sum of the Resource Requests of the scheduled Containers is less than the capacity of the Node.

Ephemeral Storage Consumption Management

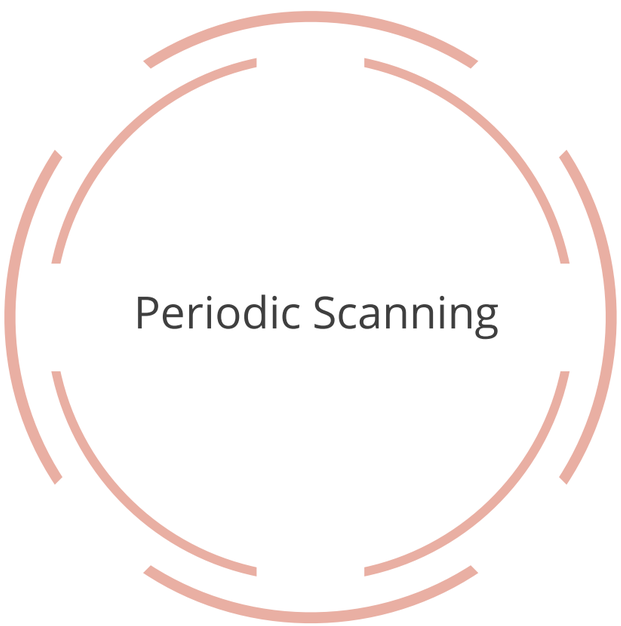
If the kubelet is managing local ephemeral storage as a resource, then it measures the storage in-use. If a Pod uses more ephemeral storage than allowed, it receives an eviction signal from kubelet.

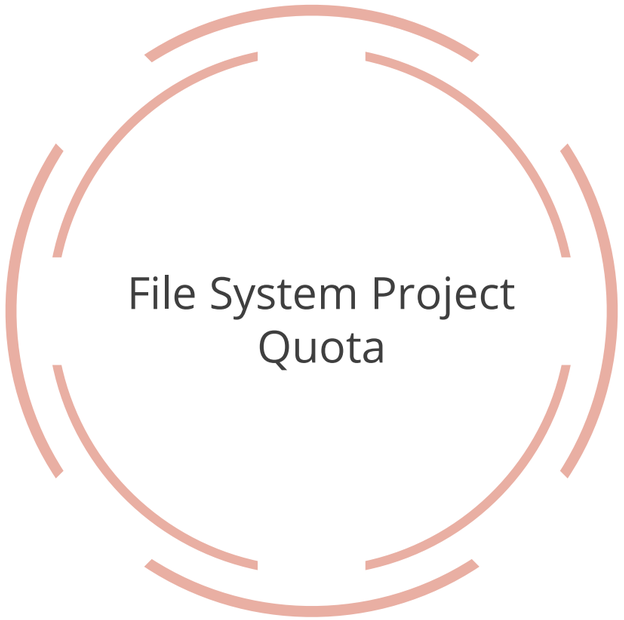
Following are the methods to manage ephemeral storage consumption:



Periodic Scanning and File System Project Quota

The kubelet supports different ways to measure Pod storage use:





Assisted Practice

Managing Container Resources

Duration: 5 mins

Problem Statement:

Learn how to manage Container resources.

Assisted Practice: Guidelines

Steps to demonstrate managing container resources in Kubernetes:

1. Add resource limits configuration entries to the Pod.
2. Open the facebookdeployment.yaml and add the following section.
3. Save the file.
4. Type ***kubectl apply -f <<name>>.yaml -n <<namespace>>***.

Health Monitoring

Overview

- Node Problem Detector collects information about Node problems from various daemons and reports these conditions to the API server as NodeCondition and Event.

Enabling Node Problem Detector

Kubectl provides the most flexible management of Node Problem Detector. For example:

Demo

```
apiVersion: apps/v1
    Kind:  DaemonSet
    Metadata:
       name:  node-problem-detector-v0.1
       namespace: kube-system
       labels:
          k8s-app:  nude-problem-detector
          version: v0.1
          kubernetes.io/cluster-service: “true”
      spec:
        selector:
         matchlabels:
          k8s-app:  nude-problem-detector
          version: v0.1
          kubernetes.io/cluster-service: “true
       template:
        metadata:
         labels:
          k8s-app:  nude-problem-detector
          version: v0.1
          kubernetes.io/cluster-service: “true”
       spec:
         hostnetwork: true
         containers:
```

Demo

```
- name: node-problem-detector
            image: k8s.gcr.io/node-problem-detector:v0.1
            securityContext:
               privileged: true
            resources:
               limits:
                 cpu: “ 200m”
                 memory:  “100M1”
               requests:
                 cpu:  “20m”
                 memory: “ 20M1”
            volumeMounts:
               - name: log
                 mountpath: /log
                 readOnly:  true
            volumes:
               - name: log
                 hostpath:
                   path:  /var/log/
```

Overwriting the Configuration

The default configuration is embedded when building the Docker image of a Node Problem Detector. Use a ConfigMap to overwrite the configuration:

Demo

```
Change the configuration files in config/
Create the ConfigMap node-problem-detector-config:
kubectl create configmap node-problem-detector-
config --from-file=config/
Change the node-problem-detector.yaml to use the
ConfigMap:
```

Overwriting the Configuration

Demo

```
apiVersion: apps/v1
    Kind:  DaemonSet
    Metadata:
       name:  node-problem-detector-v0.1
       namespace: kube-system
       labels:
          k8s-app:  nude-problem-detector
          version: v0.1
          kubernetes.io/cluster-service: “true”
      spec:
        selector:
         matchlabels:
          k8s-app:  nude-problem-detector
          version: v0.1
          kubernetes.io/cluster-service: “true
       template:
        metadata:
         labels:
          k8s-app:  nude-problem-detector
          version: v0.1
          kubernetes.io/cluster-service: “true”
       spec:
         hostnetwork: true
         containers:
          - name: node-problem-detector
            image: k8s.gcr.io/node-problem-detector:v0.1
            securityContext:
```

Demo

```
privileged: true
            resources:
               limits:
                 cpu: “ 200m”
                 memory:  “100M1”
               requests:
                 cpu:  “20m”
                 memory: “ 20M1”
            volumeMounts:
               - name: log
                 mountpath: /log
                 readOnly:  true
               - name: config # Overwrite the
config/directory with ConfigMap volume
                 mountpath: /config
                 readOnly:  true
            volumes:
               - name: log
                 hostpath:
                   path:  /var/log/
               - name: config # Define ConfigMap volume
                 configMap:
                   name: node-problem-detector-config
```

Overwriting the Configuration

Recreate the Node Problem Detector with the new configuration file:

```
# If you have a node-problem-detector running, delete before recreating
kubectl delete -f hhtps://k8s.io/examples/debug/node-problem-detector,yanl
kubectl apply -f hhtps://k8s.io/examples/debug/node-problem-detector-configmap,yanl
```

Overwriting a configuration is not supported if a Node Problem Detector runs as a cluster Addon.

Kernel Monitor

Kernel Monitor is a system log monitor daemon supported in the Node Problem Detector.

To support a new NodeCondition, create a condition definition within the conditions field in config/kernel-monitor.json, such as:

Demo

```
{
  "type": "NodeConditionType"
  "reason": "CameCaseDefaultNodeConditionReason"
  "message": "arbitrary Default node condition message"
}
```

Overwriting the Configuration

To detect new problems, extend the rules field in config/kernel-monitor.json with a new rule definition:

Demo

```
{
  "type": "temporary/permanent"
  "condition": "NodeConditionOfPermanentIssue"
  "reason": "CameCaseShortReason"
  "message": "regexp matching the issue in the kernel log"
}
```

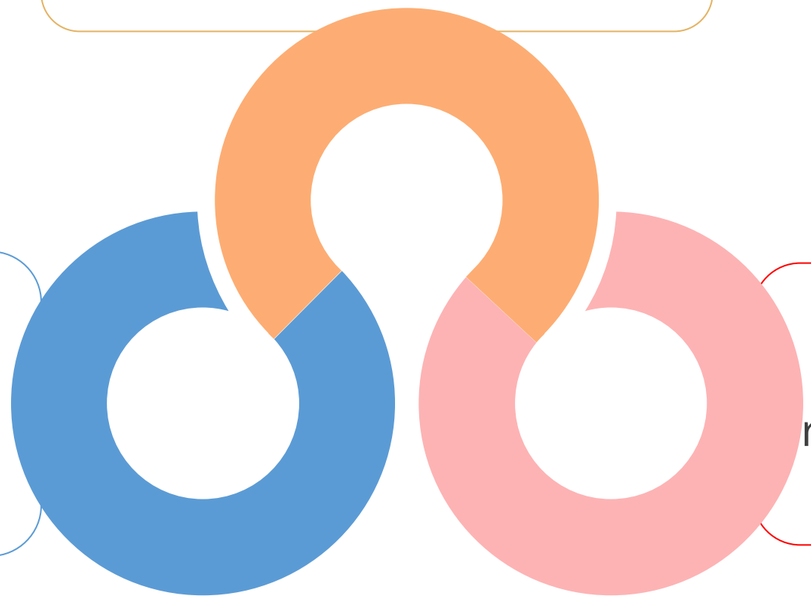
If Kernel monitor uses the translator plugin to translate the internal data structure of the kernel log. You can implement a new translator for a new log format.

Recommendations and Restrictions

When running the Node Problem Detector, expect an extra resource overhead on each node.

The kernel log grows relatively slowly.

A resource limit is set for the Node Problem Detector.



Even under high load, the resource usage is acceptable.

Assisted Practice

Understanding Health Monitoring

Duration: 5 mins

Problem Statement:

Learn about Health Monitoring in Kubernetes.

Assisted Practice: Guidelines

Steps to demonstrate Health Monitoring in Kubernetes:

1. Create a DaemonSet that monitors health.
2. Create a DaemonSet yaml file.
3. Type ***sudo kubectl apply –f***.
4. Type ***sudo kubectl get pods -A -o wide***.

Multi-Container Pods

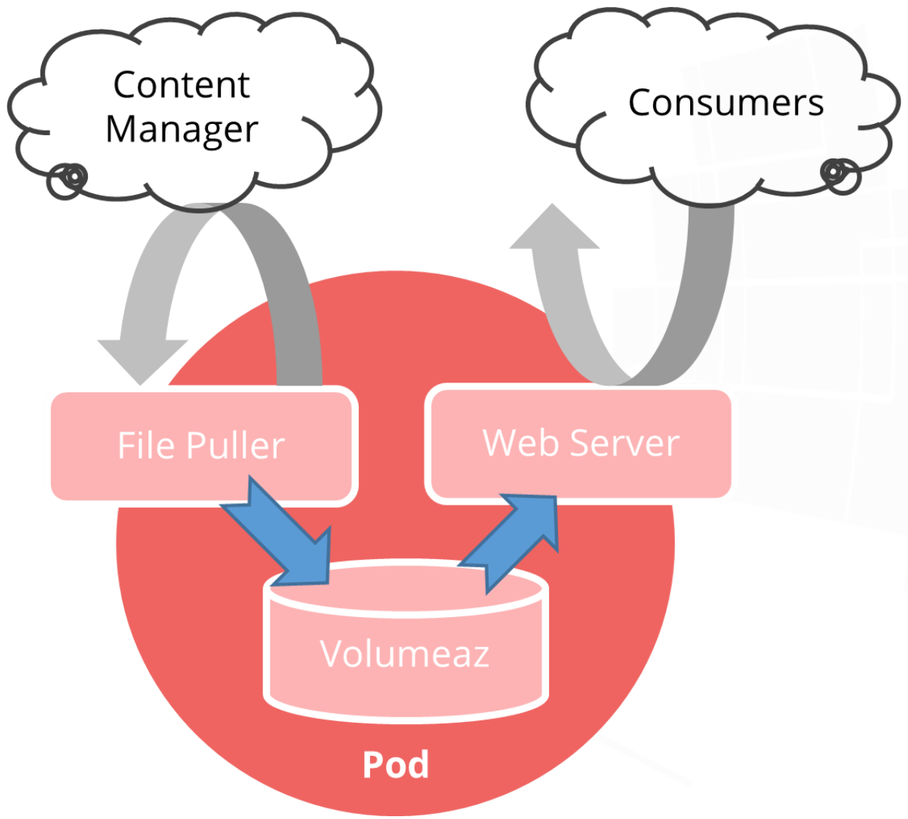
Using Pods in Different Situations

Pods are created using Workload resources such as Deployment or Job. Pods in a Kubernetes cluster are used in two main ways:

Using Pods in Different Situations

Pods are designed to support multiple cooperating processes (as Containers) that form a cohesive unit of service.

There can be a Container that acts as a web server for files in a shared volume, and a separate **sidecar** Container that updates those files from a remote source, as shown:



Assisted Practice

Creating Multi-Container Pods

Duration: 10 mins

Problem Statement:

Learn to create Multi-Container Pods.

Assisted Practice: Guidelines

Steps to demonstrate Multi-Pod Containers in Kubernetes:

1. Create a Multi-Container Pod using a configuration file.
2. Check the status.

Pod Topology Spread Constraints

Introduction

Topology spread constraints is used to control the Pods that are spread across the cluster, among failure-domains such as regions, zones, nodes, and other user-defined topology domains.

It helps achieve high availability as well as efficient resource utilization.

Prerequisites

Topology spread constraints rely on Node labels to identify the Topology domain(s) that each Node is in. For example, a Node might have labels:

```
node=node1,zone=us-east-1a,region=us-east-1
```

Consider a 4-node cluster with the following labels:

NAME	STATUS	ROLES	AGE	VERSIONS	LABELS
Node1	Ready	<none>	4m26s	v1.16.0	node=node1,zone=zoneA
Node2	Ready	<none>	3m58s	v1.16.0	node=node2,zone=zoneA
Node3	Ready	<none>	3m17s	v1.16.0	node=node3,zone=zoneB
Node4	Ready	<none>	2m43s	v1.16.0	node=node4,zone=zoneB

Spread Constraints for Pods

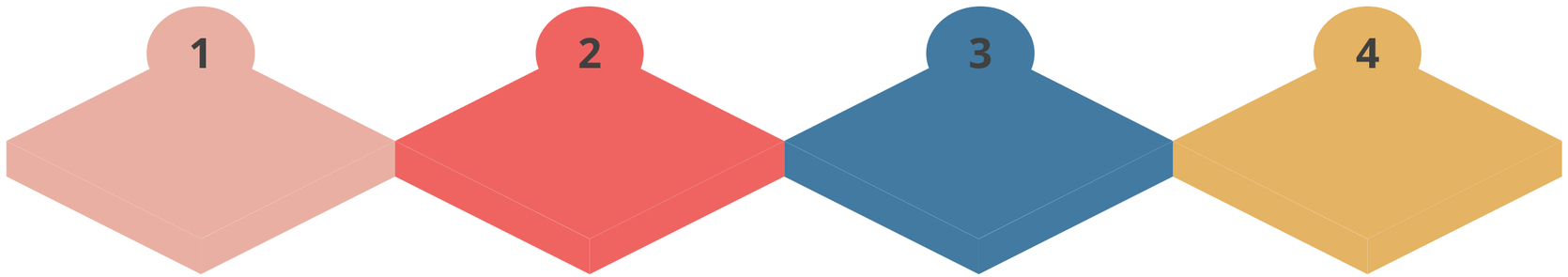
The API field **pod.spec.topologySpreadConstraints** is defined as below:

Demo

```
apiVersion: v1
Kind:  pod
Metadata:
   name:  Mypod
spec:
   topologySpreadConstraints:
      maxSkew:    <integer>
     Topology: <string>
     whenUnsatisfiable: <String>
     labelSelector: <Object>
```

Spread Constraints for Pods

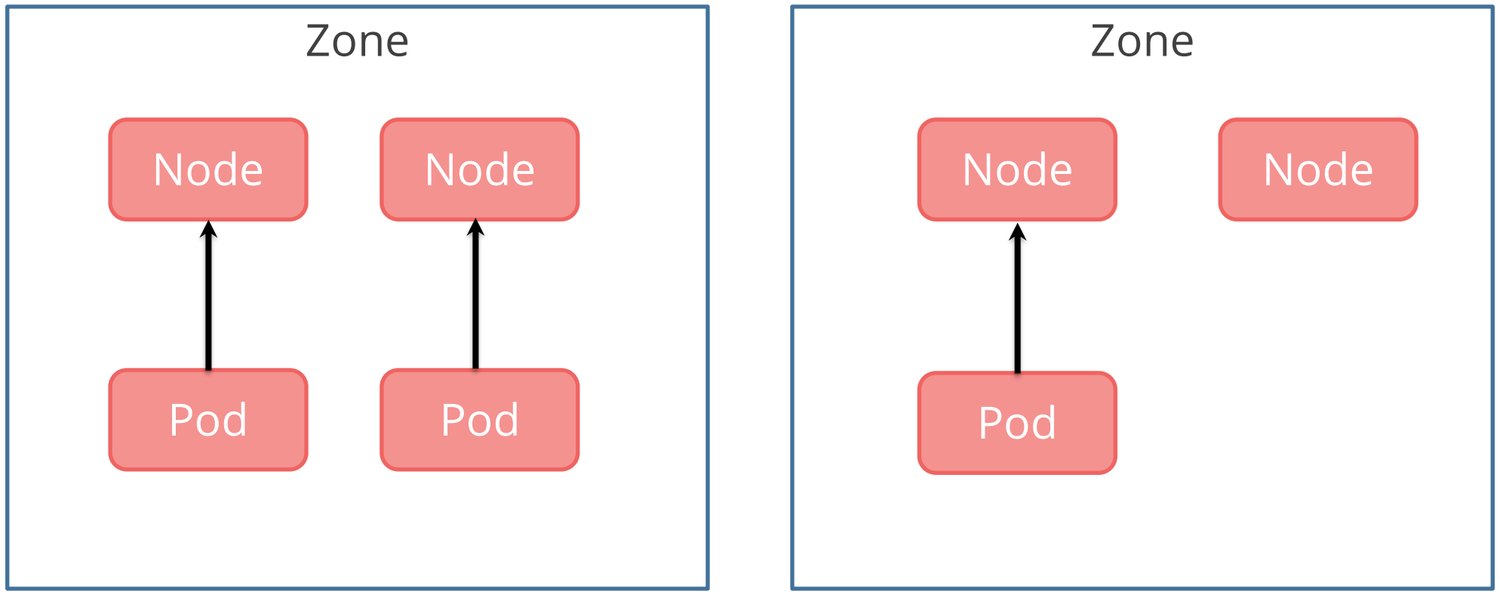
One or multiple topologySpreadConstraint can be defined to instruct the kube-scheduler on how to place each incoming Pod in relation to the existing Pods across the cluster. The fields are:



maxSkew topologyKey whenUnsatisfiable labelSelector

Example: One TopologySpreadConstraint

Consider a 4-node cluster where three Pods labeled foo:bar are located in node1, node2, and node3 respectively:



Example: One TopologySpreadConstraint

For an incoming Pod to be evenly spread with existing Pods across zones, the spec can be given as:

Demo

```
Kind:  Pod
apiVersion: v1
    Metadata:
       name:  Mypod
         labels:
         foo: bar
    spec:
       topologySpreadConstraints:
          maxSkew:   1
         Topology: zone
         whenUnsatisfiable: DoNotSchedule
         labelSelector:
         matchLabels:
         foo:
         container:
         name: pause
         image: K8s.gcr.io/pause:3.1
```

Example: One TopologySpreadConstraint

The Pod spec can be created to meet various kinds of requirements:

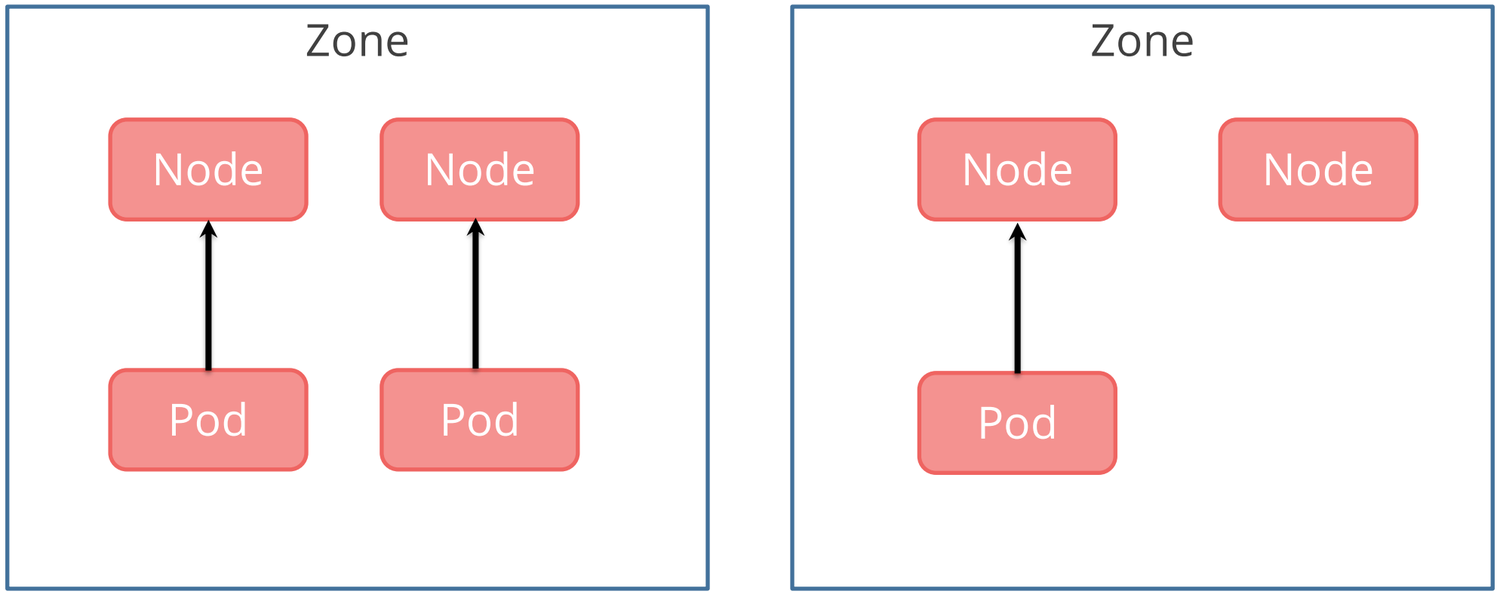
Change maxSkew to a bigger value like **2** so that the incoming Pod can be placed on **zoneA** as well.

Change topologyKey to **node** to distribute the Pods evenly across Nodes instead of zones.

Change whenUnsatisfiable: DoNotSchedule to whenUnsatisfiable: ScheduleAnyway to ensure that the incoming Pod is always schedulable.

Multiple TopologySpreadConstraints

This builds upon the previous example. Consider a 4-node cluster where three Pods labeled foo:bar are located in node1, node2, and node3 respectively:



Multiple TopologySpreadConstraints

Use two TopologySpreadConstraints to control the Pods spreading on both zone and Node:

Demo

```
Kind:  pod
apiVersion: v1
    Metadata:
       name:  Mypod
         labels:
         foo: bar
    spec:
       topologySpreadConstraints:
          maxSkew:   1
         Topology: zone
         whenUnsatisfiable: DoNotSchedule
         labelSelector:
         matchLabels:
         foo: bar
          maxSkew: node
         whenUnsatisfiable: DoNotSchedule
         labelSelector:
         matchLabels:
         foo: bar
         container:
         name: pause
         image: K8s.gcr.io/pause:3.1
```

Conventions

There are some implicit conventions worth noting here:

- Only the Pods holding the same namespace as the incoming Pod can be matching candidates.
- Nodes without topologySpreadConstraints[*].topologyKey present will be bypassed.
- Be aware of what will happen if the incomingPod's topologySpreadConstraints[*].labelSelector doesn't match its own labels.
- If the incoming Pod has spec.nodeSelector or spec.affinity.nodeAffinity defined, Nodes not matching them will be bypassed.

Cluster-level Default Constraints

It is possible to set default Topology Spread Constraints for a cluster. Default Topology Spread Constraints are applied to a Pod if, and only if:

1. It doesn't define any constraint in its .spec.topologySpreadConstraints.

2. It belongs to a Service, Replication Controller, Replica Set or Stateful Set.

Cluster-level Default Constraints

Configuration example:

Demo

```
apiVersion: kubescheduler.config.k8s.io/v1betal
Kind:  kubeschedulerconfiguration
Profile:
Pluginconfig:
       name: podTopologySpread
         args:
         defaultConstraints:
           maxSkew:   1
         TopologyKey: topology.kubernetes.io/zone
         whenUnsatisfiable: ScheduleAnyway
         defaultingType: List
```

Internal Default Constraints

With the DefaultpodTopologySpread feature gate, enabled by default, the legacy SelectorSpread plugin is disabled. Kube-scheduler uses the following default Topology Constraints for the podTopologySpread plugin configuration:

Demo

```
defaultConstraints:
          maxSkew:    3
        TopologyKey: “kubernetes.io/hostname”
        whenUnsatisfiable: ScheduleAnyway
        maxSkew:    5
        TopologyKey: “kubernetes.io/zone”
        whenUnsatisfiable: ScheduleAnyway
```

Comparison with PodAffinity and PodAntiAffinity

In Kubernetes, directives related to Affinity control how Pods are scheduled, more packed, or more scattered.

Assisted Practice

Understanding Pod Topology Spread

Duration: 10 mins

Problem Statement:

Interpret the working of Pod Topology Spread in Kubernetes.

Assisted Practice: Guidelines

Steps to demonstrate Pod Topology in Kubernetes:

1. Add Topology Spread entries to the pod configuration.
2. Open singlepod.yaml.
3. Type ***kubectl get pods -n twitter***.

ReplicaSet

Purpose of ReplicaSet

A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.

How a ReplicaSet Works

A ReplicaSet is defined with fields, including a selector that specifies how to identify Pods.

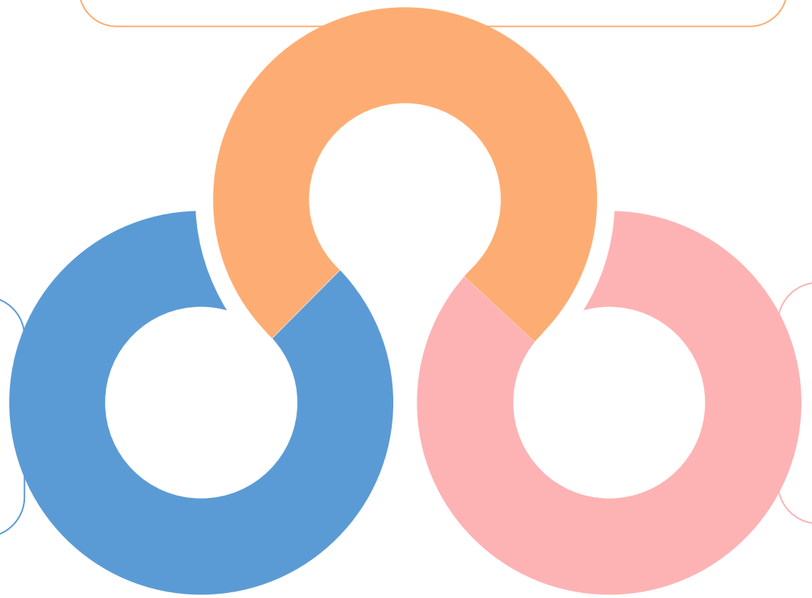
A ReplicaSet identifies new Pods to acquire by using its selector.

A ReplicaSet is linked to its Pods via the Pods' metadata.ownerReferences field, which specifies by what resource the current object is owned.

ReplicaSet in Use

A ReplicaSet ensures that a specified number of Pod replicas are running at any given time.

It provides declarative updates to Pods along with a lot of other useful features.



A Deployment is a higher level concept that manages ReplicaSets.

Deployments is recommended over directly using ReplicaSets.

Using a ReplicaSet

Example:

```
apiVersion: app/v1
Kind: ReplicaSet
Metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
Spec:
 #modify replicas according to your case
 replicas: 3
 selector:
  matchLabers:
    tier: frontend
 template:
  metadata:
   labels:
    tier: frontend
Spec:
 container:
  name: php-redis
  image: grc.io/google_samples/gb-frontened:v3
```

Using a ReplicaSet

To create the defined ReplicaSet and the Pods that it manages, save this manifest into frontend.yaml and submit it to a Kubernetes cluster.

Demo

```
kubectl apply -f
https://kubernetes.io/examples/controllers/frontend.yaml

get the current ReplicaSets deployed:
kubectl get rs

And see the frontend that is created:

NAME       DESIRED   CURRENT   READY   AGE
frontend   3         3         3       6s

Check on the state of the ReplicaSet:

kubectl describe rs/frontend
```

When to Use a ReplicaSet

Output:

```
Name: frontend
Namespace: default
Selector: tier=frontend
Labels: app=guestbook
        tier=frontend
Annotations: kubectl.kubernetes.io/last-applied-
configuration:{"apiversion":"apps/v1","kind":"ReplicaSet","metadata":{"anno
tations":{},"labels":{"app":"guestbook","tier":"frontend"},"name":"frontend
",…
Replicas: 3 current/ 3 defired
pods Status: 3 running/0 waiting/ 0 succeeded / 0 failed
pod template:
  labels: tier=frontend
  containers:
  php-redis:
  image: gcr.io/google_sample/gb-frontend:v3
  port: <none>
  Host Port: <none>
  Environment: <none>
  Mounts: <none>
  volumes: <none>
Events:
```

When to Use a Replicaset

To check for the Pods brought up, use:

```
kubectl get pods
```

The Pod information will be shown similar to:

NAME	READY	STATUS	RESTARTS	AGE
frontend-b2zdv	1/1	Running	0	6m36s
frontend-vcmts	1/1	Running	0	6m36s
frontend-wtsmm	1/1	Running	0	6m36s

When to Use a ReplicaSet

To verify that the owner reference of these Pods is set to the frontend ReplicaSet, get the yaml of one of the Pods running.

```
kubectl get Pods frontend-b2zdv -o yaml
```

The output will look similar to this, with the frontend ReplicaSet's info set in the metadata's ownerReferences field:

Demo

```
apiVersion: app/v1
Kind: pod
Metadata:
 creationTimestamp: “2020-02-12T07:06:162”
  generateName: frontend
  labels:
    tier: frontend
name: frontend-b2zdv
Namespace: default
 ownerreferences:
 apiVersion: apps/v1
 blockOwenersDeletion: true
 controller: true
 kind: replicaSet
 name: frontend
 uid: f391f6db-bb9b-4c09-ae74-6a1f77f3d5cf
```

Non-template Pod Acquisitions

To create bare Pods with no problems:

It is strongly recommended to make sure that bare Pods do not have labels which match the selector of one of the ReplicaSets.

The reason for this is because a ReplicaSet is not limited to owning Pods specified by its template. It can acquire other Pods in the manner specified in the previous sections.

Non-Template Pod Acquisitions

As those Pods do not have a controller (or any object) as their owner reference and match the selector of the frontend ReplicaSet, they will immediately be acquired by it.

Suppose the Pods are created after the frontend ReplicaSet has been deployed and has set up its initial Pod Replicas to fulfill its Replica count requirement:

```
kubectl apply -f https://kubernetes.io/examples/pods/pod-rs.yaml
```

The new Pods will be acquired by the ReplicaSet and immediately terminated as the ReplicaSet would be over its desired count.

Non-Template Pod Acquisitions

Fetching the Pods:

```
kubectl get Pods
```

The output shows that the new Pods are either already terminated or in the process of being terminated:

NAME	READY	STATUS	RESTARTS	AGE
frontend-b2zdv	1/1	Running	0	10m
frontend-vcmts	1/1	Running	0	10m
frontend-wtsmm	1/1	Running	0	10m
Pod1	0/1	Terminating	0	1S
Pod2	0/1	Terminating	0	1S

Non-template Pod Acquisitions

Create the Pods:

```
kubectl apply -f https://kubernetes.io/examples/pods/pod-rs.yaml
```

And then create the ReplicaSet:

```
kubectl apply -f
https://kubernetes.io/examples/controllers/frontend.yaml
```

Check that the ReplicaSet has acquired the Pods and has only created new ones according to its spec, until the number of its new Pods and the original match its desired count. For fetching the Pods:

```
kubectl get pods
```

Non-template Pod Acquisitions

Output:

NAME	READY	STATUS	RESTARTS	AGE
frontend-hmmj2	1/1	Running	0	9s
pod1	1/1	Running	0	36S
pod2	1/1	Running	0	36S

Writing a ReplicaSet Manifest

As with all other Kubernetes API objects, a ReplicaSet needs the apiVersion, kind, and metadata fields.

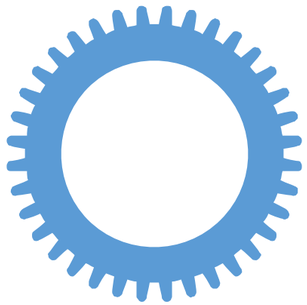
In Kubernetes 1.9, the API version apps/v1 on the ReplicaSet kind is the current version and is enabled by default.

The API version apps/v1beta2 is deprecated. Refer to the first lines of the frontend.yaml example for guidance.

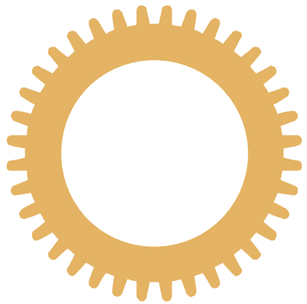
The name of a ReplicaSet object must be a valid DNS subdomain name.

A ReplicaSet also needs a .spec section.

Pod Template



The .spec.template is a Pod template which is also required to have labels in place.



For the template's Restart Policy field, .spec.template.spec.restartPolicy, the only allowed value, is Always, which is the default.

Pod Selector

1. The .spec.selector field is a label selector. These are the labels used to identify potential Pods to acquire.

2. In the ReplicaSet, .spec.template.metadata.labels must match spec.selector, or it will be rejected by the API.

Replicas

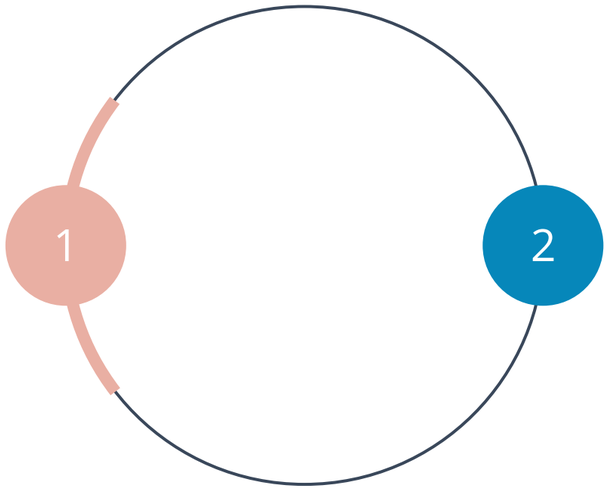
Specify the number of Pods that should run concurrently by setting .spec.replicas. The ReplicaSet will create or delete its Pods to match this number.

If .spec.replicas is not specified, then it defaults to 1.

Working with ReplicaSets

Deleting a ReplicaSet and its Pods:

To delete a ReplicaSet and all its Pods, use kubectl delete. The Garbage Collector will automatically delete all the dependent Pods by default.



When using the REST API or the client-go library, ensure that propagationPolicy is set to Background or Foreground in the -d option.

Working with ReplicaSets

Example:

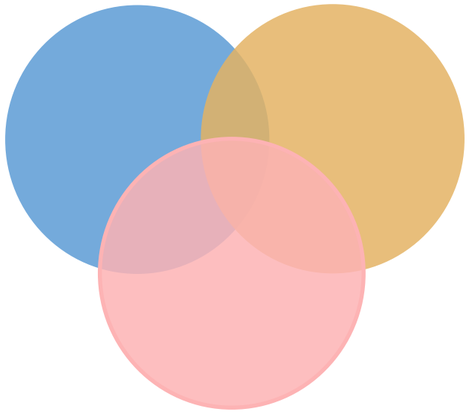
Demo

```
Kubectl proxy -port=800
Curl -x DELETE 'localhost:8080/apis/apps/v1/namespaces/default/replicasets/frontend'\
>-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationpolicy":"Foreground"}'\
> -H "content-type: application/json
```

Working with ReplicaSets

Isolating Pods from a ReplicaSet:

Pods can be removed from a ReplicaSet by changing their labels.



This technique may be used to remove Pods from Service for debugging, data recovery, etc.

Pods that are removed in this way will be replaced automatically.

Working with ReplicaSets

A ReplicaSet can be easily scaled up or down by simply updating the .spec.replicas field. When scaling down, the ReplicaSet controller chooses which Pods to delete by sorting the available Pods to prioritize scaling down Pods, based on the following general algorithm:

Pending (and unschedulable) Pods are scaled down first.

If the Pods' creation times differ, the Pod that was created more recently comes before the older Pod.

If controller.kubernetes.io/Pod-deletion-cost annotation is set, then the Pod with the lower value will come first.

Pods on Nodes with more Replicas come before Pods on Nodes with fewer Replicas.

ReplicaSet as a Horizontal Pod Autoscaler Target

A ReplicaSet can also be a target for Horizontal Pod Autoscalers (HPA). That is, a ReplicaSet can be auto-scaled by an HPA. Here is an example of HPA targeting the ReplicaSet:

Demo

```
Apiversion: autoscaling/v1
Kind: horizontalpodAutoscaler
Metadata:
        name: fronted-scalar
Spec:
        scaleTargetRef:
        kind: ReplicaSet
        name: frontend
        minReplicas: 3
        maxReplicas: 10
        targetCPUUtilizationpercentage: 50
```

ReplicaSet as a Horizontal Pod Autoscaler Target

Save this manifest into hpa-rs.yaml and submit it to a Kubernetes cluster to create the defined HPA that autoscales the target ReplicaSet, depending on the CPU usage of the replicated Pods.

```
kubectl apply -f https://k8s.io/examples/controllers/hpa-rs.yaml
```

Alternatively, use the kubectl autoscale command to accomplish the same:

```
kubectl autoscale rs frontend --max=10 --min=3 --cpu-percent=50
```

Alternatives to ReplicaSet

Deployment

Bare Pods

Job

DaemonSet

ReplicationController

Assisted Practice

Understanding ReplicaSets

Duration: 5 mins

Problem Statement:

Understand the creation and working of ReplicaSets in Kubernetes.

Assisted Practice: Guidelines

Steps to demonstrate ReplicaSets in Kubernetes:

1. Create Replica Sets.
2. Create a replica set configuration.
3. Type ***sudo kubectl apply -f singlereplica.yaml -n twitter***.
4. Type ***sudo kubectl get pods -n twitter***.
5. Type ***sudo kubectl get rs -n twitter***.

Static Pods

Overview

- Static Pods are managed directly by the kubelet daemon on a specific Node, without the API server observing them.
- Static Pods are always bound to one Kubelet on a specific Node.
- The main use of Static Pods is to run a self-hosted Control Plane.
- The kubelet automatically tries to create a mirror Pod on the Kubernetes API server for each Static Pod.
- The Pods running on a Node are visible on the API server but cannot be controlled from there.

Assisted Practice

Understanding Static Pods

Duration: 5 mins

Problem Statement:

Learn about Static Pods in Kubernetes.

Assisted Practice: Guidelines

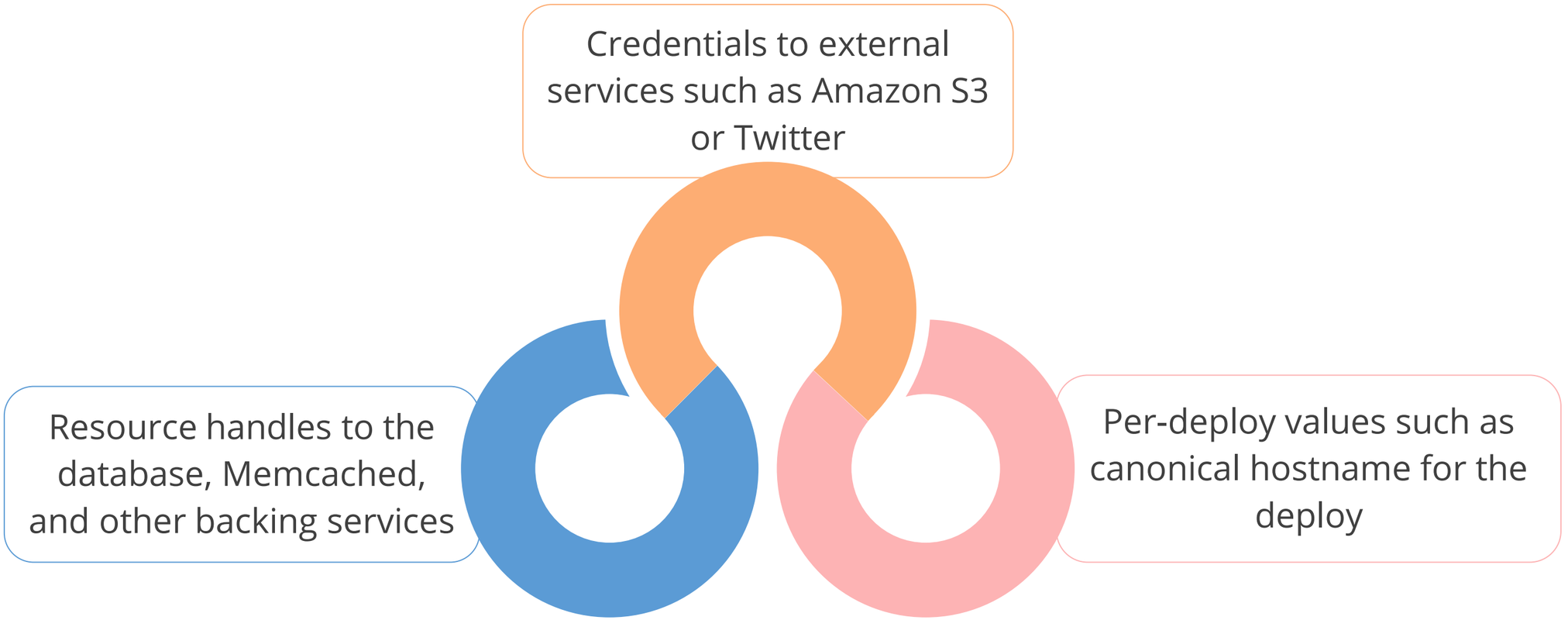
Steps to demonstrate Static Pods in Kubernetes:

1. Create Static Pod in worker node.
2. Choose one of the worker nodes.
3. Go to the directory.
4. Type sudo kubectl apply -f singlereplica.yaml -n twitter.
5. Type sudo kubectl get pods -n twitter.
6. Type sudo kubectl get rs -n twitter.

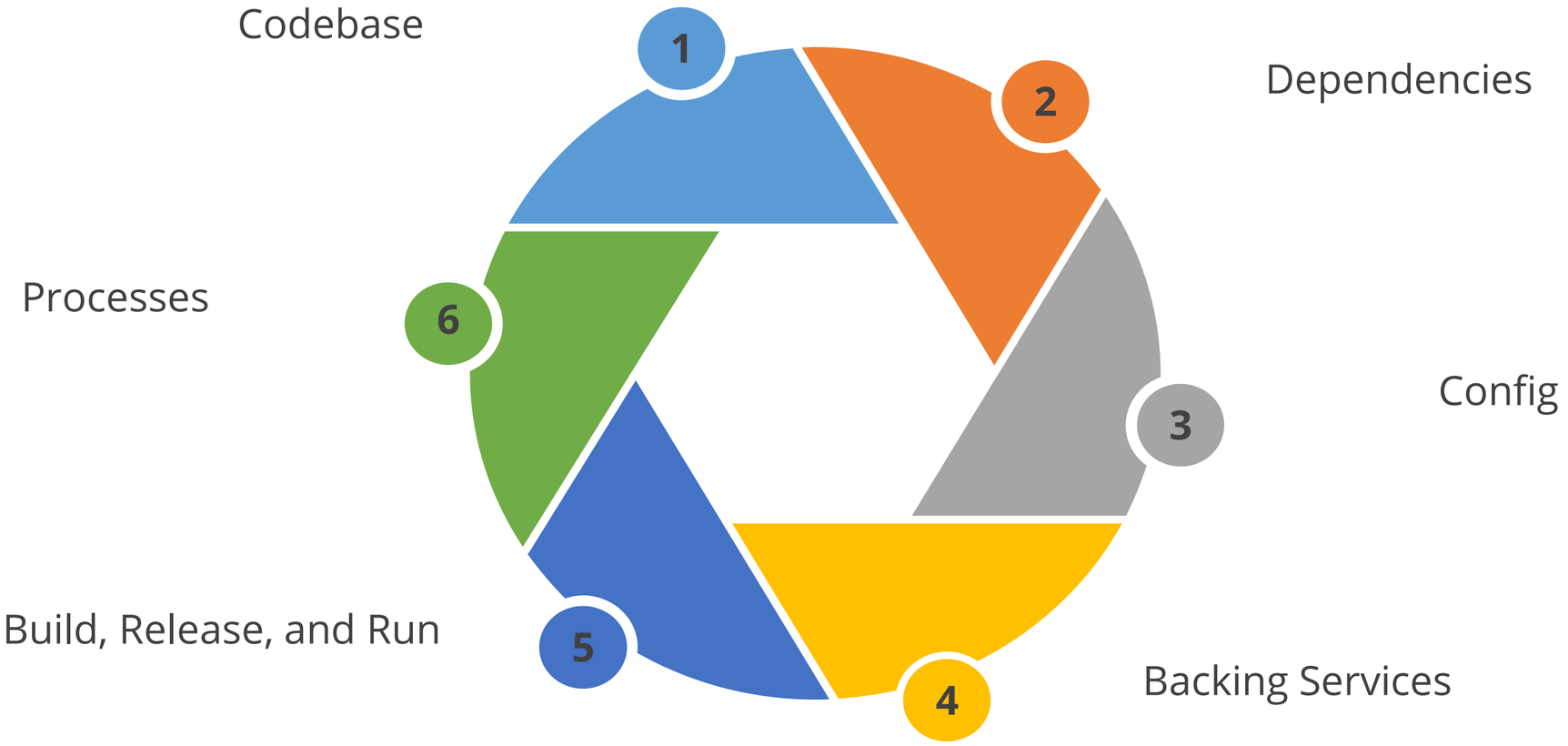
Application Configuration

Overview

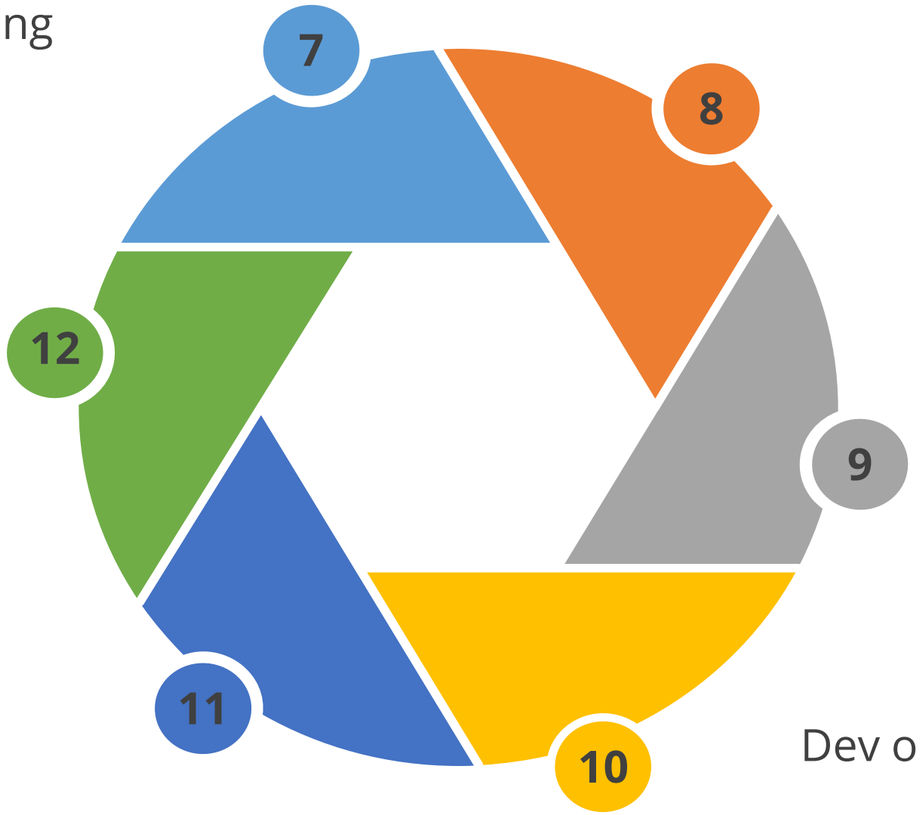
An application’s configuration is everything that is likely to vary between deploys (staging, production, developer environments, etc.). This includes:



12-Factor Principles



12-Factor Principles



Port Binding

Concurrency

Disposability

Dev or Prod Parity

Logs

Admin Processes

A Typical Way for Configuration in Kubernetes

To configure your apps in Kubernetes, use:

Environment Variables for Configuration

ConfigMap

Secret

Using Environment Variables for Configuration

Example of how to use Environment Variables by specifying them directly within the Pod specification:

Demo

```
apiVersion: v1
Kind: pod
Metadata:
        name: pod1
Spec:
        containers:
                name: nginx
                image: nginx
                env:
                        name: ENVVAR1
                        value: value 1
                        name: ENVVAR2
                        value: value2
```

Using Environment Variables for Configuration

Define two variables in spec.containers.env — **ENVVAR1** and **ENVVAR2** with values **value1** and **value2** respectively.

Use the **kubectl apply** command to submit the Pod Information to Kubernetes:

```
$ kubectl apply -f
https://raw.githubusercontent
.com/abhirockzz/kubernetes-
in-a-
nutshell/master/configuration
/kin-config-envvar-in-
pod.yaml
pod/pod1 created
```

Use grep to filter for the variable(s):

```
$ kubectl exec pod1 -it --
env | grep ENVVAR
ENVVAR1=value1
ENVVAR2=value2
```

Assisted Practice

Understanding Application Configuration

Duration: 10 mins

Problem Statement:

Learn about application configuration in Kubernetes.

Assisted Practice: Guidelines

Steps to demonstrate Application Configuration in Kubernetes:

1. Configure Application.
2. Create a Static Pod in a worker node.
3. Check in Master by typing.
4. Check whether the Container runs in Docker by typing.

ConfigMap and Secrets

ConfigMaps

A ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as Environment Variables, command-line arguments, or as configuration files in a volume.

A ConfigMap allows to decouple environment-specific configuration from the Container images so that the applications are easily portable.

A ConfigMap is not designed to hold large chunks of data. The data stored in a ConfigMap cannot exceed 1 MiB.

ConfigMap Object

A ConfigMap is an API object that lets you store configuration for other objects to use. Unlike most Kubernetes objects that have a spec, a ConfigMap has data and binaryData fields.

The data field is designed to contain UTF-8 byte sequences while the binaryData field is designed to contain binary data as base64-encoded strings.

Each key under the data or the binaryData field must consist of alphanumeric characters, -, _ or .. These keys that are stored in the data must not overlap with the keys in the binaryData field.

ConfigMaps and Pods

Any user can create a Pod specification that refers to a ConfigMap. The spec can configure the Container or Containers in that particular Pod, based on the ConfigMap data. The Pod and the ConfigMap must be in the same namespace.

Here is a sample ConfigMap that has some keys with single values:

Demo

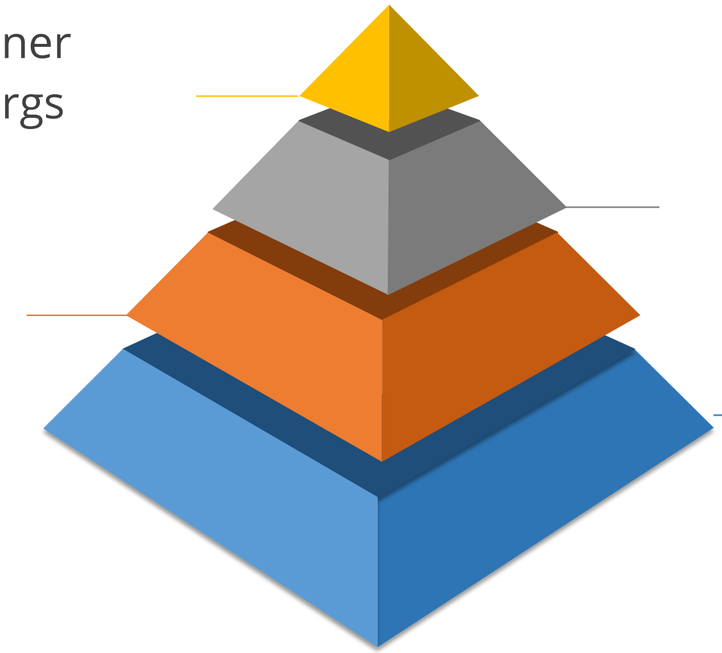
```
Kind: configmap
Metadata:
        name: game-demo
data:
        # property-like keys; each key map to a simple value
        player_initial_lives: 3
        ui_properties_file_name: "user-interface.properties"

        #file-like keys
        game.properties:
        enemy.types=aliens, monster
        players.maximum-lives=5
        user-interface.properties::
        color.good=purple1
        color.bad=yellow
        allow.textmode=true
```

ConfigMaps and Pods

There are four different ways that a ConfigMap is used to configure a Container inside a Pod:

Is inside a container command and args

Add a file in read-only volume, for the application to read



There are Environment Variables for a container

Write code to run inside the Pod that uses the Kubernetes API to read a ConfigMap

Using ConfigMaps

ConfigMaps can be mounted as data volumes. They may also be used by other parts of the system, without having direct exposure to the Pod.

The most common way to use ConfigMaps is to configure settings for Containers running in a Pod in the same namespace. A ConfigMap can be used separately.

Using ConfigMaps

To use ConfigMaps as files from a Pod to consume a ConfigMap in a volume in a Pod:

Create a ConfigMap or use an existing one.

Modify the Pod definition to add a volume under .spec.volumes[].

Add a .spec.containers[].volumeMounts[] to each Container that needs the ConfigMap.

Modify the image or command line so that the program looks for files in that directory.

Using ConfigMaps

Each ConfigMap used needs to be referred to in .spec.volumes.

If there are multiple Containers in the Pod, then each Container needs its own volumeMounts block, but only one .spec.volumes is needed per ConfigMap.

Mounted ConfigMaps are updated automatically when a ConfigMap currently consumed in a volume is updated or projected keys are updated.

The kubelet checks whether the mounted ConfigMap is fresh on every periodic sync. However, the kubelet uses its local cache for getting the current value of the ConfigMap.

Immutable ConfigMaps

The Kubernetes feature Immutable Secrets and ConfigMaps provides an option to set individual Secrets and ConfigMaps as immutable. For clusters that extensively use ConfigMaps (at least tens of thousands of unique ConfigMap to Pod mounts), preventing changes to their data has the following advantages:

Protects from accidental (or unwanted) updates that could cause applications outages

Improves performance of the cluster by significantly reducing the load on kube-apiserver and closing watches for ConfigMaps marked as immutable

Immutable ConfigMaps

This feature is controlled by the ImmutableEphemeralVolumes feature gate. An immutable ConfigMap can be created by setting the immutable field to true.

Demo

```
apiVersion: v1
Kind: ConfigMap
Metadata:
          ….
Data:
          ….
Immutable: true
```

Once a ConfigMap is marked as immutable, it is not possible to revert the change, or mutate the contents of the data or the binaryData field.

Secrets

Kubernetes Secrets help store and manage sensitive information such as passwords, OAuth tokens, and SSH keys.

To use a Secret, a Pod needs to reference the Secret. A Secret can be used with a Pod in three ways:

As files in a volume mounted on one or more of its Containers

As Container Environment Variables

When images are pulled for the Pod by the kubelet

Types of Secrets

When creating a Secret, specify its type using the type field of the Secret resource, or certain equivalent kubectl command line flags. These types vary in terms of the validations performed and the constraints Kubernetes imposes on them:

Builtin Type	Usage
Opaque	arbitrary user-defined data
kubernetes.io/service-account-token	service account token
kubernetes.io/dockercfg	serialized ~/.dockercfg file
kubernetes.io/dockerconfigjson	serialized ~/.docker/config.json file
kubernetes.io/basic-auth	credentials for basic authentication
kubernetes.io/ssh-auth	credentials for SSH authentication
kubernetes.io/tls	data for a TLS client or server
bootstrap.kubernetes.io/token	bootstrap token data

Types of Secrets

Kubernetes doesn't impose any constraints on the type name. However, if a built-in type is being used, all the requirements defined for that type must be met.

- Opaque Secrets
- SSH Authentication Secrets
- Service Account Token Secrets
- TLS Secrets
- Docker Config Secrets
- Bootstrap Token Secrets
- Basic Authentication Secret

Creating a Secret

There are several options to create a Secret:

01 — Use kubectl command

02 — Use config file

03 — Use kustomize

Editing a Secret

Demo

```
# please edit the object below. Lines beginning with a # will be
ignored,and an empty file will abort the edit. If an error occurs
while saving this file will be
#reopened with the relevant failure

apiVersion: v1
data:
        username: YWRtaW4=
        password: MWYyZDFlMmU2NRm
Kind: Secret
Metadata:
        annotations: game-demo
        kubectl.kubernetes.io/last-applied-configuration: {..}
        creationTimestamp: 2016-01-22T18:41:56Z
name: mysecret
        namespace: default
        resourceVersion: "164619"
        uid: cfee02d6-c137-11e5-8d73-42010af00002
Type: Opaque
```

An existing Secret may be edited with the command shown.

kubectl edit secrets mysecret

This will open the default configured editor and allow for updating the base64 encoded Secret values in the data field.

How to Use Secrets

Mount Secrets as data volumes or expose them as Environment Variables that can be used by a Container in a Pod. Other parts of the system can also use Secrets, without being directly exposed to the Pod.

Use a pre-existing Secret or create a new one.

Change the Pod definition to add a volume under .spec.volumes[].

Specify a .spec.containers[].volumeMounts[] to each Container that needs the Secret.

Alter the command line or image so that the program looks for files in the specific directory.

Usage of Secrets as Environment Variables

To use a Secret in a Pod's Environment Variable:

Create a Secret or use an existing one. Multiple Pods can reference the same Secret.

Modify the image and/or command line so that the program looks for values in the specified environment variables.

Modify the Pod definition in each container. The Environment Variable that consumes the Secret key should populate the Secret's name and key in `env[].valueFrom.secretKeyRef`.

Using Secrets

Example of a Pod that uses Secrets from Environment Variables:

Demo

```
apiVersion: v1
Kind: pod
Metadata:
 name: secret-env-pod
Spec:
 containers:
  name: mycontainer
  image: redis
  env:
   name: SECRET-USERNAME
   valueFrom:
    secretKeyRef:
     name: mysecret
     key: username
    name: SECRET_PASSWORD
     valueFrom:
      secretKeyRef:
       name: mysecret
       key: password
Restartpolicy: Never
```

Using Secrets

Inside a Container that consumes a Secret in the Environment Variables, the Secret keys appear as normal Environment Variables containing the baseContainer from the example:

```
echo $SECRET_USERNAME
```

```
Admin echo $SECRET_PASSWORD
```

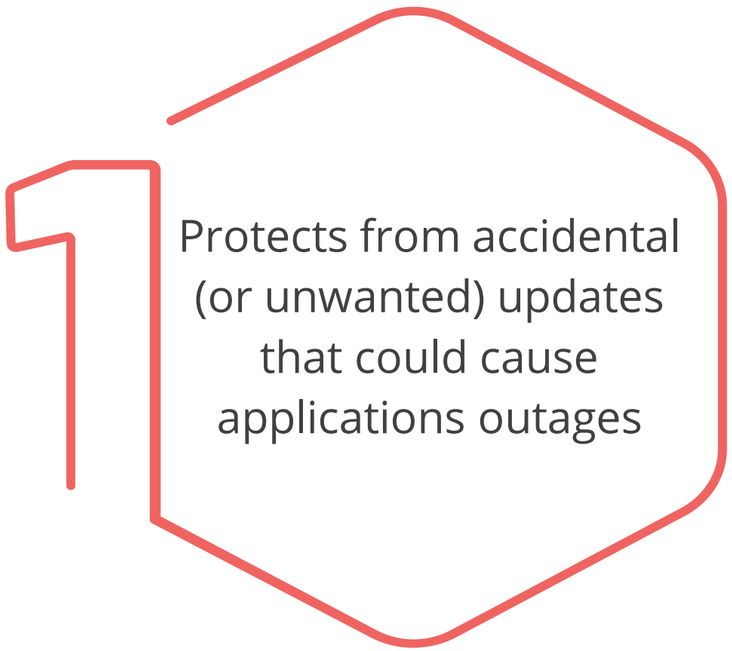
The output is similar to:

```
1f2d1e2e67df
```

Environment Variables are not updated after a Secret update.

Immutable Secrets

The Kubernetes feature Immutable Secrets and ConfigMaps provides an option to set individual Secrets and ConfigMaps as immutable:



Assisted Practice

Understanding Config Maps and Secrets

Duration: 5 mins

Problem Statement:

Understand the working of ConfigMap and Secrets in Kubernetes.

Assisted Practice: Guidelines

Steps to demonstrate ConfigMap and Secrets in Kubernetes:

1. Adding Config Map entry to the Pod.
2. Define Secrets using kubectl.

Achieving Scalability

Overview

Kubernetes has many features to help application developers ensure that their applications are deployed in a highly available, scalable, and fault-tolerant way. When deploying a horizontally scalable application to Kubernetes, ensure the following configuration:

- Resource Limits and Requests
- Deployment Strategies
- Node and Pod, Affinities and Anti-affinities
- Pod Disruption Budgets
- Health Checks
- Horizontal Pod Autoscaling

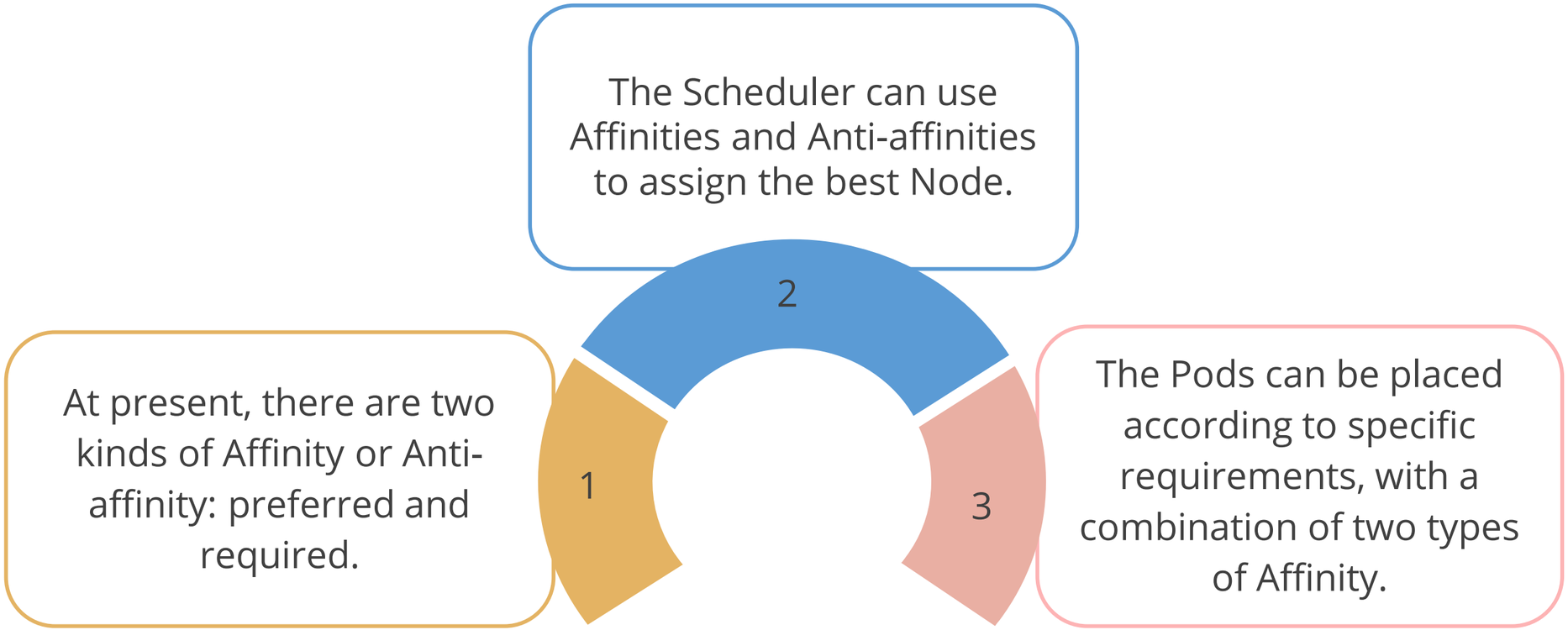
Resource Requests and Limits

Resource Requests and Limits dictate how much memory and CPU the application may use.

It is vital to set Limits and Requests on a Deployment.

If there are no Limits, the Kubernetes Scheduler cannot ensure that Workloads are spread evenly across the Nodes.

Node and Pod, Affinities and Anti-affinities



Health Checks

Kubernetes has two types of Health Checks: liveness and readiness probes.

A readiness probe tells Kubernetes that the Pod is ready to start receiving requests.

Deployment Strategies

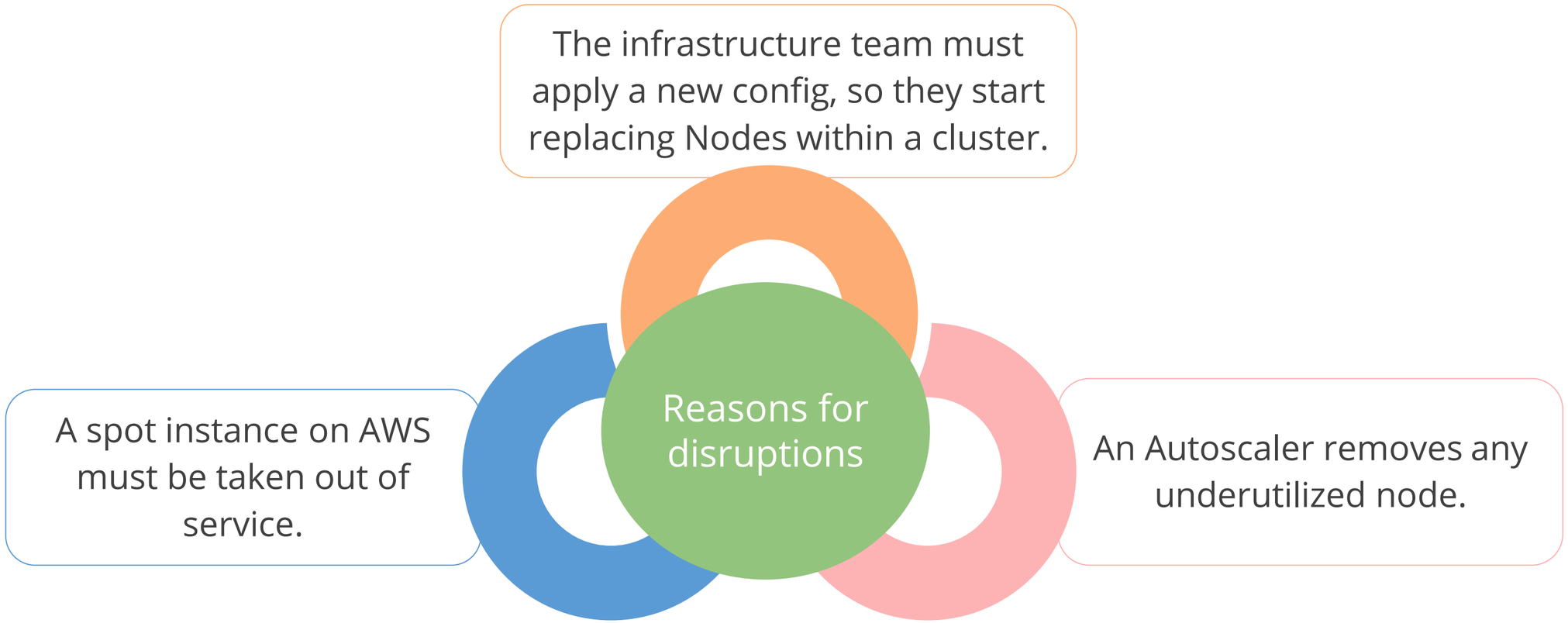
These dictate how replacement of running Pods is done on Kubernetes when configuration updates are happening.

The **recreate** strategy: This strategy kills all the Pods managed by the Deployment, prior to starting a new Pod.

The **rolling update** strategy: This strategy runs old as well as new configurations alongside each other.

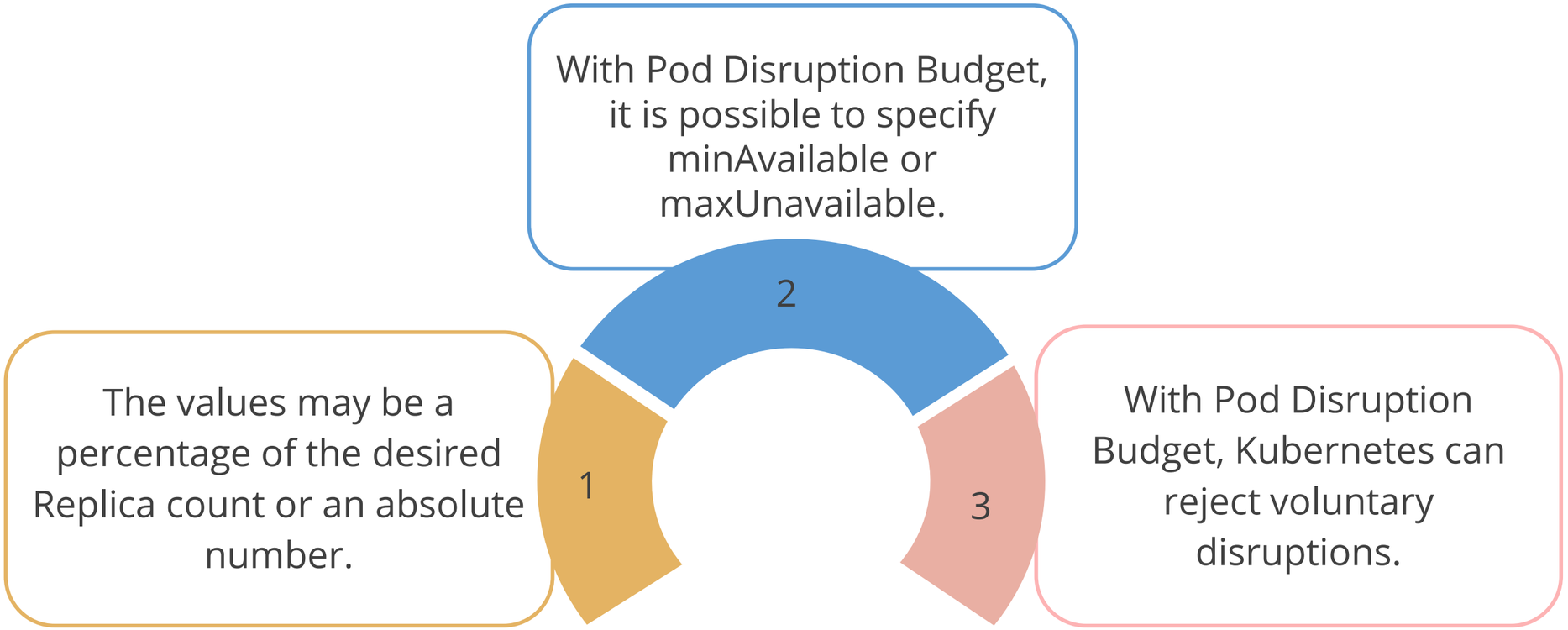
Pod Disruption Budgets

Expect disruptions in Kubernetes clusters: they are the norm, not the exception.



Pod Disruption Budgets

The job of the Pod Disruption Budget is to ensure that there is always a minimum number of ready Pods for Deployment.



Horizontal Pod Autoscalers

Kubernetes Control Plane has the capability to scale applications based on their resource utilization. As Pods become busier, the Plane brings up new Replicas to share the load automatically.

Periodically, the controller monitors metrics provided by the metrics-server to scale the deployment up or down as necessary, based on the Pods' load.

It is possible to scale when configuring a Horizontal Pod Autoscaler, based on CPU and memory usage. But custom metrics servers can help extend the metrics available.

Assisted Practice

Achieving Scalability

Duration: 5 mins

Problem Statement:

Learn how to achieve scalability in Kubernetes.

Assisted Practice: Guidelines

Steps to demonstrate Achieving Scalability in Kubernetes:

1. Modify the Pods with scalability parameters like Resources, Requests and Limits.
2. Add Resources, Requests and Limits to the Pod configuration.

Building Self-Healing Pods with Restart Policies

Introduction: No Time for Downtime!

Today, any application that does not operate 24x7 is considered inefficient. Applications are expected to run uninterrupted, even if there are technical glitches, feature updates, or natural disasters.

Kubernetes facilitates the smooth working of the application by abstracting machines physically. It is a Container orchestration tool.
The Pods and Containers in Kubernetes are self-healing.

The Three Container States-1

Waiting: Created but not running. A Container in waiting stage can still run operations such as pulling images or applying Secrets. To check the status of a waiting, use:

```
kubectl describe pod [podname]
```

Along with the Pod's state, a message and reason for the state are displayed.

```
...
State: Waiting
Reason: ErrImagePull
...
```

The Three Container States-2

Running Pods: These are Containers running sans issues. The postStart command is run before the Pod enters the running state:

```
postStart
```

Running pods display the time of starting of the Container.

```
...
State: Running
Started: Sat, 30 Jan 2021 16:48:38 +0530
...
```

The Three Container States-3

Terminated Pods: A Container that has failed or completed its execution. The preStop command is executed before the Pod is moved to be terminated:

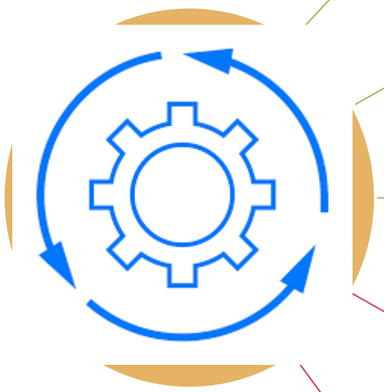
```
preStop
```

Terminated Pods will display the time of the entrance of the Container.

```
...
State: Terminated
Reason: Completed
Exit Code: 0
Started: Mon, 11 Jan 2021 12:23:45 +0530
Finished: Mon, 18 Jan 2021 21:32:54 +0530
...
```

Self-Healing in Kubernetes

The Pod phase in Kubernetes offers insights into the Pod's placement. It can have:



- Pending Pods: Pods have been created but not running
- Running Pods: Pods run all the Containers
- Succeeded Pods: Pods have completed the Container lifecycle
- Failed Pods: Have at least one Container failed, and all Containers terminated
- Unknown Pods

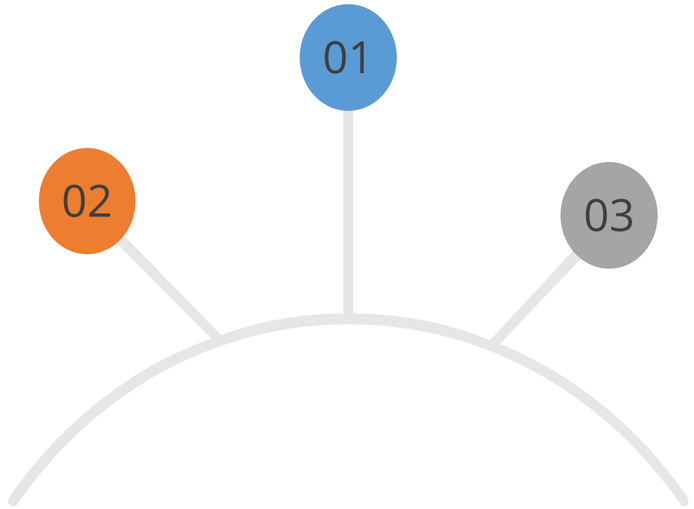
Kubernetes' Self-Healing

The probes include:

01 ExecAction to execute commands in Containers.

02 TCPSocketAction to implement a TCP check w.r.t to the IP address of a Container.

03 HTTPGetAction to implement a HTTP Get check w.r.t to the IP address of a Container.



Kubernetes' Self-Healing Concepts

Each probe provides one of three results:

Failure

The container failed the diagnostic.

Unknown

The diagnostic failed, so no action can be taken.

Success

The container passed the diagnostic.

Assisted Practice

Building Self Healing Pods with Restart Policies

Duration: 5 min

Problem Statement:

Learn to build Self-Healing Pods with Restart Policies.

Assisted Practice: Guidelines

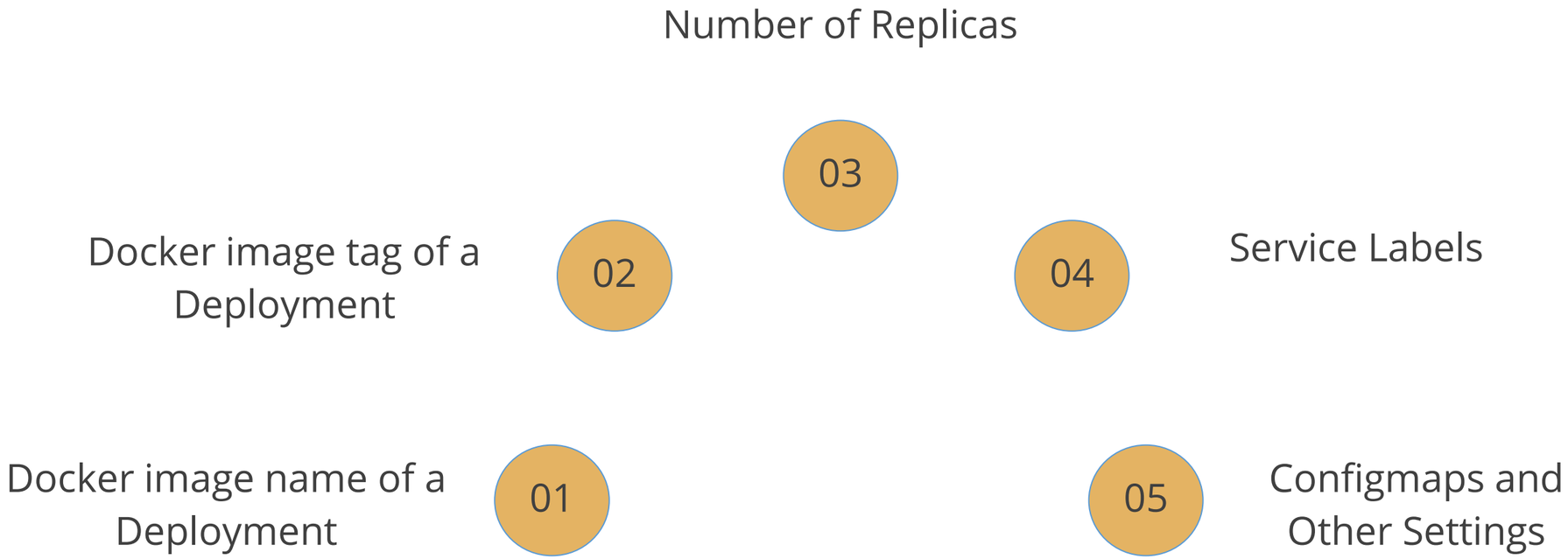
Steps to demonstrate Self-Healing Pods in Kubernetes:

1. Create a Self-Healing Pod.
2. Edit the singlepod.yaml that was created earlier by adding the configuration entry Replicas.
3. Type ***sudo kubectl apply -f singlepod.yaml -n test***.
4. Check the Pod.

Manifest Management and Common Templating Tools

Overview

There is a need to use templates in Kubernetes manifests for common parameters. These include:



Overview

Kubernetes does not have any default templating mechanism.

Deployed manifests are static yaml files.

If the parameters need to pass in the manifests, an external solution is needed.

Helm is the package manager for Kubernetes. It has templating capabilities.
Helm is the Kubernetes equivalent of **yum** or **apt**.

Helm deploys charts. It is a collection of all pre-configured, versioned application resources that may be deployed as a single unit.

Assisted Practice

Understanding Manifest Management and Common Templating Tools

Duration: 10 mins

Problem Statement:

Understand manifest management and common templating tools.

Assisted Practice: Guidelines

Steps to demonstrate Common Templating Tools in Kubernetes:

1. Install Helm.
2. Search for publicly available charts.
3. Create a sample template.

Key Takeaways

- Node Isolation is utilised to make sure only certain Pods run on Nodes with certain properties: isolation, security, or regulatory.
- Kernel Monitor is a system log monitor daemon supported in the Node Problem Detector.
- A ReplicaSet ensures that a specified number of Pod replicas are running at any given time.
- Kubernetes Control Plane can scale applications based on their resource utilization. As Pods become busier, it automatically brings up new replicas to share the load.

Thank You