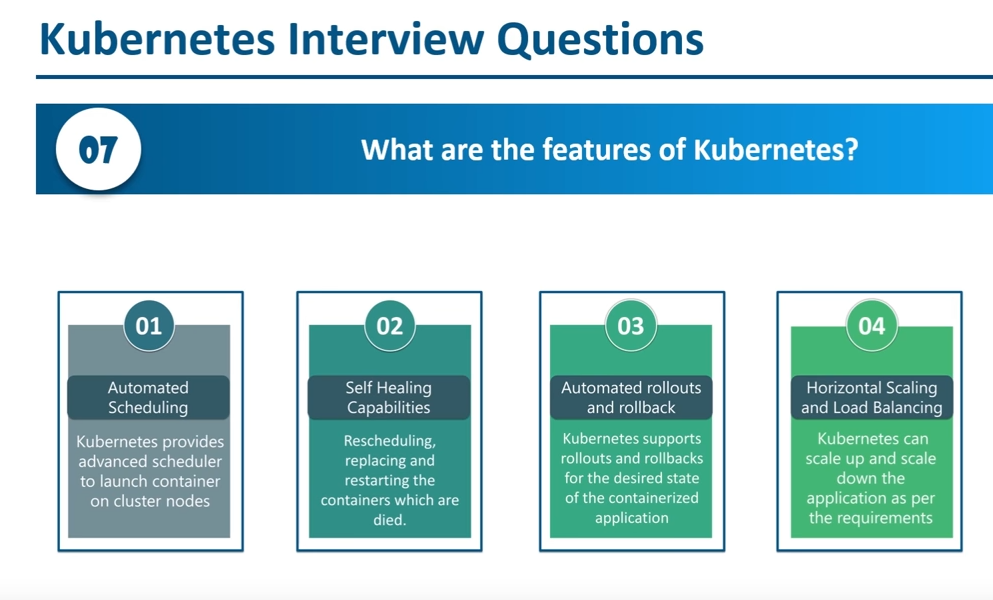
**KUBERNETES**



**Pods**

Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.

 **Pods that run a single container**. The "one-container-per-Pod" model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container; Kubernetes manages Pods rather than managing the containers directly.

 **Pods that run multiple containers that need to work together**. A Pod can encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers form a single cohesive unit of service—for example, one container serving data stored in a shared volume to the public, while a separate sidecar container refreshes or updates those files. The Pod wraps these containers, storage resources, and an ephemeral network identity together as a single unit.

**Note:** Grouping multiple co-located and co-managed containers in a single Pod is a relatively advanced use case. You should use this pattern only in specific instances in which your containers are tightly coupled.

---

apiVersion: v1

kind: Pod

metadata:

 name: rss-site

 labels:

   app: web

spec:

**containers:**

**- name: front-end**

**image: nginx**

**ports:**

**- containerPort: 80**

**- name: rss-reader**

**image:** nickchase/rss-php-nginx:v1

**ports:**

**- containerPort: 88**

**Configmap**

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

Command: kubectl create configmap <map-name> <data-source>

where <map-name> is the name you want to assign to the ConfigMap and <data-source> is the directory, file, or literal value to draw the data from.

**apiVersion**: v1

**kind**: ConfigMap

**metadata**:

**creationTimestamp**: 2016-02-18T18:52:05Z

**name**: game-config

**namespace**: default

**resourceVersion**: "516"

**uid**: b4952dc3-d670-11e5-8cd0-68f728db1985

**data**:

**game.properties**: |

*enemies=aliens*

*lives=3*

*enemies.cheat=true*

*enemies.cheat.level=noGoodRotten*

*secret.code.passphrase=UUDDLRLRBABAS*

*secret.code.allowed=true*

*secret.code.lives=30*

**ui.properties**: |

*color.good=purple*

*color.bad=yellow*

*allow.textmode=true*

*how.nice.to.look=fairlyNice*

**Secret**

Kubernetes Secrets let you store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys. Storing confidential information in a Secret is safer and more flexible than putting it verbatim in a [Pod](https://kubernetes.io/docs/concepts/workloads/pods/) definition or in a [container image](https://kubernetes.io/docs/reference/glossary/?all=true#term-image).

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in an image. Users can create Secrets and the system also creates some Secrets.

**Caution:**

Kubernetes Secrets are, by default, stored as unencrypted base64-encoded strings. By default they can be retrieved - as plain text - by anyone with API access, or anyone with access to Kubernetes' underlying data store, etcd. In order to safely use Secrets, it is recommended you (at a minimum):

1. [Enable Encryption at Rest](https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/) for Secrets.
2. [Enable or configure RBAC rules](https://kubernetes.io/docs/reference/access-authn-authz/authorization/) that restrict reading and writing the Secret. Be aware that secrets can be obtained implicitly by anyone with the permission to create a Pod.
3. **apiVersion**: v1
4. **kind**: Secret
5. **metadata**:
6. **name**: secret-sa-sample
7. **annotations**:
8. **kubernetes.io/service-account.name**: "sa-name"
9. **type**: kubernetes.io/service-account-token
10. **data**:
11. *# You can include additional key value pairs as you do with Opaque Secrets*
12. **extra**: YmFyCg==

**DaemonSet**

A *DaemonSet* ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

Some typical uses of a DaemonSet are:

* running a cluster storage daemon on every node
* running a logs collection daemon on every node
* running a node monitoring daemon on every node

In a simple case, one DaemonSet, covering all nodes, would be used for each type of daemon. A more complex setup might use multiple DaemonSets for a single type of daemon, but with different flags and/or different memory and cpu requests for different hardware types

**apiVersion**: apps/v1

**kind**: DaemonSet

**metadata**:

**name**: fluentd-elasticsearch

**namespace**: kube-system

**labels**:

**k8s-app**: fluentd-logging

**spec**:

**selector**:

**matchLabels**:

**name**: fluentd-elasticsearch

**template**:

**metadata**:

**labels**:

**name**: fluentd-elasticsearch

**spec**:

**tolerations**:

*# this toleration is to have the daemonset runnable on master nodes*

*# remove it if your masters can't run pods*

- **key**: node-role.kubernetes.io/master

**effect**: NoSchedule

**containers**:

- **name**: fluentd-elasticsearch

**image**: quay.io/fluentd\_elasticsearch/fluentd:v2.5.2

**resources**:

**limits**:

**memory**: 200Mi

**requests**:

**cpu**: 100m

**memory**: 200Mi

**volumeMounts**:

- **name**: varlog

**mountPath**: /var/log

- **name**: varlibdockercontainers

**mountPath**: /var/lib/docker/containers

**readOnly**: **true**

**terminationGracePeriodSeconds**: 30

**volumes**:

- **name**: varlog

**hostPath**:

**path**: /var/log

- **name**: varlibdockercontainers

**hostPath**:

**path**: /var/lib/docker/containers

**Jobs**

A Job creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete. Deleting a Job will clean up the Pods it created.

A simple case is to create one Job object in order to reliably run one Pod to completion. The Job object will start a new Pod if the first Pod fails or is deleted (for example due to a node hardware failure or a node reboot).

**apiVersion**: batch/v1

**kind**: Job

**metadata**:

**name**: pi

**spec**:

**template**:

**spec**:

**containers**:

- **name**: pi

**image**: perl

**command**: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]

**restartPolicy**: Never

**backoffLimit**: 4

**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.

**apiVersion**: apps/v1

**kind**: ReplicaSet

**metadata**:

**name**: frontend

**labels**:

**app**: guestbook

**tier**: frontend

**spec**:

*# modify replicas according to your case*

**replicas**: 3

**selector**:

**matchLabels**:

**tier**: frontend

**template**:

**metadata**:

**labels**:

**tier**: frontend

**spec**:

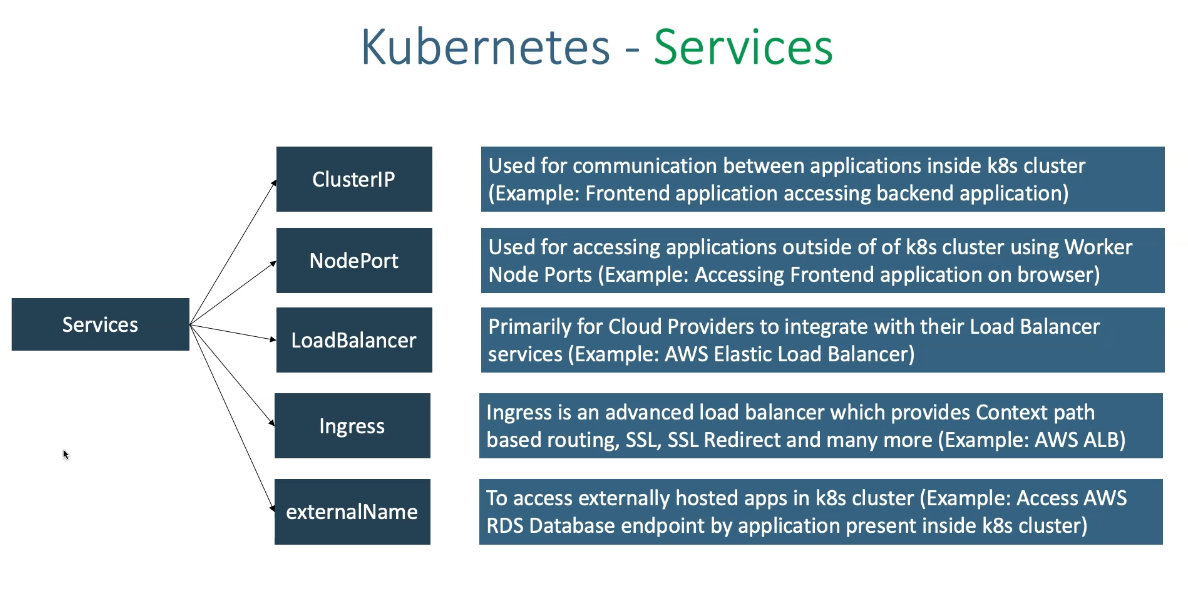
**containers**:

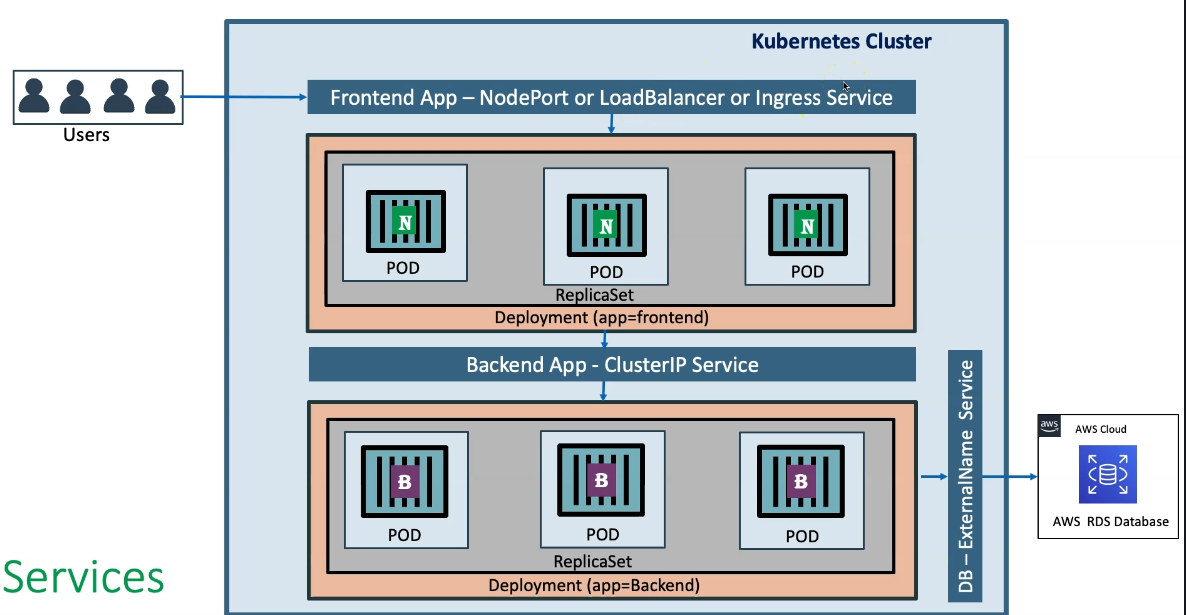
- **name**: php-redis

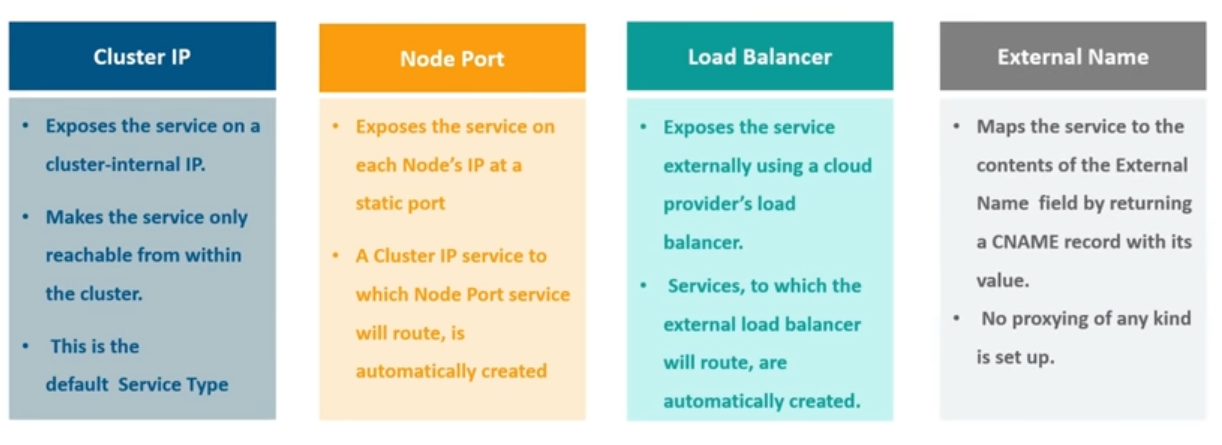
**image**: gcr.io/google\_samples/gb-frontend:v3

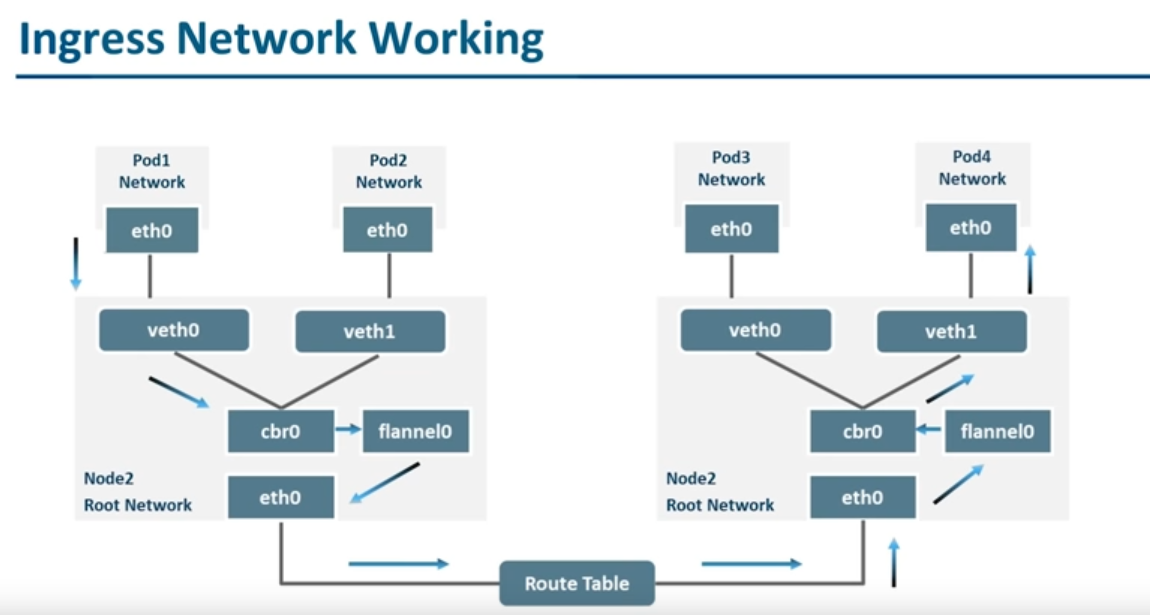
**Service**

In Kubernetes, a Service is an abstraction which defines a logical set of Pods and a policy by which to access them (sometimes this pattern is called a micro-service).









**apiVersion**: v1

**kind**: Service

**metadata**:

**name**: my-service

**spec**:

**selector**:

**app**: MyApp

**ports**:

- **protocol**: TCP

**port**: 80

**targetPort**: 9376

**serviceAccount**

A service account provides an identity for processes that run in a Pod.

**apiVersion**: v1

**kind**: ServiceAccount

**metadata**:

**creationTimestamp**: 2015-06-16T00:12:59Z

**name**: build-robot

**namespace**: default

**resourceVersion**: "272500"

**uid**: 721ab723-13bc-11e5-aec2-42010af0021e

**secrets**:

- **name**: build-robot-token-bvbk5

**namespace**

Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.

When to Use Multiple Namespaces

Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all. Start using namespaces when you need the features they provide.

**Statefulset**

StatefulSet is the workload API object used to manage stateful applications.

Manages the deployment and scaling of a set of [Pods](https://kubernetes.io/docs/concepts/workloads/pods/), *and provides guarantees about the ordering and uniqueness* of these Pods

**Volumes**

On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem is the loss of files when a container crashes. The kubelet restarts the container but with a clean state. A second problem occurs when sharing files between containers running together in a Pod. The Kubernetes [volume](https://kubernetes.io/docs/concepts/storage/volumes/) abstraction solves both of these problems. Familiarity with [Pods](https://kubernetes.io/docs/concepts/workloads/pods/) is suggested.

**A PersistentVolume (PV)** is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using [Storage Classes](https://kubernetes.io/docs/concepts/storage/storage-classes/). It is a resource in the cluster just like a node is a cluster resource

**apiVersion**: v1

**kind**: PersistentVolume

**metadata**:

**name**: pv0003

**spec**:

**capacity**:

**storage**: 5Gi

**volumeMode**: Filesystem

**accessModes**:

- ReadWriteOnce

**persistentVolumeReclaimPolicy**: Recycle

**storageClassName**: slow

**mountOptions**:

- hard

- nfsvers=4.1

**nfs**:

**path**: /tmp

**server**: 172.17.0.2

A **PersistentVolumeClaim (PVC)** is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory)

**apiVersion**: v1

**kind**: PersistentVolumeClaim

**metadata**:

**name**: myclaim

**spec**:

**accessModes**:

- ReadWriteOnce

**volumeMode**: Filesystem

**resources**:

**requests**:

**storage**: 8Gi

**storageClassName**: slow

**selector**:

**matchLabels**:

**release**: "stable"

**matchExpressions**:

- {**key: environment, operator: In, values**: [dev]}

1. **Emptydir this volume is created when the pod is launch and die when the pod die**
2. **Hostpath this create when the node is create and does not die**
3. deployment
4. readiness probe
5. livenessprobe
6. ingress
7. =======
8. =======