Published in Towards Data Science

David Chong Follow

20022 Jun 19 · · ·

IT DOESN'T HAVE TO BE THAT COMPLICATED, RIGHT?

A Beginner-Friendly Introduction to Kubernetes

. . .

Introduction

In a nutshell, K8s is simply a container orchestration framework. What this essentially means is that K8s is a system designed to **automate the lifecycle of containerized applications** — from predictability, scalability to availability.

. . .

Why do we even need Kubernetes?

The driving reason behind the rise and need for K8s stems from the increasing use of microservices, away from traditional monolithic-type applications. As a result, containers provide the perfect host for these individual microservices as containers manage dependencies, are independent, OS-agnostic and ephemeral, amongst other benefits.

Complex applications that have many components are often made up of hundreds or even thousands of microservices. Scaling these microservices up while ensuring availability is an extremely painful process if we were to manage all these different components using custom-written programs or scripts, resulting in the demand for a proper way of managing these components.

Cue *Kubernetes*.

Benefits of Kubernetes

Kubernetes promises to solve the above problem using these following features:

1. **High Availability** — this simply means that your application will always be up and running, whether you have a new update to roll-out or have some unexpected pods crashing.
2. **Scalability** — this ensures high performance of your application, whether you have a single user or a thousand users flooding your application concurrently.
3. **Disaster Recovery** — this ensures that your application will always have the latest data and states of your application if something unfortunate happens to your physical or cloud-based infrastructure.

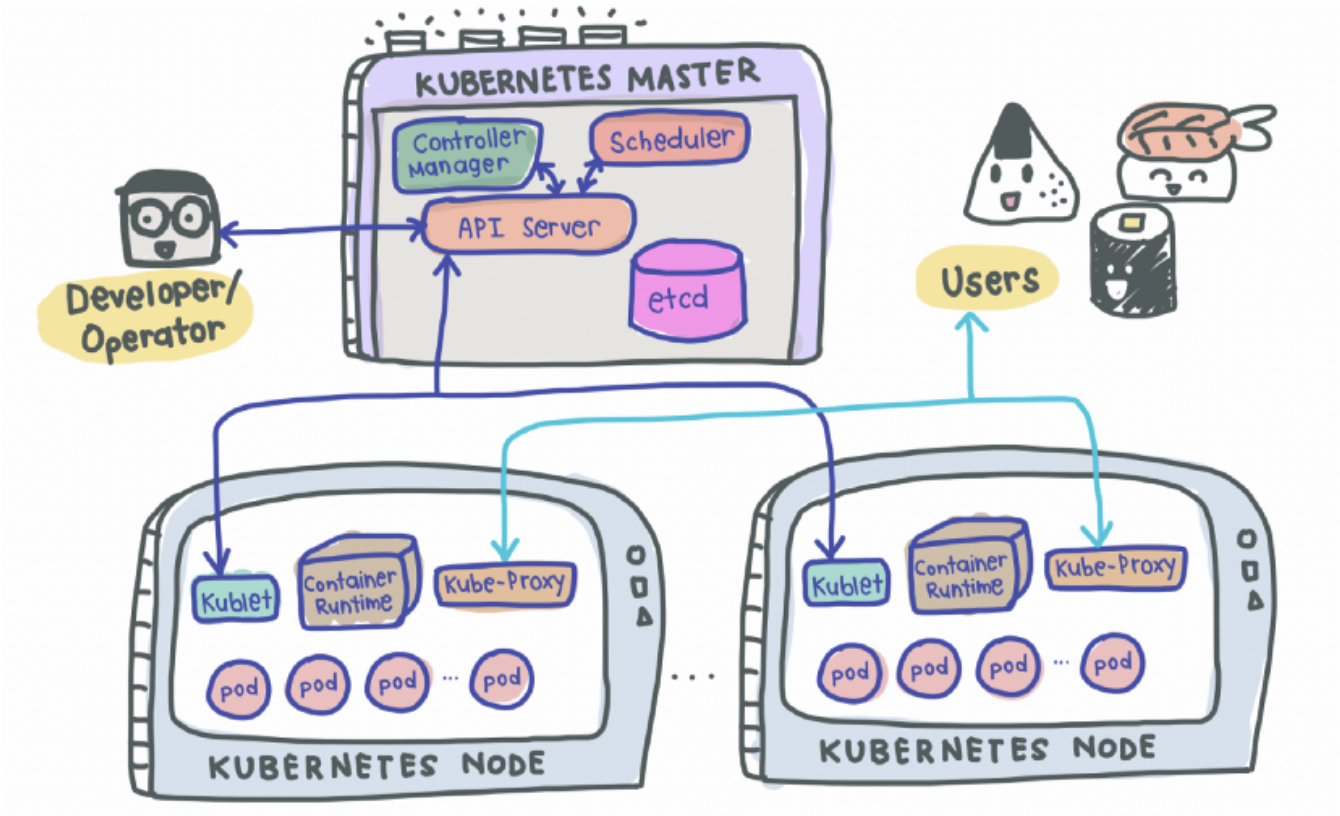
. . .

How It Works Beneath The Hood

K8s uses a Master-Slave type of architecture where a node acts as the Master, calling the shots in the cluster while the other nodes act as slaves/worker nodes, executing application workloads decided by the Master.

A Simple Kubernetes Architecture

A simple K8s setup with a single Master node along with 2 worker nodes look something like this:



Example K8s setup with a single master and two slave nodes (Illustrated by Author)

Master Node(s)

As its name suggests, the Master node is the boss of the cluster, deciding the cluster state and what each worker node does. In order to setup a Master node, 4 processes are required to run on it:

1. API Server

- Main **entrypoint** for users to interact with the cluster (i.e., cluster gateway); it is where requests are sent when we use `kubectl`
- Gatekeeper for **authentication** and request validation, ensuring that only certain users are able to execute requests

2. Scheduler

- Decide which node the next pod will be spun up on but does NOT spin up the pod itself (*kubelet* does this)

3. Controller Manager

- Detects cluster state changes (e.g., pods dying) and tries to restore the cluster back to its original state

- For example, if a pod unexpectedly dies, the *Controller Manager* makes a request to the *Scheduler* to decide which node to spin up the new pod to replace the dead pod. *Kubelet* then spins up the new pod.

4. *etcd*

- Cluster BRAIN!
- Key-Value store of the cluster state
- Any cluster changes made will be stored here
- Application data is NOT stored here, only cluster state data. Remember, the master node does not do the work, it is the **brain** of the cluster. Specifically, *etcd* stores the cluster state information in order for other processes above to know information about the cluster

Slave/Worker Node(s)

Each worker node has to be installed with 3 node processes in order to allow Kubernetes to interact with it and to independently spin up pods within each node. The 3 processes required are:

1. Kubelet a.k.a. `kubelet`

- Interacts with both the node AND the container
- In charge of taking configuration files and spinning up the pod using the container runtime (*see below!*) installed on the node

2. Container Runtime

- Any container runtime installed (e.g., *Docker*, *containerd*)

3. Kube Proxy a.k.a. `kube-proxy`

- A network proxy that implements part of the Kubernetes *Service* concept (*details below*)
- Sits between nodes and forwards the requests intelligently (either intra-node or inter-node forwarding)

. . .

Components of Kubernetes

Now that we know K8s work, let's look at some of the most *common* components of Kubernetes that we will use to deploy our applications.

1. Pod

- Smallest unit of K8s and usually houses **an instance** of your application
- Abstraction over a container
- Each pod gets its own IP address (public or private)
- Ephemeral — new IP address upon re-creation of pod

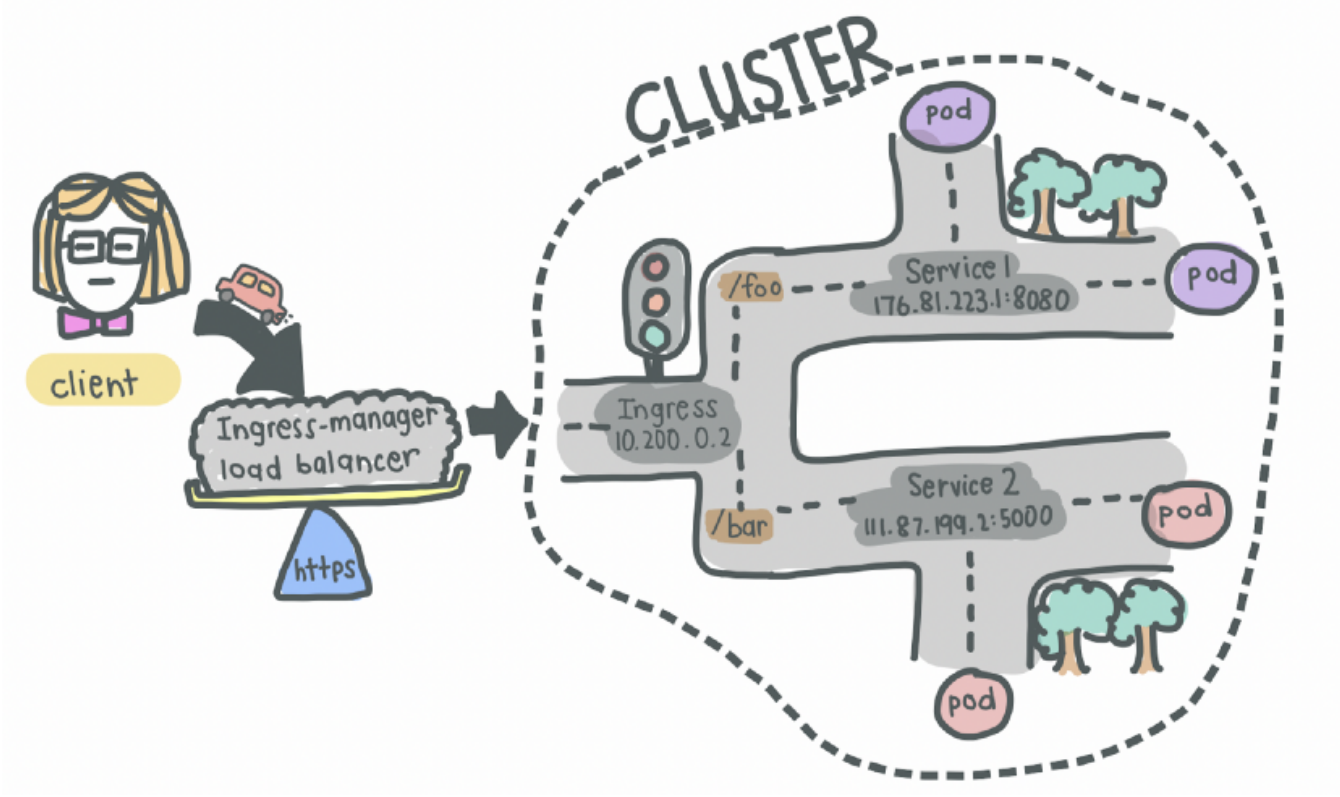
2. Service

- Because pods are meant to be ephemeral, Service provides a way to "give" pods a permanent IP address
- With *Service*, if the pod dies, its IP address will not change upon re-creation
- Acts *almost* as a **load balancer** that routes traffic to pods while maintaining a static IP
- Like load balancers, the *Service* can also be internal or external, where external *Service* is public facing (public IP) and internal *Service* which is meant for internal applications (private IP)

3. Ingress

- With Services, we may now have a web application exposed on a certain port, say 8080 on an IP address, say 10.104.35. In practice, it is impractical to access a public-facing application on `http://10.104.35:8080`.
- We would thus need an entrypoint with a proper domain name (e.g., `https://my-domain-name.com`, which then forwards the request to the Service (e.g., `http://10.104.35:8080`)
- In essence, **Ingress exposes HTTP and HTTPs routes from *outside* the cluster to services *within* the cluster** [1].
- SSL termination (a.k.a. *SSL offloading*) — i.e., traffic to Service and its Pods is in plaintext

- That being said, creating an Ingress resource alone has no effect. An *Ingress-controller* is also required to satisfy an Ingress.



How Ingress works with Ingress Controller (Illustrated by Author)

4. Ingress Controller

- Load balances incoming traffic to services in the cluster
- Also manages egress traffic for services that require communication with external services

> **What is the difference between Ingress and Ingress Controller?**
>
> *Ingress* contains the rules for routing traffic, deciding which *Service* the incoming request should route to within the cluster.
>
> *Ingress Controller* is the *actual implementation* of Ingress, in charge of the Layer-4 or Layer-7 proxy. Examples of *Ingress Controller* include Ingress NGINX Controller and Ingress GCE. Each cloud provider and other 3rd party providers will have their own implementation of the *Ingress Controller*.
>
> A full list can be found here.

5. ConfigMap

- As its name suggests, it is essentially a configuration file that you want exposed for users to modify

6. Secret

- Also a configuration file, but for sensitive information like passwords
- Base64-encoded

7. Volumes

- Used for persistent data storage
- As pods themselves are ephemeral, volumes are used to persist information so that existing and new pods can reference some state of your application
- Acts almost like an “external hard drive” for your pods
- Volumes can be stored locally on the same node running your pods or remotely (e.g., cloud storage, NFS)

8. Deployment

- Used to define blueprint for pods
- In practice, we deal with deployments and not pods themselves
- Deployments usually have replicas such that when any component of the application dies, there is always a backup
- However, components like databases cannot be replicated because they are stateful applications. In this case, we would need the Kubernetes component: **StatefulSet**. This is hard and more often than not, databases should be hosted outside the Kubernetes cluster

. . .

. . .

. . .

. . .