


# K8s 101

## k8s Components

### KUBERNETES KOMPONENTER

- ▶ Komponenter

- ▶ Pod 

- ▶ Service

- ▶ Ingress

- ▶ Volumes

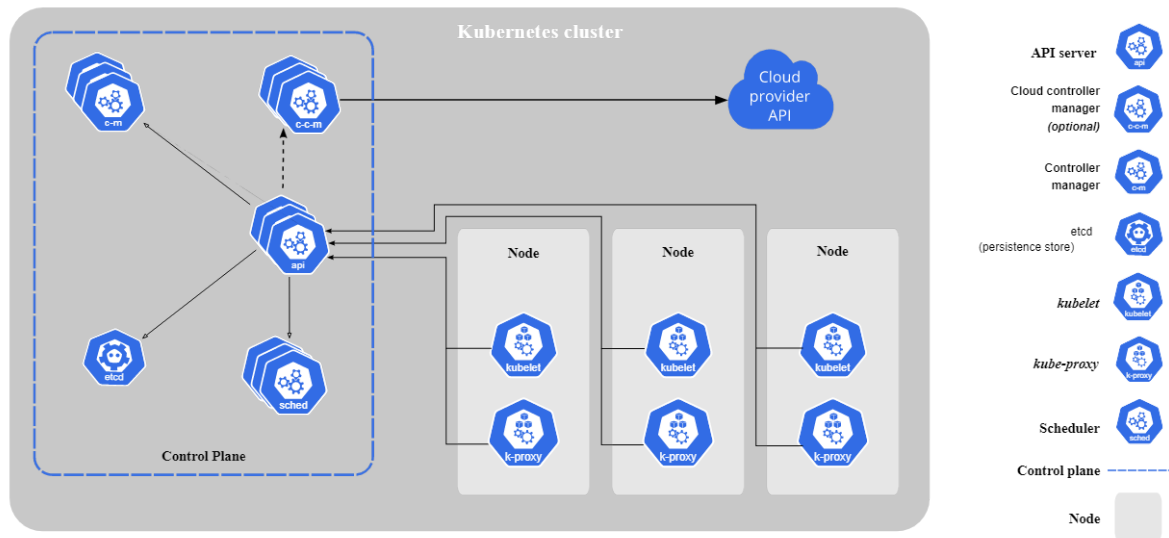
- ▶ Secrets

- ▶ ConfigMap

- ▶ StatefulSet

- ▶ Deployment

# A picture of an Cluster



When you deploy Kubernetes, you get a cluster.

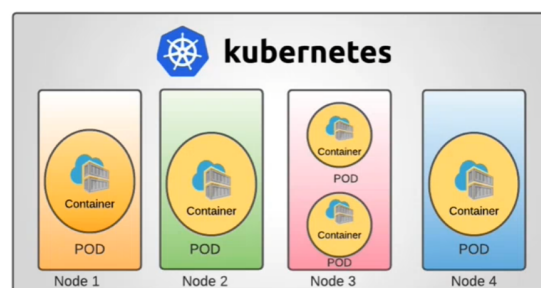
A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.

The worker node(s) host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

## Nodes and Pods:

### KUBERNETES KOMPONENTER

- ▶ Node och Pod
  - ▶ Node (Server eller VM)
  - ▶ Pod (Abstraktion över container)
  - ▶ Vanligtvis 1 container/Pod
  - ▶ Varje Pod => unik IP

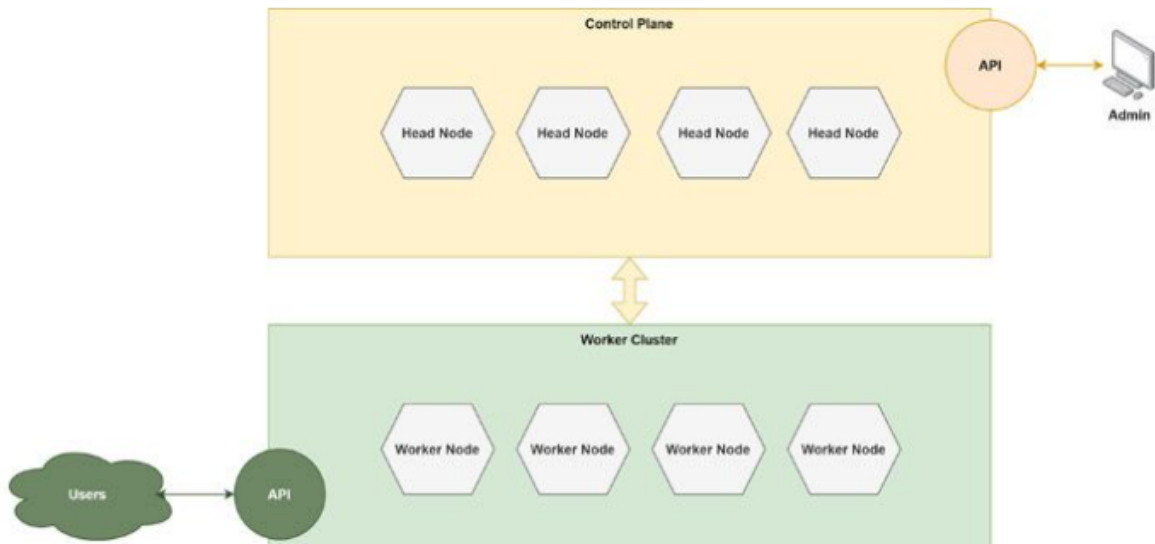


There are r.

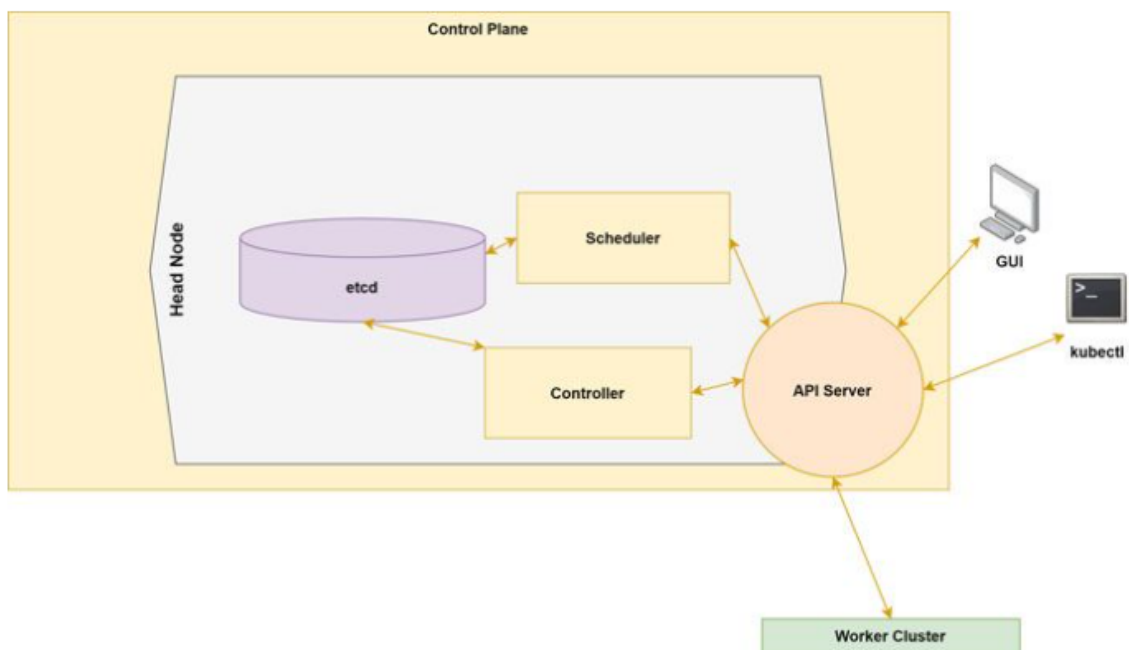
## Master Nodes (head nodes):

A Node is a **VM** or a “**computer**”.

Inside of the worker nodes here is where the containerized apps live, and in the master nodes are the tools to handle that cluster. See the figures.



## A Master Node in detail:



**etcd (et:si:di)**

Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

If your Kubernetes cluster uses etcd as its backing store, make sure you have a [back up](#) plan for those data.

You can find in-depth information about etcd in the official [documentation](#).

**Scheduler**

Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on.

Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

**Api server**

The core of Kubernetes' control plane is the API server. The API server exposes an HTTP API that lets end users, different parts of your cluster, and external components communicate with one another.

The Kubernetes API lets you query and manipulate the state of API objects in Kubernetes (for example: Pods, Namespaces, ConfigMaps, and Events).

The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.

Apiserver is designed to scale horizontally—that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

**Controller-Manager**

Control plane component that runs controller processes.

The controller is the component that ensures that the cluster remains in the configured state and returns it to that state if it begins to drift. The controller acts as a kind of thermostat that sets a desired state and then strives to maintain it.

Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

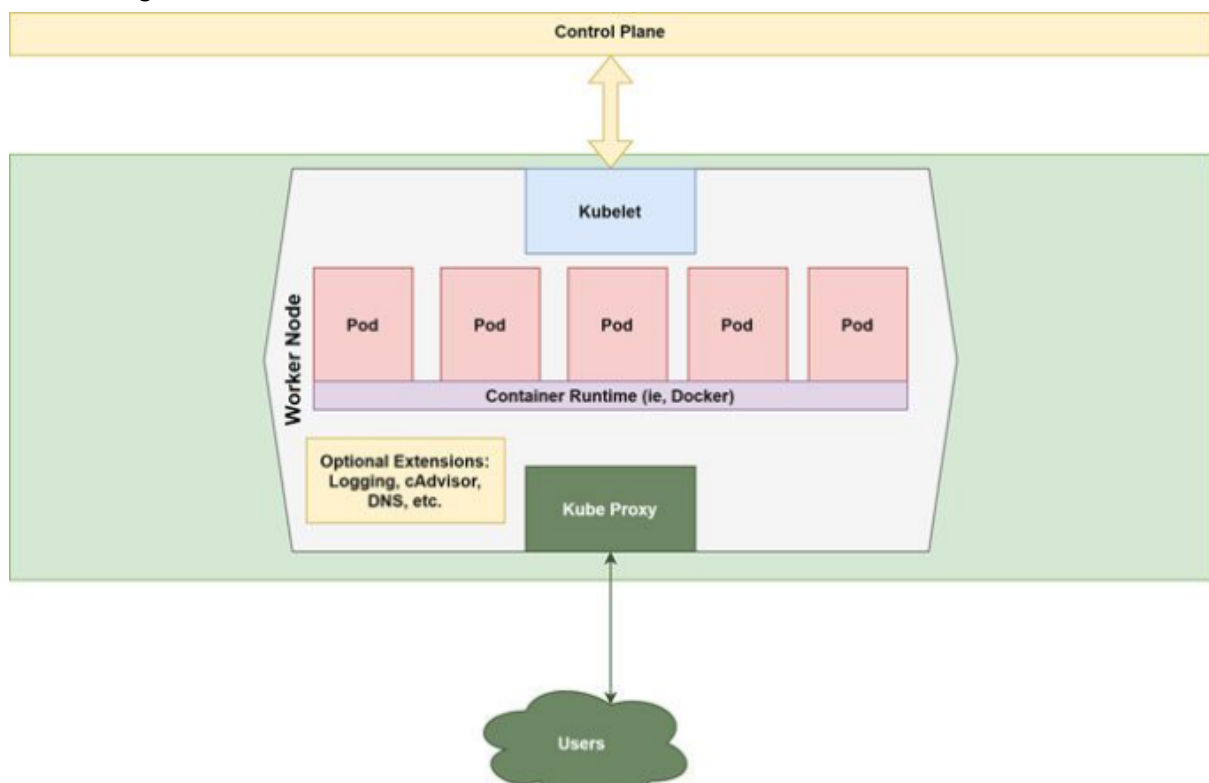
Some types of these controllers are:

- Node controller: Responsible for noticing and responding when nodes go down.
- Job controller: Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
- EndpointSlice controller: Populates EndpointSlice objects (to provide a link between Services and Pods).
- ServiceAccount controller: Create default ServiceAccounts for new namespaces.

### Worker Nodes:

The Components inside of a worker node consist of a “Kubelet”, “Kubeproxy”, “Container Runtime” (like Docker for instance) and Pods. Also There are optional Extensions.

See the figure.



### Kubelet

An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.

The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

## **Kube-proxy**

Kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept.

[kube-proxy](#) maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.

Kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

## **Container runtime**

The container runtime is the software that is responsible for running containers.

Kubernetes supports container runtimes such as Containerd, CRI-O, and any other implementation of the [Kubernetes CRI \(Container Runtime Interface\)](#).

## **Pods**

Each pod has a unique IP address, also a pod is an abstraction over a container.

Pods are discrete units of work on the node. They are at the replication level. Pods are abstractions that wrap one or more containerized applications.

Pods provide the ability to logically group and isolate containers that run together, while enabling communication between pods on the same machine. The relationship between containers and pods is controlled by Kubernetes deployment descriptors.

Read more on pods on kubernetes doc:

<https://kubernetes.io/docs/concepts/workloads/pods/>

## **Deployments and Replica Sets**

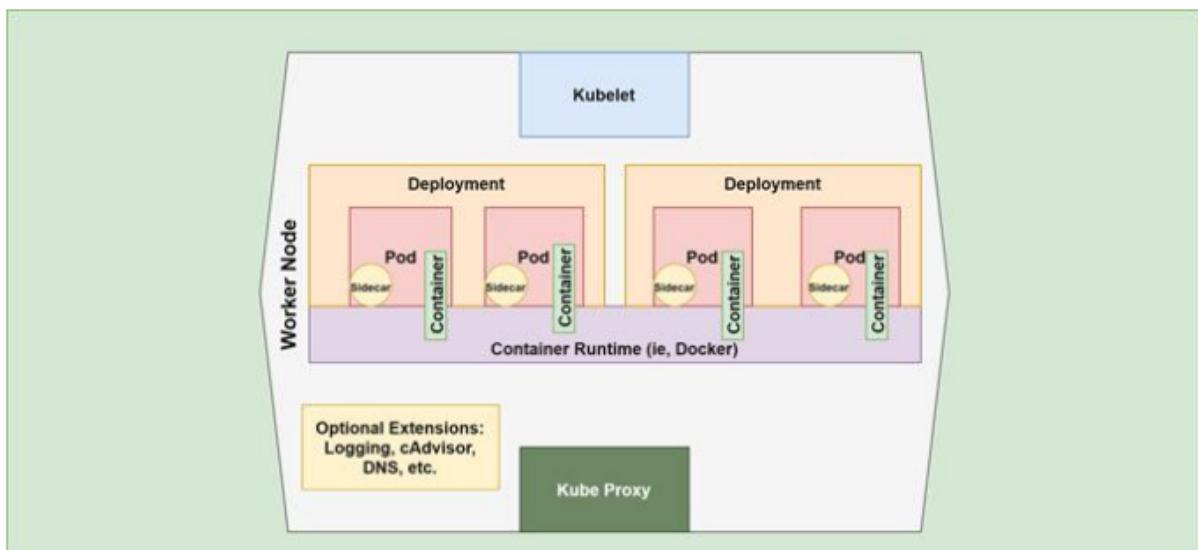
Typically, pods are configured and deployed as parts of ReplicaSets. A ReplicaSet defines the desired execution properties of the pod and makes Kubernetes work to maintain that state.

ReplicaSets are usually defined by a Deployment, which specifies both the parameters of the current ReplicaSet and the strategy to be used (that is, if pods are to be updated or newly created) when managing the cluster.

### Sidecar (Add-ons)

At the pod level, extra functions can be added through add-ons called sidecars. The sidecars handle tasks such as logging at the pod level and collecting statistics.

### A Pod in Detail:



## Service

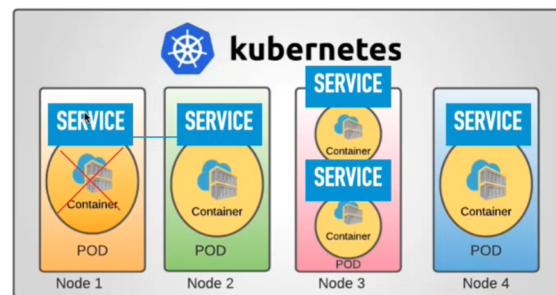
In Kubernetes, a Service is a method for exposing a network application that is running as one or more Pods in your cluster.

A key aim of Services in Kubernetes is that you don't need to modify your existing application to use an unfamiliar service discovery mechanism. You can run code in Pods, whether this is a code designed for a cloud-native world, or an older app you've containerized. You use a Service to make that set of Pods available on the network so that clients can interact with it.

If you use a Deployment to run your app, that Deployment can create and destroy Pods dynamically. From one moment to the next, you don't know how many of those Pods are working and healthy; you might not even know what those healthy Pods are named. Kubernetes Pods are created and destroyed to match the desired state of your cluster. Pods are ephemeral resources (you should not expect that an individual Pod is reliable and durable).

## KUBERNETES KOMPONENTER

- ▶ Service ger Pod fast IP
  - ▶ Om Pod dör lever Service kvar
  - ▶ Podar kan prata ostört!
  - ▶ 2 typer
    - ▶ External service (öppen/frontend)
    - ▶ Internal service (privat/backend)



Each Pod gets its own IP address (Kubernetes expects network plugins to ensure this). For a given Deployment in your cluster, the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later.

This leads to a problem: if some set of Pods (call them "backends") provides functionality to other Pods (call them "frontends") inside your cluster, how do the frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload?

<https://kubernetes.io/docs/concepts/services-networking/service/>

## Defining a Service

A Service in Kubernetes is an object (the same way that a Pod or a ConfigMap is an object).

You can create, view or modify Service definitions using the Kubernetes API. Usually you use a tool such as `kubectl` to make those API calls for you.

For example, suppose you have a set of Pods that each listen on TCP port 9376 and are labeled as `app.kubernetes.io/name=MyApp`. You can define a Service to publish that TCP listener:



```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Applying this manifest creates a new Service named "my-service", which targets TCP port 9376 on any Pod with the `app.kubernetes.io/name: MyApp` label.

Kubernetes assigns this Service an IP address (the *cluster IP*), that is used by the virtual IP address mechanism. For more details on that mechanism, read [Virtual IPs and Service Proxies](#).

## Port definitions

Port definitions in Pods have names, and you can reference these names in the `targetPort` attribute of a Service. For example, we can bind the `targetPort` of the Service to the Pod port in the following way:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app.kubernetes.io/name: proxy
spec:
  containers:
    - name: nginx
      image: nginx:stable
      ports:
        - containerPort: 80
          name: http-web-svc

---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app.kubernetes.io/name: proxy
  ports:
    - name: name-of-service-port
      protocol: TCP
      port: 80
      targetPort: http-web-svc
```

# Ingress

## What is Ingress?

An API object that manages external access to the services in a cluster, typically HTTP.

**Ingress** exposes HTTP and HTTPS routes from outside the cluster to **services** within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

Here is a simple example where an Ingress sends all its traffic to one Service:



An Ingress may be configured to give Services externally-reachable URLs, load balance traffic, terminate SSL / TLS, and offer name-based virtual hosting. An **Ingress controller** is responsible for fulfilling the Ingress, usually with a load balancer, though it may also configure your edge router or additional frontends to help handle the traffic.

An Ingress does not expose arbitrary ports or protocols. Exposing services other than HTTP and HTTPS to the internet typically uses a service of type **Service.Type=NodePort** or **Service.Type=LoadBalancer**.

Read More on Ingress Here:

<https://kubernetes.io/docs/concepts/services-networking/ingress/>

# Volumes

In Kubernetes, a volume can be thought of as a directory which is accessible to the containers in a pod. We have different types of volumes in Kubernetes and the type defines how the volume is created and its content.

The concept of volume was present with the Docker, however the only issue was that the volume was very much limited to a particular pod. As soon as the life of a pod ended, the volume was also lost.

On the other hand, the volumes that are created through Kubernetes is not limited to any container. It supports any or all the containers deployed inside the pod of Kubernetes. A key advantage of Kubernetes volume is, it supports different kind of storage wherein the pod can use multiple of them at the same time.

[https://www.tutorialspoint.com/kubernetes/kubernetes\\_volumes.htm](https://www.tutorialspoint.com/kubernetes/kubernetes_volumes.htm)

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 abstraction solves both of these problems. Familiarity with Pods is suggested.

<https://kubernetes.io/docs/concepts/storage/volumes/>

# Configmap

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 you to decouple environment-specific configuration from your container images, so that your applications are easily portable.

Use a ConfigMap for setting configuration data separately from application code.

For example, imagine that you are developing an application that you can run on your own computer (for development) and in the cloud (to handle real traffic). You write the code to look in an environment variable named `DATABASE_HOST`. Locally, you set that variable to `localhost`. In the cloud, you set it to refer to a Kubernetes Service that exposes the database component to your cluster. This lets you fetch a container image running in the cloud and debug the exact same code locally if needed.

A ConfigMap is not designed to hold large chunks of data. The data stored in a ConfigMap cannot exceed 1 MiB. If you need to store settings that are larger than this limit, you may want to consider mounting a volume or use a separate database or file service.

## 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. These fields accept key-value pairs as their values. Both the `data` field and the `binaryData` are optional. The `data` field is designed to contain UTF-8 strings while the `binaryData` field is designed to contain binary data as base64-encoded strings.

The name of a ConfigMap must be a valid [DNS subdomain name](#).

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

Starting from v1.19, you can add an `immutable` field to a ConfigMap definition to create an [immutable ConfigMap](#).

## ConfigMaps and Pods

You can write a Pod `spec` that refers to a ConfigMap and configures the container(s) in that Pod based on the data in the ConfigMap. The Pod and the ConfigMap must be in the same namespace.

**Note:** The `spec` of a static Pod cannot refer to a ConfigMap or any other API objects.

Here's an example ConfigMap that has some keys with single values, and other keys where the value looks like a fragment of a configuration format.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  # property-like keys; each key maps to a simple value
  player_initial_lives: "3"
  ui_properties_file_name: "user-interface.properties"

  # file-like keys
  game.properties: |
    enemy.types=aliens,monsters
    player.maximum-lives=5
  user-interface.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
```

There are four different ways that you can use a ConfigMap to configure a container inside a Pod:

1. Inside a container command and args
2. Environment variables for a container
3. Add a file in read-only volume, for the application to read
4. Write code to run inside the Pod that uses the Kubernetes API to read a ConfigMap

These different methods lend themselves to different ways of modeling the data being consumed. For the first three methods, the kubelet uses the data from the ConfigMap when it launches container(s) for a Pod.

The fourth method means you have to write code to read the ConfigMap and its data. However, because you're using the Kubernetes API directly, your application can subscribe to get updates whenever the ConfigMap changes, and react when that happens. By accessing the Kubernetes API directly, this technique also lets you access a ConfigMap in a different namespace.

Read more on Configmap here:

<https://kubernetes.io/docs/concepts/configuration/configmap/>