# Kubernetes Objects Guide

August 11, 2022

KUBERNETES

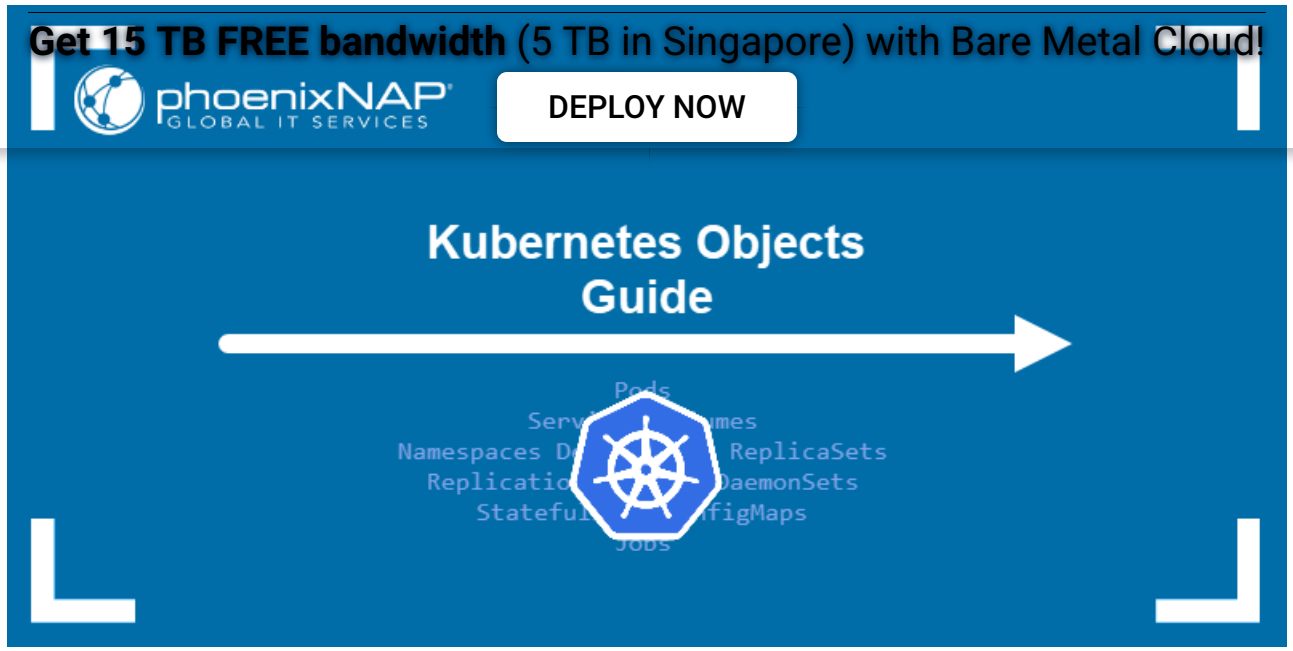


## Introduction

Container orchestration is an essential aspect of managing operational complexity in DevOps. Teams adopting Kubernetes for containerized workload management aim to improve agility and speed by utilizing the platform's declarative approach and automation features.

One of the most important ways Kubernetes administrators interact with the platform is by creating and managing Kubernetes objects. Objects are instrumental in helping users to deploy apps and maintain the cluster.

**This guide will provide a detailed overview of Kubernetes objects, analyze their structure, and offer helpful object management tips. It will also list and describe the most frequently used objects.**




# What are Kubernetes Objects?

Kubernetes objects are entities in a Kubernetes cluster that serve as a record of intent. Administrators create objects to express the cluster's desired state, and Kubernetes uses them to maintain this state automatically.

Another way to understand Kubernetes objects is by looking at them as class instances. Each created object references to a pre-defined class that tells the API server how to handle system resources and communicate with specific components.

To see the complete list of available objects in a Kubernetes cluster, type the following kubectl command:

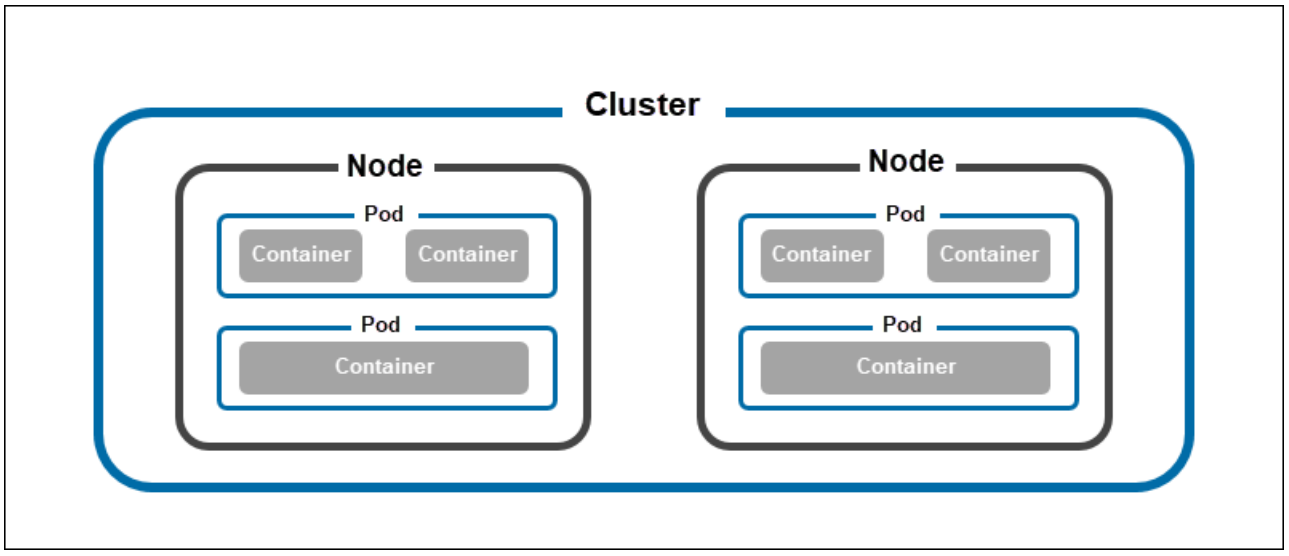
```
kubectl api-resources
```

```
marko@test-main:~$ kubectl api-resources
NAME                              SHORTNAMES   APIVERSION   NAMESPACED   KIND
bindings                                       v1           true         Binding
componentstatuses                 cs           v1           false        ComponentStatus
configmaps                        cm           v1           true         ConfigMap
endpoints                         ep           v1           true         Endpoints
events                            ev           v1           true         Event
limitranges                       limits       v1           true         LimitRange
namespaces                        ns           v1           false        Namespace
nodes                             no           v1           false        Node
persistentvolumeclaims            pvc          v1           true         PersistentVolumeClaim
persistentvolumes                 pv           v1           false        PersistentVolume
pods                              po           v1           true         Pod
podtemplates                                   v1           true         PodTemplate
replicationcontrollers            rc           v1           true         ReplicationController
resourcequotas                    quota        v1           true         ResourceQuota
secrets                                        v1           true         Secret
serviceaccounts                   sa           v1           true         ServiceAccount
services                          svc          v1           true         Service
```

The following sections explain the object concept by describing the most important Kubernetes objects.

## Pods

Kubernetes pods are the smallest unit of deployment in Kubernetes. They reside on cluster nodes and have their IP addresses, enabling them to communicate with the rest of the cluster. A single pod can host one or more containers, providing storage and networking resources.




One of the key characteristics of Kubernetes pods is that they are ephemeral. In practice, a pod can fail without impacting the system's functioning. Kubernetes automatically replaces each failed pod with a new pod replica and keeps the cluster running.

Aside from being container wrappers, pods also store configuration information that instructs Kubernetes on how to run the containers.

## Services

Services provide a way to expose a[n] pods. Their purpose is to represent a set of pods that perform the same function and set the policy for accessing those pods.

Although pod failure is an expected event in a cluster, Kubernetes replaces the failed pod with a replica with a different IP address. This creates problems in communication between pods that depend on each other.

Using the **`kube-proxy`** process that runs on each cluster node, Kubernetes maps the service's virtual IP address to pod IP addresses. This process allows for easier internal networking but also enables exposing of the deployment to external networks via techniques such as load balancing.

## Volumes

Volumes are objects whose purpose is to provide storage to pods. There are two basic types of volumes in Kubernetes:

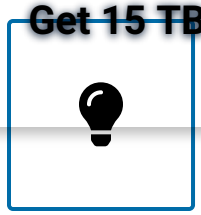
- **Ephemeral volumes** persist only during the lifetime of the pod they are tied to.
- Persistent volumes, which are not destroyed when the pod crashes. Persistent volumes are created by issuing a request called **`PersistentVolumeClaim`** (PVC). Kubernetes uses PVCs to provision volumes, which then act as links between pods and physical storage.

## Namespaces

The purpose of the Namespace object is to act as a separator of resources in the cluster. A single cluster can contain multiple namespaces, allowing administrators to organize the cluster better and simplify resource allocation.

A new cluster comes with multiple namespaces created for system purposes and the **`default`** namespace for users. Administrators can create any number of additional namespaces - for example, one for development and one for testing.

```
marko@test-main:~$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   337d
development       Active   21s
kube-node-lease   Active   337d
kube-public       Active   337d
kube-system       Active   337d
testing           Active   8s
marko@test-main:~$
```



Note: Learn how to [illegible]amespace or how to Delete Kubernetes Namespace in our detailed guides.

## Deployments

Deployments are controller objects that provide instructions on how Kubernetes should manage the pods hosting a containerized application. Using deployments, administrators can:

- Scale the number of pod replicas.
- Rollout updated code.
- Perform rollbacks to older code versions.

Once created, the deployment controller monitors the health of the pods and nodes. In case of a failure, it destroys the failed pods and creates new ones. It can also bypass the malfunctioning nodes, enabling the application to remain functional even when a hardware error occurs.

## ReplicationControllers

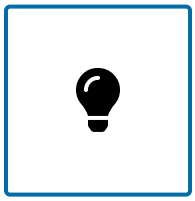
ReplicationControllers ensure that the correct number of pod replicas are running on the cluster at all times. When creating a ReplicationController, the administrator specifies the desired number of pods. The controller then maintains their number, creating additional pods and terminating the extra ones when necessary.

ReplicationControllers support equality-based selectors, which allow filtering by label keys and values. The controller will manage all the pods whose label matches the one provided in the **`.spec.selector`** field of the configuration file.

For example, the following declaration tells Kubernetes to run five **`nginx`** pods:

```
spec:
  replicas: 5
  selector:
    app: nginx
```

Since manually created pods are not automatically replaced when they fail, using replication is a recommended prac[illegible]en when the desired number of pods is one.



**Note**: Although they are not deprecated, ReplicationControllers are no longer the recommended way of setting up replication. Kubernetes documentation recommends using ReplicaSets in Deployments.

## ReplicaSets

ReplicaSets serve the same purpose as ReplicationControllers, i.e. maintaining the same number of pod replicas on the cluster. However, the difference between these two objects is the type of selectors they support. While ReplicationControllers accept only equality-based selectors, ReplicaSets additionally support set-based selectors.

Set-based selectors allow using a set of values to filter keys. The statements accept three operators: **`in`**, **`notin`**, and **`exists`**. For example, the following **`selector`** section instructs the ReplicaSet to run an Nginx pod that belongs to **`production`** and **`qa`** environments:

```
selector:
  matchLabels:
    app: nginx
  matchExpressions:

    - {key: environment, operator: In, values: [production, q
a]}
```

## DaemonSets

DaemonSets are controller objects whose purpose is to ensure that specific pods run on specific (or all) nodes in the cluster. Kubernetes scheduler ignores the pods created by a DaemonSet, so those pods last for as long as the node exists. This object is particularly useful for setting up daemons that need to run on each node, like those used for cluster storage, log collection, and node monitoring.

By default, a DaemonSet creates a pod on every node in the cluster. If the object needs to target specific nodes, their selection can be defined in the `nodeSelector` field in the configuration file.

## StatefulSets

While Deployments and Replication Controllers can handle stateless apps, stateful apps require a workload object called StatefulSet. A StatefulSet gives each pod a unique identity, which persists across pod restarts.

The applications that benefit from StatefulSet are those which require:

- **Unique persistent storage for each pod** - PersistentVolumeClaims that are used with Deployments provide shared storage for pod replicas. StatefulSets instead use VolumeClaimTemplates, which assign a unique PVC to each replica.
- **Unique network ID** - A headless service controls the pod network identity.
- **Ordered deployment, scaling, and rolling updates.**

Databases such as MySQL and PostgreSQL are examples of applications that are deployed using StatefulSets.

## ConfigMaps

ConfigMaps are Kubernetes objects used to store container configuration data in key-value pairs. By separating configuration data from the rest of the container image, ConfigMaps enable the creation of lighter and more portable images. They also allow developers to use the same code with different configurations depending on whether the app is in the development, testing, or production phase.

A pod can be configured to use the ConfigMap data by mounting the ConfigMap as a volume inside the pod:

```
volumes:
  - name: config
  configMap
    name: config-example
    items:
    - key: key1
    path: config.example
```

Alternatively, the environment variables can be used pull the specific values from the configuration:

```
env:
- name: SPECIAL_LEVEL_KEY
  valueFrom:
    configMapKeyRef:
      name: config-example
      key: key1
```

## Jobs

Jobs are workload controller objects that execute finite tasks. While other controller objects have the task of permanently maintaining the desired state and number of pods, jobs are designed to finish a task and terminate the associated pods. This property makes them useful for maintenance, monitoring, batch tasks, and work queue management.

Job instances run simultaneously or consecutively. Scheduled jobs are a separate controller object called CronJob.

**Note**: Our Bare Metal Cloud solution offers Rancher integration for easy Kubernetes management. Get started by finding the servers that suit your needs.

# Object Spec and Status

Object **spec** and object **status** are two nested fields in the object configuration that Kubernetes utilizes to control the o

- The **spec** field is used to declare the desired state of the object, i.e., the characteristics of the resource. The user provides this field when creating the object.
- The **status** field provides information about the current object state. This field is provided by Kubernetes during the lifetime of the object.

Kubernetes control plane monitors the status of every object in the cluster and attempts to match the current state to the desired state.

For example, consider the following deployment YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-test
  labels:
    app: nginx
spec:
  replicas: 5
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.23.1
        ports:
        - containerPort: 80
```

The **spec** field states that the desired number of replicas for the nginx deployment is five. After the deployment is successfully created, Kubernetes monitors its status and updates

the status field accordingly. If one replica fails, the status field reports only four running replicas, which triggers Kubernetes

# Object Required Fields

When the user wants to create a Kubernetes object, the following fields must be provided in the YAML file:

- **`apiVersion`** - Specifies the version of Kubernetes API for creating the object.
- **`kind`** - Provides the object type, for example, Deployment, ReplicaSet, or Service.
- **`metadata`** - Lists object identifiers, such as its name, UID, labels, and namespace.
- **`spec`** - States the desired state for the object, like the number of replicas and the container image.

# Object Names and IDs

Each Kubernetes object has a name and a UID. They are used to identify the object across the cluster.

Object names are user-defined and unique for object types. This means that, for example, there can be only one pod named **`test-app`** within a single namespace, but the deployment of the same name is allowed to exist.

The system assigns object UIDs, which are unique for each object instance across types and namespaces. This means that even the historical versions of the same object have different UIDs.

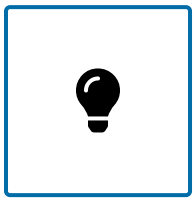
# Working with Kubernetes Objects

Kubernetes objects are managed using various GUI dashboards or using the kubectl CLI tool. With kubectl, users can manage objects by employing three distinct management techniques:

- **Imperative commands** are single action words that can change the cluster in one step. However, tracking changes in the cluster is significantly more difficult if users apply many of these commands.
- **Imperative configuration** specifies a command (such as **`create`**), options, and a YAML or JSON file containing object configuration. This approach allows storing the configuration in source control systems (such as Git) for easier access but features a

steeper learning curve and is more time-consuming.

- **Declarative configuration** transf[illegible]g the necessary operations to kubectl while requiring the user [illegible]guration files. When the user applies a configuration file, kubectl performs the necessary actions on the object.



**Note**: Always use a single technique to manage a single Kubernetes object. Mixing techniques can have undesired consequences.

## Create Objects

You can create Kubernetes using any of the management techniques listed above.

For example, the following imperative command creates an Nginx deployment:

```
kubectl create deployment nginx ––image nginx
```

The same deployment can be created using the imperative object configuration:

```
kubectl create –f nginx.yaml
```

The third option involves using the declarative approach:

```
kubectl apply –f nginx.yaml
```

## Edit Objects

Kubectl features multiple subcommands for editing an existing object. Below are the most common commands:

- **`scale`** - Update the replica count of the controller to perform horizontal scaling.
- **`annotate`** - Edit the annotation of the object.
- **`label`** - Edit the object label.

- **set fields** - Update the object's aspect.

The commands above allow users s without having to know the specific fields that need to change. The following set of commands requires a better understanding of the object schema:

- **edit** - Open and edit the object's configuration in a text editor.
- **patch** - Use a patch string to modify specific object fields.

## Delete Objects

Use the **kubectl delete** command to delete objects from the cluster. To employ the imperative approach with this command, pass the object as a command argument. For example:

```
kubectl delete deployment/nginx
```

You can also delete an object by using imperative object configuration and providing the configuration file:

```
kubectl delete –f nginx.yaml
```

and make technology less daunting to everyone.

## Conclusion

This guide introduced you to the concept of a Kubernetes object. It presented the most commonly used objects and provided insight into their form and function. Finally, some b ues were listed and explained.

F netes Architecture with Diagrams if you would like to delve d s works.

Marko Aleksic

## Next you should read

DevOps and Development, Virtualization

**Understanding Kubernetes Architecture with Diagrams**

**November 12, 2019**

Yes No

n
d
er
st
a
n
di

DevOps and Development

## Kubernetes Use Cases

**August 11, 2022**

This article focuses on Kubernetes as the most popular orchestration tool on the market. We will analyze a broad range of Kubernetes use cases from the platform's...

READ MORE

DevOps and Development, Networking

## Kubernetes Networking Guide

**June 7, 2022**

The guide shows the essentials of Kubernetes networking

n
d
h
o

DevOps and Development, Networking

## Kubernetes Service Discovery Guide

**June 2, 2022**

A service helps manage internal and external traffic to pods through IP addresses, ports, and DNS records. Service discovery...

READ MORE

ar
io
u
s



u
b
er
n
et
e
s.
..
R