What is Kubernetes?

[Kubernetes](https://www.aquasec.com/cloud-native-academy/kubernetes-101/kubernetes-complete-guide/) is an open source orchestration tool developed by Google for managing microservices or containerized applications across a distributed cluster of nodes.

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available

Why you need Kubernetes and what it can do

Containers are a good way to bundle and run your applications.

In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime.

For example, if a container goes down, another container needs to start. Wouldn't it be easier if this behavior was handled by a system?

What kubernetes provide =>

* **Service discovery and load balancing** Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
* **Storage orchestration** Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
* **Automated rollouts and rollbacks** You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate.

For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.

* **Automatic bin packing** You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
* **Self-healing** Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve
* **Secret and configuration management** Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.

**Kubernetes Components and Architecture=>**



Master Components=>

* [**etcd**](https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/)

a simple, distributed key value storage which is used to store the Kubernetes cluster data (such as number of pods, their state, namespace, etc), API objects and service discovery details.

It is only accessible from the API server for security reasons.

* [**kube-apiserver**](https://kubernetes.io/docs/reference/generated/kube-apiserver/) – Kubernetes API server is the central management entity that receives all REST requests for modifications (to pods, services, replication sets/controllers and others).

Also, this is the only component that communicates with the etcd cluster, making sure data is stored in etcd and is in agreement with the service details of the deployed pods.

* [**kube HYPERLINK "https://kubernetes.io/docs/reference/generated/kube-controller-manager/" HYPERLINK "https://kubernetes.io/docs/reference/generated/kube-controller-manager/" HYPERLINK "https://kubernetes.io/docs/reference/generated/kube-controller-manager/" HYPERLINK "https://kubernetes.io/docs/reference/generated/kube-controller-manager/" HYPERLINK "https://kubernetes.io/docs/reference/generated/kube-controller-manager/" HYPERLINK "https://kubernetes.io/docs/reference/generated/kube-controller-manager/" HYPERLINK "https://kubernetes.io/docs/reference/generated/kube-controller-manager/" controller-manager**](https://kubernetes.io/docs/reference/generated/kube-controller-manager/)

runs a number of distinct controller processes in the background.

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.
* Endpoints controller: Populates the Endpoints object (that is, joins Services & Pods).
* Service Account & Token controllers: Create default accounts and API access tokens for new namespaces
* **kube-scheduler=>**

Control plane component that watches for newly created [Pods](https://kubernetes.io/docs/concepts/workloads/pods/) with no assigned [node](https://kubernetes.io/docs/concepts/architecture/nodes/), and selects a node for them to run on.

 on the various nodes based on resource utilization. It reads the service’s operational requirements and schedules it on the best fit node. For example, if the application needs 1GB of memory and 2 CPU cores, then the pods for that application will be scheduled on a node with at least those resources. The scheduler runs each time there is a need to schedule pods. The scheduler must know the total resources available as well as resources allocated to existing workloads on each node.

**Node (worker) components=>**

Kubelet=>

An agent that runs on each [node](https://kubernetes.io/docs/concepts/architecture/nodes/) in the cluster. It makes sure that [containers](https://kubernetes.io/docs/concepts/containers/) are running in a [Pod](https://kubernetes.io/docs/concepts/workloads/pods/).

the main service on a node, regularly taking in new or modified pod specifications (primarily through the kube-apiserver) and ensuring that pods and their containers are healthy and running in the desired state. This component also reports to the master on the health of the host where it is running.

* [**kube HYPERLINK "https://kubernetes.io/docs/reference/generated/kube-proxy/" HYPERLINK "https://kubernetes.io/docs/reference/generated/kube-proxy/" HYPERLINK "https://kubernetes.io/docs/reference/generated/kube-proxy/" HYPERLINK "https://kubernetes.io/docs/reference/generated/kube-proxy/" HYPERLINK "https://kubernetes.io/docs/reference/generated/kube-proxy/" HYPERLINK "https://kubernetes.io/docs/reference/generated/kube-proxy/" HYPERLINK "https://kubernetes.io/docs/reference/generated/kube-proxy/"-proxy**](https://kubernetes.io/docs/reference/generated/kube-proxy/) =>

a proxy service that runs on each worker node to deal with individual host subnetting and expose services to the external world. It performs request forwarding to the correct pods/containers across the various isolated networks in a cluster.

Kubernetes Concepts

Making use of Kubernetes requires understanding the different abstractions it uses to represent the state of the system, such as services, pods, volumes, namespaces, and deployments.

* [**Pod**](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) – generally refers to one or more containers that should be controlled as a single application. A pod encapsulates application containers, storage resources, a unique network ID and other configuration on how to run the containers.
* [**Service**](https://kubernetes.io/docs/concepts/services-networking/service/) – pods are volatile, that is Kubernetes does not guarantee a given physical pod will be kept alive (for instance, the replication controller might kill and start a new set of pods). Instead, a service represents a logical set of pods and acts as a gateway, allowing (client) pods to send requests to the service without needing to keep track of which physical pods actually make up the service.
* [**Volume**](https://kubernetes.io/docs/concepts/storage/volumes/) – similar to a container volume in Docker, but a Kubernetes volume applies to a whole pod and is mounted on all containers in the pod. Kubernetes guarantees data is preserved across container restarts. The volume will be removed only when the pod gets destroyed. Also, a pod can have multiple volumes (possibly of different types) associated.
* [**Namespace**](https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/) – a virtual cluster (a single physical cluster can run multiple virtual ones) intended for environments with many users spread across multiple teams or projects, for isolation of concerns. Resources inside a namespace must be unique and cannot access resources in a different namespace. Also, a namespace can be allocated a [resource quota](https://kubernetes.io/docs/concepts/policy/resource-quotas/) to avoid consuming more than its share of the physical cluster’s overall resources.
* [**Deployment**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) – describes the desired state of a pod or a replica set, in a yaml file. The deployment controller then gradually updates the environment (for example, creating or deleting replicas) until the current state matches the desired state specified in the deployment file. For example, if the yaml file defines 2 replicas for a pod but only one is currently running, an extra one will get created. Note that replicas managed via a deployment should not be manipulated directly, only via new deployments.

Kubernetes Design Principles=>

Kubernetes was designed to support the features required by highly available distributed systems, such as (auto-)scaling, high availability, security and portability.

* **Scalability** – Kubernetes provides horizontal scaling of pods on the basis of CPU utilization. The threshold for CPU usage is configurable and Kubernetes will automatically start new pods if the threshold is reached. For example, if the threshold is 70% for CPU but the application is actually growing up to 220%, then eventually 3 more pods will be deployed so that the average CPU utilization is back under 70%. When there are multiple pods for a particular application, Kubernetes provides the load balancing capacity across them. Kubernetes also supports horizontal scaling of stateful pods, including NoSQL and RDBMS databases through Stateful sets. A Stateful set is a similar concept to a Deployment, but ensures storage is persistent and stable, even when a pod is removed.
* **High Availability** – Kubernetes addresses highly availability both at application and infrastructure level. Replica sets ensure that the desired (minimum) number of replicas of a stateless pod for a given application are running. Stateful sets perform the same role for stateful pods. At the infrastructure level, Kubernetes supports various distributed storage backends like AWS EBS, Azure Disk, Google Persistent Disk, NFS, and more. Adding a reliable, available storage layer to Kubernetes ensures high availability of stateful workloads. Also, each of the master components can be configured for multi-node replication (multi-master) to ensure higher availability.
* **Security** – [Kubernetes](https://wiki.aquasec.com/display/containers/Kubernetes+Security) addresses security at multiple levels: cluster, application and network. The API endpoints are secured through transport layer security (TLS). Only authenticated users (either service accounts or regular users) can execute operations on the cluster (via API requests). At the application level, Kubernetes secrets can store sensitive information (such as passwords or tokens) per cluster (a virtual cluster if using namespaces, physical otherwise). Note that secrets are accessible from any pod in the same cluster. Network policies for access to pods can be defined in a deployment. A network policy specifies how pods are allowed to communicate with each other and with other network endpoints.
* **Portability** – Kubernetes portability manifests in terms of operating system choices (a cluster can run on any mainstream Linux distribution), processor architectures (either virtual machines or bare metal), cloud providers (AWS, Azure or Google Cloud Platform), and new container runtimes, besides Docker, can also be added. Through the concept of federation, it can also support workloads across hybrid (private and public cloud) or multi-cloud environments. This also supports availability zone fault tolerance within a single cloud provider.

controler manager types:=>

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.

Deployments

A Deployment provides declarative updates for Pods and ReplicaSets.

You describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate.

StatefulSets

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

Manages the deployment and scaling of a set of Pods, and provides guarantees about the ordering and uniqueness of these Pods.

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.

Pods State=>

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. This includes time a Pod spends waiting to be scheduled as well as the time spent downloading container images over the network.

Running-

The Pod has been bound to a node, and all of the containers have been created. At least one container is still running, or is in the process of starting or restarting.

Succeeded-

All containers in the Pod have terminated in success, and will not be restarted.

Failed-

All containers in the Pod have terminated, and at least one container has terminated in failure. That is, the container either exited with non-zero status or was terminated by the system.

Unknown-

For some reason the state of the Pod could not be obtained. This phase typically occurs due to an error in communicating with the node where the Pod should be running.

Service Types=>

Kubernetes NodePort vs LoadBalancer vs Ingress? When should I use what?

Recently, someone asked me what the difference between NodePorts, LoadBalancers, and Ingress were. They are all different ways to get external traffic into your cluster, and they all do it in different ways. Let’s take a look at how each of them work, and when you would use each.

Note: Everything here applies to Google Kubernetes Engine. If you are running on another cloud, on prem, with minikube, or something else, these will be slightly different. I’m also not going into deep technical details. If you are interested in learning more, the official documentation is a great resource!

ClusterIP

A ClusterIP service is the default Kubernetes service. It gives you a service inside your cluster that other apps inside your cluster can access. There is no external access.

The YAML for a ClusterIP service looks like this:

apiVersion: v1

kind: Service

metadata:

name: my-internal-service

spec:

selector:

app: my-app

type: ClusterIP

ports:

- name: http

port: 80

targetPort: 80

protocol: TCP

If you can’t access a ClusterIP service from the internet, why am I talking about it? Turns out you can access it using the Kubernetes proxy!

Thanks to

Ahmet Alp Balkan

for the diagrams

Start the Kubernetes Proxy:

$ kubectl proxy --port=8080

Now, you can navigate through the Kubernetes API to access this service using this scheme:

[http://localhost:8080/api/v1/proxy/namespaces/ HYPERLINK "http://localhost:8080/api/v1/proxy/namespaces/%3CNAMESPACE%3E/services/%3CSERVICE-NAME%3E:%3CPORT-NAME%3E/"< HYPERLINK "http://localhost:8080/api/v1/proxy/namespaces/%3CNAMESPACE%3E/services/%3CSERVICE-NAME%3E:%3CPORT-NAME%3E/"NAMESPACE HYPERLINK "http://localhost:8080/api/v1/proxy/namespaces/%3CNAMESPACE%3E/services/%3CSERVICE-NAME%3E:%3CPORT-NAME%3E/"> HYPERLINK "http://localhost:8080/api/v1/proxy/namespaces/%3CNAMESPACE%3E/services/%3CSERVICE-NAME%3E:%3CPORT-NAME%3E/"/services/ HYPERLINK "http://localhost:8080/api/v1/proxy/namespaces/%3CNAMESPACE%3E/services/%3CSERVICE-NAME%3E:%3CPORT-NAME%3E/"< HYPERLINK "http://localhost:8080/api/v1/proxy/namespaces/%3CNAMESPACE%3E/services/%3CSERVICE-NAME%3E:%3CPORT-NAME%3E/"SERVICE-NAME HYPERLINK "http://localhost:8080/api/v1/proxy/namespaces/%3CNAMESPACE%3E/services/%3CSERVICE-NAME%3E:%3CPORT-NAME%3E/"> HYPERLINK "http://localhost:8080/api/v1/proxy/namespaces/%3CNAMESPACE%3E/services/%3CSERVICE-NAME%3E:%3CPORT-NAME%3E/": HYPERLINK "http://localhost:8080/api/v1/proxy/namespaces/%3CNAMESPACE%3E/services/%3CSERVICE-NAME%3E:%3CPORT-NAME%3E/"< HYPERLINK "http://localhost:8080/api/v1/proxy/namespaces/%3CNAMESPACE%3E/services/%3CSERVICE-NAME%3E:%3CPORT-NAME%3E/"PORT-NAME HYPERLINK "http://localhost:8080/api/v1/proxy/namespaces/%3CNAMESPACE%3E/services/%3CSERVICE-NAME%3E:%3CPORT-NAME%3E/"> HYPERLINK "http://localhost:8080/api/v1/proxy/namespaces/%3CNAMESPACE%3E/services/%3CSERVICE-NAME%3E:%3CPORT-NAME%3E/"/](http://localhost:8080/api/v1/proxy/namespaces/%3CNAMESPACE%3E/services/%3CSERVICE-NAME%3E:%3CPORT-NAME%3E/)

So to access the service we defined above, you could use the following address:

<http://localhost:8080/api/v1/proxy/namespaces/default/services/my-internal-service:http/>

When would you use this?

There are a few scenarios where you would use the Kubernetes proxy to access your services.

Debugging your services, or connecting to them directly from your laptop for some reason

Allowing internal traffic, displaying internal dashboards, etc.

Because this method requires you to run kubectl as an authenticated user, you should NOT use this to expose your service to the internet or use it for production services.

NodePort

A NodePort service is the most primitive way to get external traffic directly to your service. NodePort, as the name implies, opens a specific port on all the Nodes (the VMs), and any traffic that is sent to this port is forwarded to the service.

This isn’t the most technically accurate diagram, but I think it illustrates the point of how a NodePort works

The YAML for a NodePort service looks like this:

apiVersion: v1

kind: Service

metadata:

name: my-nodeport-service

spec:

selector:

app: my-app

type: NodePort

ports:

- name: http

port: 80

targetPort: 80

nodePort: 30036

protocol: TCP

Basically, a NodePort service has two differences from a normal “ClusterIP” service. First, the type is “NodePort.” There is also an additional port called the nodePort that specifies which port to open on the nodes. If you don’t specify this port, it will pick a random port. Most of the time you should let Kubernetes choose the port; as

thockin

says, there are many caveats to what ports are available for you to use.

When would you use this?

There are many downsides to this method:

You can only have one service per port

You can only use ports 30000–32767

If your Node/VM IP address change, you need to deal with that

For these reasons, I don’t recommend using this method in production to directly expose your service. If you are running a service that doesn’t have to be always available, or you are very cost sensitive, this method will work for you. A good example of such an application is a demo app or something temporary.

LoadBalancer:

A LoadBalancer service is the standard way to expose a service to the internet. On GKE, this will spin up a Network Load Balancer that will give you a single IP address that will forward all traffic to your service.

Thanks to

Ahmet Alp Balkan

for the diagrams

When would you use this?

If you want to directly expose a service, this is the default method. All traffic on the port you specify will be forwarded to the service. There is no filtering, no routing, etc. This means you can send almost any kind of traffic to it, like HTTP, TCP, UDP, Websockets, gRPC, or whatever.

The big downside is that each service you expose with a LoadBalancer will get its own IP address, and you have to pay for a LoadBalancer per exposed service, which can get expensive!

Ingress:

Unlike all the above examples, Ingress is actually NOT a type of service. Instead, it sits in front of multiple services and act as a “smart router” or entrypoint into your cluster.

You can do a lot of different things with an Ingress, and there are many types of Ingress controllers that have different capabilities.

The default GKE ingress controller will spin up a HTTP(S) Load Balancer for you. This will let you do both path based and subdomain based routing to backend services. For example, you can send everything on foo.yourdomain.com to the foo service, and everything under the yourdomain.com/bar/ path to the bar service.

Thanks to

Ahmet Alp Balkan

for the diagrams

The YAML for a Ingress object on GKE with a L7 HTTP Load Balancer might look like this:

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

name: my-ingress

spec:

backend:

serviceName: other

servicePort: 8080

rules:

- host: foo.mydomain.com

http:

paths:

- backend:

serviceName: foo

servicePort: 8080

- host: mydomain.com

http:

paths:

- path: /bar/\*

backend:

serviceName: bar

servicePort: 8080

When would you use this?

Ingress is probably the most powerful way to expose your services, but can also be the most complicated. There are many types of Ingress controllers, from the Google Cloud Load Balancer, Nginx, Contour, Istio, and more. There are also plugins for Ingress controllers, like the cert-manager, that can automatically provision SSL certificates for your services.

Ingress is the most useful if you want to expose multiple services under the same IP address, and these services all use the same L7 protocol (typically HTTP). You only pay for one load balancer if you are using the native GCP integration, and because Ingress is “smart” you can get a lot of features out of the box (like SSL, Auth, Routing, etc)

-------------------------------------------------------------------------------------------------------------

ex=>

Deployment file=>

deployment configation and desired states

- API version-

- kind-Deployment ,pod, service,secirets,replica set

- metadata- information about application like name ,author

- Spec:(Specification)=>

replicas=2

Selector=> matchlebels: this is lable of pod used for idetification and it mention in service file

template=> in that we write sepcifiction for container , in thst name ,image ,port

services file => it provide external or interal network to our pods

ex=>

apiVersion: v1

kind: Service

metadata:

name: appsvc1

spec:

ports:

- port: 80

protocol: TCP

targetPort: 80

selector:

app: app1

type: LoadBalancer

under sepc=>

Port: Port is the port number which makes a service visible to other services running within the same K8s cluster. ...

Target Port: Target port is the port on the POD where the service is running.

selector= we give lable same as deployment file

type=> it is type of service

1)cluster ip -for internal pod communication

2)load balancer-external network using DNS

3)Node port=> access external network through Node ip address

label-it is key value pair for identify pods and container

selector-it select the service file on basis of lable