Kubernetes Components explained

By TechWorld with Nana

TechWorld with Nana

Table of Contents

TechWorld with Nana

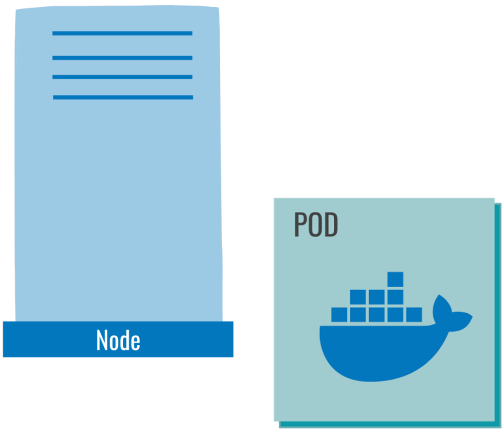
Many Kubernetes Components

Kubernetes has **tons of components**, but most of the time you work with just a handful of them. With this overview of the main Kubernetes components, you will learn the most important ones to get started with Kubernetes as fast as possible.

WHAT YOU WILL LEARN 💡

I built a case of a simple JavaScript application with a database. With this example I explain **step by step how each component** of Kubernetes actually helps you to deploy your application and **explain what role each of these components has**.

Node and Pod



Let's start with the basic setup of a Worker Server or in Kubernetes terms a **Node**. A Node is a simple server - it can be a physical or a virtual machine that has the resources to run containers and pods.
Pod is the basic component and the smallest unit of Kubernetes. So in Kubernetes you never work with containers, you work with Pods.

Pod is an **abstraction over a container**. A Pod creates a running environment or a layer on top of the container, e.g. Docker container. 🐳

The reason why Kubernetes abstracts away the container runtime or container technologies is, that you can easily replace them if you want to without making changes to the existing Kubernetes configuration. So you don't have to directly work with Docker or whatever container technology you use in Kubernetes.

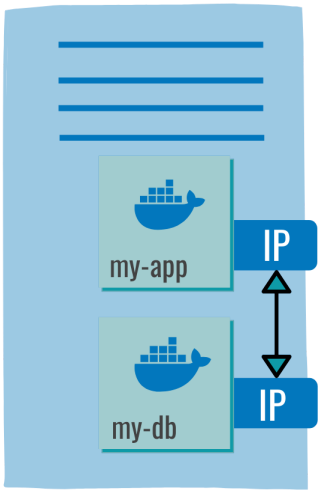
This means, you **only interact with the Kubernetes layer**.

A Pod is usually meant to run **one application container** inside of it. While you can run multiple containers inside one pod, this is usually only the case if you have one main application (Mysql) and a helper container or some side service (Metrics collector, Backup app) that has to run inside that pod.

Imagine we have our own application (“my-app”) deployed in the cluster, that uses a database Pod (“my-db”).

This setup is nothing special, you just have one server and there are two application Pods running on it.

Now let's see how they communicate with each other in the Kubernetes world.



Communication between Pods 💬

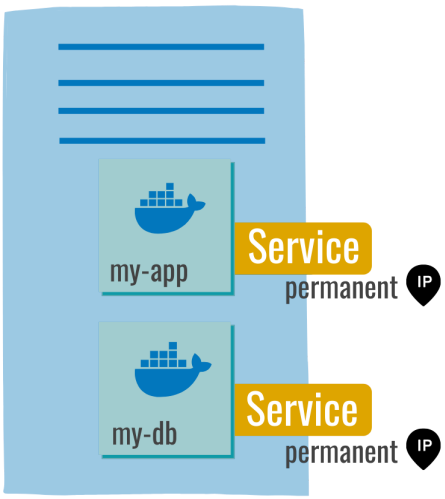
Kubernetes offers a virtual network layer that spans all the cluster nodes, which means that **each Pod gets its own IP address**. Each Pod can communicate with each other using that IP address, which is a cluster internal IP address.

So “*my-app*” application container can communicate with the database “*my-db*” using the Pod IP address.

Pods are ephemeral

However, Pod components in Kubernetes are ephemeral, which means that they can **die very easily** ☠ This can happen, when the server runs out of resources or the application crashes or the server gets restarted.
When a pod dies, a new one will get created in its place. And it will get **assigned a new IP address**. This is obviously inconvenient if you are communicating with the database using the IP address. Because now you have to adjust it every time the Pod restarts. 🙄
To solve that, another component of Kubernetes called Service is used.

Service and Ingress



Service is basically a static IP address or **permanent IP address** that can be attached to each Pod. So *my-app* will have its own Service and the database Pod *my-db* will have its own Service.

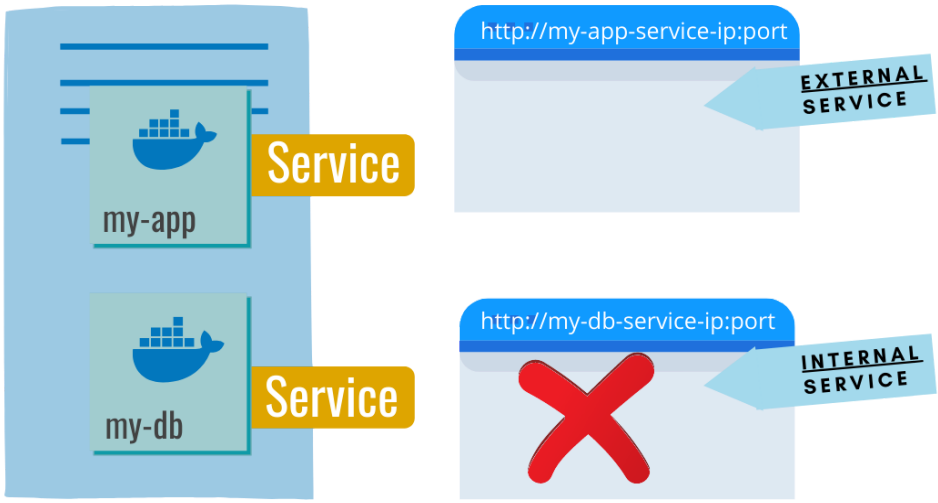
The life cycles of Service and the Pod are **not** connected to each other. Therefore, if the Pod dies, the **Service and its IP address will stay**. So you don't have to change that endpoint anymore. 😎

Internal vs. External Service

The next step would be to make your application accessible through a browser. To achieve this you need to create an **external Service**. External Service is a Service that makes the **application accessible** from **external clients**, like a browser.

However, you don't want your database to be open to the public requests. 🧐 For that you would create an **internal service**.

When creating a Service, you specify the type of the Service.

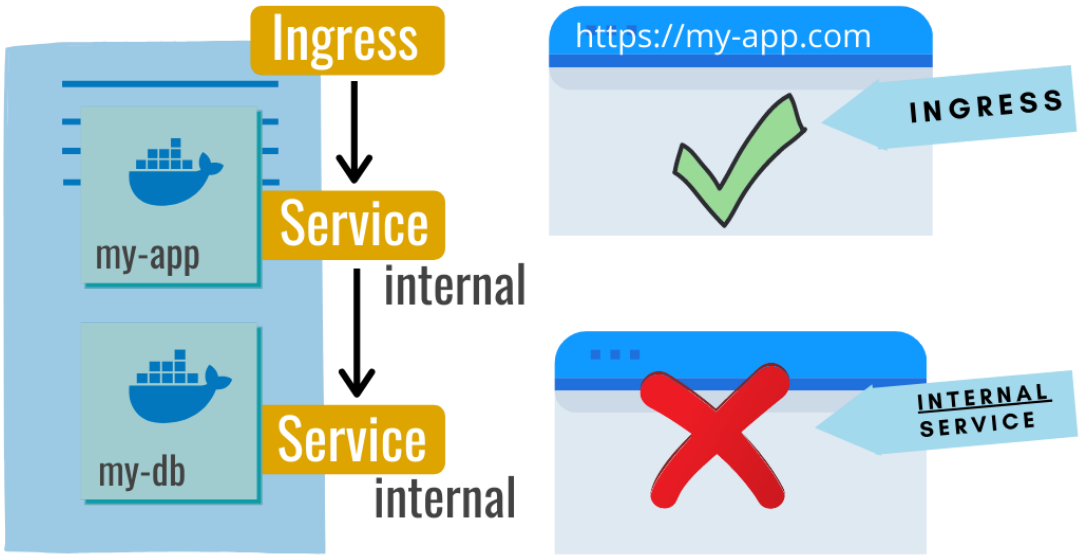


Ingress

The URL of the external Service would look something like this: *http://124.89.101.2:8080* - the HTTP protocol with the Worker Node IP address and the port number of the Service. This URL is good for test purposes, for example if you want to test something very fast, but not for the end product. 🙈

Usually you want your URL to look like this: *https://my-app.com* With a **secure protocol** and a **domain name**.

You achieve that with another component of Kubernetes called Ingress. Ingress is the **entrypoint** to your **Kubernetes cluster**. So, instead of a Service, the request goes first to Ingress, which does the forwarding to the Service.



ConfigMap and Secret

As I mentioned earlier, Pods communicate with each other using a Service, so "*my-app*" will use a database endpoint, "*my-db-service*" to communicate with the database.

ConfigMap

Usually you would configure this database URL or endpoints in an **application properties file** or some kind of **external environmental variable**. So, it's usually packed **inside the container image** of the application.

Problem

This means, if the Service name or endpoint changes, you would have to **adjust that URL** in the application. Meaning you would have to rebuild the application image with a new version and you have to push it to the repository and then pull that new image in your cluster and restart the pod. 🙄 This process is too tedious for a small change like the database endpoint name.

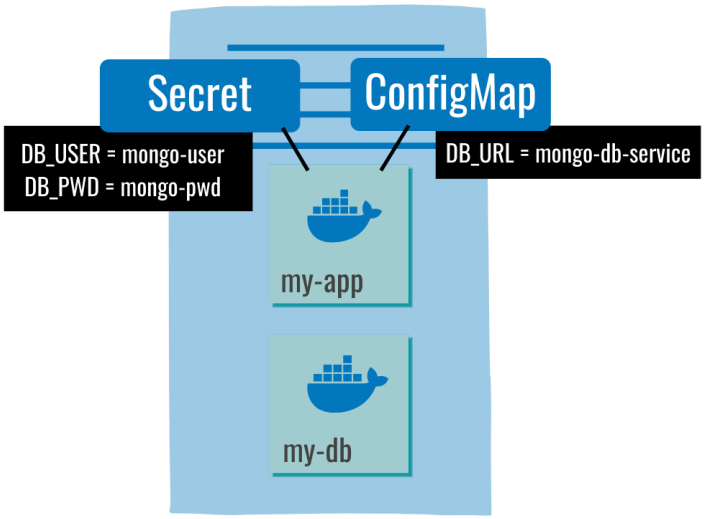
Solution ✔️

To solve this problem, Kubernetes has a component called ConfigMap. ConfigMap is basically an **external configuration** for your application. It usually contains configuration data like URLs of databases or some other Services that your application uses.

You create a ConfigMap independently of the pod and then pass it to the pod on startup. Now if you change the name of the service endpoint, you just **adjust the ConfigMap**, restart the pod and that's it. 😎 You don't have to rebuild an image and go through this whole cycle.

Secret 🔐

Part of the external configuration can also be database or other service credentials, which may also change. But putting a password or other **credentials** in a ConfigMap in plain text format would be **insecure**.



For this purpose Kubernetes has another component called Secret. Secret is just like ConfigMap, but the difference is that it's used to store secret data, like credentials or certificates. It's stored in base64 encoded format.

Secrets would contain things that you don't want other people to have access to. 🤫

Just like ConfigMap you just **connect it to your Pod** so that Pod can actually see those data and read from the Secret.
You can use the data from ConfigMap or Secret inside your application Pod either as environmental variables or as a properties file.

Storing the data in a Secret component doesn't automatically make it secure. There are built-in mechanisms (like encryption, defining authorization policies) for basic security, which are not enabled by default! Many also use third-party secret management tools, because the provided capabilities by Kubernetes are not enough for most companies.

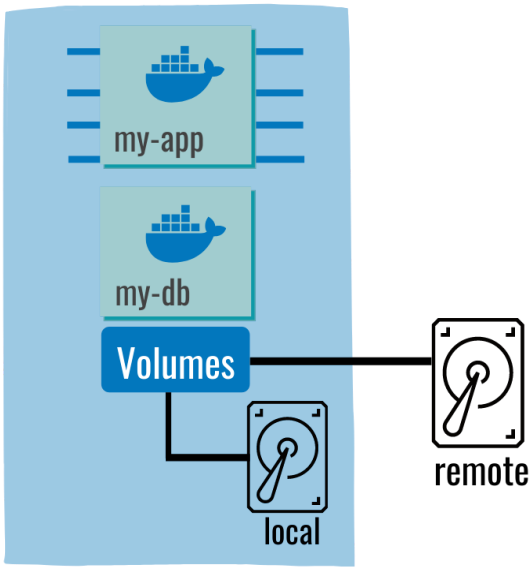
Volumes

Kubernetes **doesn't provide data persistence out of the box**, which means when a Pod dies, the data is lost. 🤯 That's of course problematic and inconvenient, because you want your database data or log data to be persisted reliably and long-term.

The way you can do it in Kubernetes is using another component called Volumes.

How it works

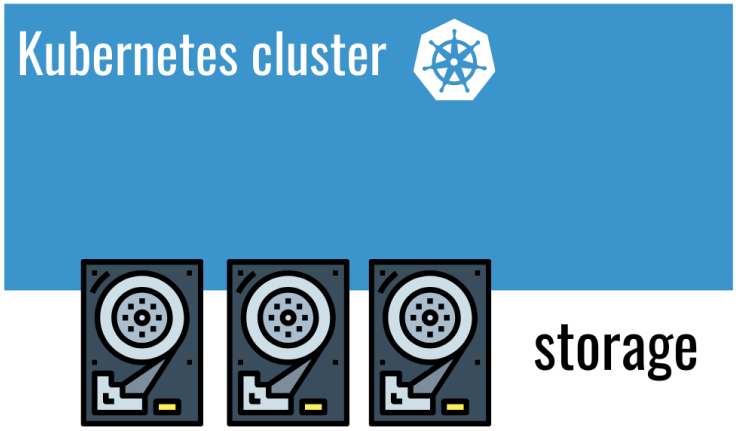
The Volume component basically attaches a physical storage on a hard drive to your Pod. That storage could be either on a local server, meaning on the same server Node, where the Pod is running or it could be on a remote server, outside the Kubernetes cluster. This remote storage can be a cloud storage or your company's on-premise storage, which is not part of the Kubernetes cluster. So you just have an **external reference** to it.



With volumes the application data will be persisted between the pod restarts. 👍

It's important to understand 💡 the relation between the Kubernetes cluster and the storage, whether it's a local or a remote storage.

Think of storage as an **external hard drive plugged in** to your Kubernetes cluster. The point here is, that Kubernetes explicitly doesn't manage any data persistence.



This means that you as a Kubernetes user or an administrator are responsible for backing up the data, replicating and managing it and making sure that it's kept on a proper hardware etc.

Deployment and StatefulSet

Problem with a single Pod setup 🧐

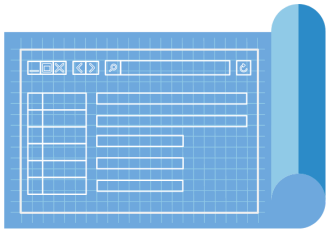
Let's go back to our example setup with an application pod and a database pod. With Ingress and services configured, users can access our application through a browser. So far, so good!
But with this setup, when "my-app" Pod dies, there will be a downtime, where the users can't reach the application. 😱 This downtime is obviously a very bad thing if it happens in production and avoiding or minimising these downtimes is exactly the advantage of distributed systems and containers.

Solution 💡

Instead of relying on just one application Pod and one database Pod, we are **replicating everything on multiple servers**. So, we would have another Node, where a replica or clone of "*my-app*" and "*my-db*" applications will run.

"*my-app*" replica pods will then share the my-app service and the same goes for the "*my-db*" pods. So the **Service is also a load balancer**, which will catch the request directed to the application and forward it to one of its replica pods.

Deployment - blueprint for Pods



But in order to create the second replica of the *my-app* Pod you wouldn't create a second Pod, but instead you would **define a blueprint** for *my-app* and **specify how many replicas** of that Pod you would like to run.

And that component or that blueprint is called Deployment, which is another component of Kubernetes. In practice you would not be working with Pods, but with Deployments. In Deployments you can specify how many replicas you need and you can also **scale up or scale down the number of replicas** you need.

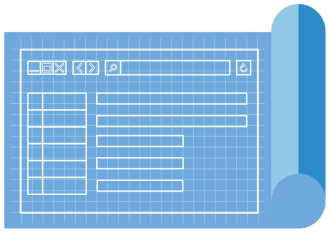
Also if you need to update the docker image for *"my-app"*, you don't have to adjust it in each pod. You update it in the deployment and deployment will then automatically update and restart all running pods.

Pod is a layer of abstraction on top of containers and Deployment is another abstraction on top of Pods, which makes it more convenient to interact with the Pods, replicate them and do some other configuration.

Now, after replicating *"my-app"*, if one of the replicas died, the Service would forward the requests to another one. So your application would still be accessible for the users.

StatefulSet - blueprint for stateful applications

Now you're probably wondering what about replicating the database, because if the database Pod died your application also wouldn't be accessible. So, we need database replicas as well.



Problem 🤔

However we can't replicate a database using a Deployment and the reason for that is, that the database has a **state** and **data**.
If we have clones or replicas of the database, they will all access and update the same data storage. So you would need some mechanism that manages which replicas are currently writing to that storage and which ones are reading from that storage, in order to **avoid data inconsistencies**.

Solution 💡

Support for that mechanism, in addition to replicating features, is offered by another Kubernetes component called StatefulSet. This component is meant specifically for applications like databases: Mysql, MongoDB, Elasticsearch and other stateful applications.

It's a very important distinction and StatefulSet just like Deployment would take care of replicating the Pods and scaling them up or down, but in addition support that **database reads and writes are synchronized**, so that no database inconsistencies happen.

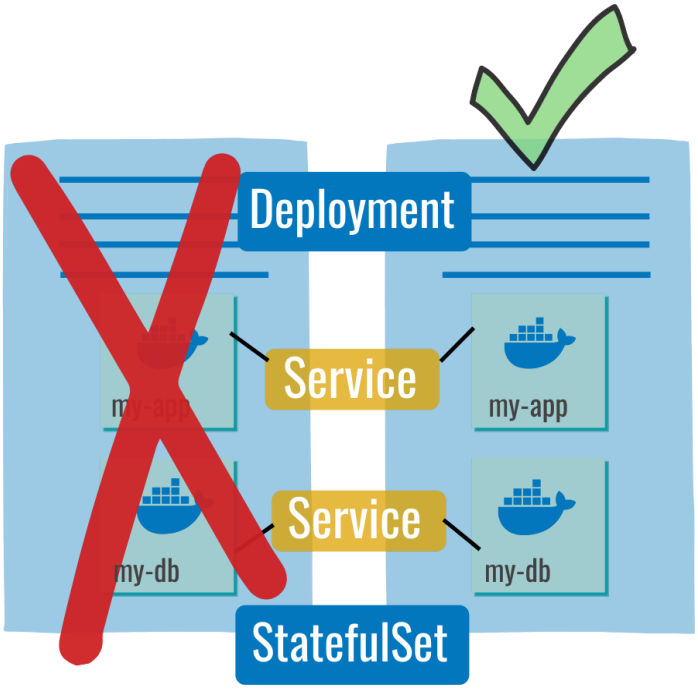
Host stateful applications outside the Kubernetes cluster

However, I must mention here that replicating database applications using StatefulSet in Kubernetes clusters can still be difficult. It's definitely more difficult than working with Deployments, where you don't have all these challenges. 😮

Some database applications, like elastic search, have part of the replication logic implemented inside the application itself, so less needs to be done and configured from outside.

For other databases, like Mysql, which doesn't support replication out of the box, it's also a common practice to host them outside the Kubernetes cluster and have the stateless applications communicate with these external databases.

Now that we have two replicas of *my-app* Pod and two replicas of the database *my-db* and they are both load balanced our **setup is more robust**. This means that even if one Node server was rebooted or crashed and nothing could run on it, we would still have a second Node with application and database Pods running on it. And the application would still be accessible for the users. 🤘



Main Kubernetes components summarized

To summarize we have looked at the most used Kubernetes components.

- ★ **Pods** are the smallest unit in Kubernetes, wrapping the containers.
- ★ **Services** are used in order to communicate between the Pods and the **Ingress** component is used to route traffic into the cluster.
- ★ We have also covered external configuration using **ConfigMaps** and **Secrets** for sensitive configuration data.
- ★ **Volumes** are used for data persistence.
- ★ Finally, we've looked at Pod blueprints with replicating mechanisms like **Deployments** and **StatefulSet**, where StatefulSet is used specifically for stateful applications like databases.

There are a lot more components that Kubernetes offers, but these are the core or basic ones. Just using these core components, you can actually build pretty powerful Kubernetes clusters. 🚀