# Introduction

In short Kubernetes allows us to manage multiple applications running across multi node cluster.

It helps with allocating computer resources (CPU, RAM etc.) to applications. In that aspect, it is like a kernel but for a cluster consisting of multiple servers.

It gives us a possibility of deploying multiple applications as containers by using YAML files where we provide a specification about how those containers need to be created.

It also helps with networking between containers. Each container can get assigned easily a static IP address and a DNS name.

# Materials to learn from

* [youtube - TechWorld with Nana](https://www.youtube.com/watch?v=X48VuDVv0do) – 4h Kubernetes beginner tutorial.

# CronJobs

It is a scheduler which can orchestrate deploying Pods for batch tasks. That means that those Pods are supposed to execute some task relatively quickly and stop. It is not for deploying long running applications.

# ConfigMap

ConfigMap is a resource which we can use to provide an external configuration for an application running as a container in a Pod.

We can store there data asf key value pairs which will be used by our app running in Pod, and we can change them without changing the app itself.

# CoreDNS

It is a DNS server. It translates domain (DNS) names into IP addresses inside of a cluster. That means that once we assign a DNS name to a Pod then every other Pod from a cluster can resolve it.

Every Pod get assigned a default DNS name like this:

* <pod-ip-address>.<namespace>.pod.cluster.local

We can also use a Service resource in order to assign a static IP address and our chosen domain name to a Pod. Then everytime we recreate that Pod it will have always the same domain name.

# Deployment

It is a blueprint, set of instructions for creating Pods. It is abstraction of Pods. It specifies which app we want to run (which container), how many replicas of that Pod to create etc.

# Helm

Helm is a package manager for Kubernetes that helps you define, install, and manage Kubernetes applications.

## Sharing Helm charts

Helm uses charts, which are packages of pre-configured Kubernetes resources written in YAML. They can be uploaded and downloaded from repositories.

## Templating YAML files

Helm also allows for creating YAML files which are using variables and we can put different values into them.

## Example chart

**File structure**

Our chart might look for example like that:

A screenshot of a computer program

AI-generated content may be incorrect.

In the templates folder we have manifests (templates) of resources which we want to deploy.

**Chart.yaml**

In the Chart.yaml file we can specify dependencies, that is what other Helm charts we want to install except for resources from the templates folder. Here for example we install MySQL chart from the bitnami repository:

A screenshot of a computer

AI-generated content may be incorrect.

**Values.yaml**

In the values.yaml file we are specifying variables which will be used in manifests in the templates folder.

For example we can have a variables like this:



And refer to it in a manifest like this:



**Dependency parameters**

Also we can provide in the values.yaml values for parameters for isntalling dependencies. In our example we have MySQL as a dependency. We can provide values in the values.yaml for parameters for this dependency like this:

A screen shot of a computer

AI-generated content may be incorrect.

Here the name ‘mysql’ needs to be the same as name of the dependency.

That will be equivalent to installing MySQL using Helm this way:

A screen shot of a computer program

AI-generated content may be incorrect.

## Helpers

Helpers are named templates which allow us to create named configurations which we can use in our YAML files by using their name.

Helpers are defined in the templates/\_helpers.tpl file.

For example we can define a helper like this:

A computer screen with text

AI-generated content may be incorrect.

That created a helper called ‘mlflow-job.labels’ and its content are those two lines between {{-define .. -}} and {{- end }} blocks.

And use it in a YAML file like this:

A computer screen with text on it

AI-generated content may be incorrect.

So in place of a helper name will be inserted its content like this:

A computer screen with text and images

AI-generated content may be incorrect.

# Ingress

Ingress is a resource which we can use to specify which URL will be redirecting to which Service when making a HTTP request.

# Kubectl

This is a tool for users to interact with a Kubernetes cluster. It allows for example to get list of Pods, create, update or delete resources.

# Master node vs worker node

A master node is used to orchestrate containers with applications running on worker nodes.

Worker nodes are used to run containers.

When user wants to run a container, they communicate with a master node, and master node then communicates with worker nodes to run that container.

We have different processes running on a master node and worker nodes. They are explained in more detail in subsections below.

## Processes running on master and worker nodes

On a master node we have different processes running than on worker nodes.

The main processes we are running on a master node are:

* API Server
* Scheduler
* Controller manager
* ETCD

While on a worker node the main processes are:

* Container runtime
* Kubelet
* Kube proxy

### Container runtime

Each node needs to have installed a container runtime, like docker or containerd, which will be running containers.

### Kubelet

It is a process running containers on nodes. It interacts with a container and node. It creates containers based on provided configuration, YAML manifests.

### Kube proxy

It forwards requests from Services to Pods.

### API server

Users interacting with Kubernetes cluster are using the API server. It takes requests from users about what they want to do with a cluster, and then API server talks to other processes running on Kubernetes.

It is also used for authentication.

### Scheduler

API server talks to the Scheduler and Scheduler decides on which node to run requested Pods. It check how much resources (CPU, RAM) will be needed for running a Pod, and which nodes have those resources available.

Scheduler doesn’t run the app itself but it talks to the kubelet and kubelet runs an app.

### Controller manager

It monitors the state of Pods, if they are running correctly or if they have died. If a Pod died then Controller manager talks to the Scheduler in order to rerun a Pod.

### Etcd

Etcd Stores data about all the changes that happens in a Kubernetes cluster, for example when Pod dies, and when a new Pod gets created.

It provides data used by other servicer (API server, Scheduler, Controller manager).

# Namespace

Namespaces are a way for organizing resources in Kubernetes. They are groups containing specified resources.

Each namespace doesn’t have separate resources (nodes, CPU, RAM). They share common, cluster’s resources. They just provide a way to logically group resources.

## The kube-system Namespace

This namespace contains all the system pods which are running required Kubernetes services like kube proxy, kube scheduler, api server etc.

# Node

Node is a single computer in Kubernetes cluster (a group of nodes).

# Operators

Operators can manage applications (resources) in Kubernetes. For example, in case of PostgreSQL it can:

* Create a PostgreSQL instance
* Monitor its health
* Automatically handle failures
* Manage backups

Or in case of Spark Operator:

* It watches for the SparkApplication custom resource
* Once that resource is created it creates a Spark Driver Pod

Or it can:

* Watch for MySQL secret to be created
* Once it is created Operator fetches a value from this secret and saves it in another secret

# Pod

It is a resource which is the smallest unit in Kubernetes. It is an abstraction of a container. Inside of a Pod we have a container running with app, and that Pod is running on one of the nodes.

We don’t work with the container itself in Kubernetes, just with a Pod or abstractions of Pod (for example Deployment).

# podTemplateFile

podTemplateFile is used in custom resource definitions (CRDs) to provide additional parameters which we are not able to provide in the CRD itself.

podTemplateFile specifies a path to another YAML manifest where we specify additional parameters we want to have in our CRD.

# Resources

In Kubernetes we are creating resources which are objects with different purposes, which we manage and which helps us to manage our applications.

Those are tools used for running our application, for example we might have the following resources:

* Pod – containing a container running our app
* Service – Assigning a static IP address to the container running our app
* Volume – mounting external storage to the container running our app

More resources are mentioned and explained in further sections of this documentation.

## Manifests

Manifests are conifguration files (mainly YAML) which defines what resources we want to create in Kubernetes cluster.

We are preparing Manifest first and then we apply it in order to create the specified resource.

## Custom resources

Standard resources and available by default in every Kubernetes cluster and Custom resources are additional ones added by using the CRD (Custom Resource Definition) which is a YAML manifest specifying that resource.

# Replicas

We can have multiple replicas of the same app (Pod). That means that we have multiple copies of the same app running. It is useful because when one app goes down, then the other one can take its place in handling requests.

We can use a Service (described in the previous section) to create a static IP address, and each request made to that IP address will be redirected by a Service to a replica which is the least busy.

# Service

Service is a resource which we can assign to a Pod and it provides a static IP address to a Pod.

Pods have its own IP address, which can change over time, and Service has a separate, static IP address and it is independent on the IP address of the Pod.

In Service we can also specify a domain name for a Pod.

It also acts as a load balancer. We can have two replicas (described in the next section) of the same app (Pod), both attached to the same service, and service will redirect a request to the replica which is the least busy.

# Service Account

Service Account is a special kind of account used by rpocesses running in pods to interact with the Kubernetes API. It’s like a “user identity” for pods.

**Purpose**

A Service Account provides authentication when Pod wants to read/write Kubernetes resources (e.g., secrets, config maps, jobs).

**Default Service Account**

Every namespace has a default service account. If we don’t specify one, pods automatically use it.

**Custom Service Account**

We can create a service account with specific **roles/permissions** via Role-Based Access Control (RBAC).

For example:

Create a Service Account:

A screen shot of a computer

AI-generated content may be incorrect.

Attach it to a Pod or Deployment:

A black background with pink text

AI-generated content may be incorrect.

Assign role to the Service Account:

A screen shot of a computer

AI-generated content may be incorrect.

# StatefulSet

It is like a Deployment but specifically designed for applications which have a state. For example databases have a state which is their data (which changes over time). We can’t simply delete and recreate such an app because we will loose that state.

Even if we are using volumes for our databases running in Pods, then still we should use the StatefulSet for them to guarantee data consistency.

# Secret

Secret is a resource like ConfigMap but for storing confidential data.

We use this command to create a secret:

* Kubectl create secret <type> <name> [options]

Using this command we can:

* Define a secret value during executing this command
* Read secret value from a file
* Create a secret for a specific container registry which can be used for pulling images from it. It generates a new secret value based on credentials to a container registry which we pass in that command.
* Get TLS key and certificate from files.

# Taints

Taints are a way to control which Pods can be scheduled on which Nodes.

We can create a taint on a Node and then every Pod that doesn’t tolerate it will not be able to be scheduled on that Node.

In a Pod’s manifest we can specify which taints it tolerates and then we can create that Pod on Nodes which have those taints.

Every Taint consists of 3 parts: key, value and effect. Key and value identifies a Taint (unique identifier of a Taint) and effect defines what happnes when we try to schedule a Pod on a Node that has a Taint not tolerated by that Pod.

# Volumes and volumeMounts

Volumes are a resource which works like volumes in Docker. Volumes allow us to save data from a container outside of a container (on a host running a container).

So container will be using this data and even after container restart, that data will be still accessible.

Volumes need to be used together with VolumeMounts which will specify the path in the container where data from a Volume will be accessible.

We can additionally use the hostPath parameter to specify where data from a Volume will be stored on a host.

For example we might use Volumes in a YAML manifest in this way:

A screenshot of a computer

AI-generated content may be incorrect.

The other way of doing this is to create a separate Volume resource. Here is an example of Persistent Volume (it preserves data even after Pod is deleted). We create here separately one resource for Volume and one for creating a Pod which uses that Volume:

A screen shot of a computer

AI-generated content may be incorrect.

And yet another option is that we don’t create a Volume resource on our own, we just create a volume claim (PersistentVolumeClaim) and it will create a volume resource automatically:

A screen shot of a computer

AI-generated content may be incorrect.

## Volume with configMap

We can use a configMap to create a volume and mount it to a Pod.

This way in the path inside the container specified by the ‘mountPath’ parameter we are creating:

* A file for each item from the ConfigMap
* Each file has the same name as name of the key from the ConfigMap item
* Each file called ‘key-name’ has content equal to value from ConfigMap corresponding to the key called ‘key-name’

So for example if we create a ConfigMap by reading content of a file:

A screen shot of a computer program

AI-generated content may be incorrect.

So here we create an item in ConfigMap with key-name = ‘filename.txt’ and value for that key is a content of the /home/username/filename.txt file.

And then we create a Volume and volumeMount like this:

A screen shot of a computer program

AI-generated content may be incorrect.

Then inside that container we can access the content of the /home/username/filename.txt under the /etc/config/filename.txt path.

## Volume claims

As mentioned earlier in the ‘Volumes and volumeMounts’ section, we can use volume claims to attach a volume to our Pod, for example the PersistentVolumeClaims resource.

If we deploy a Volume claim, then we need to have either a volume prepared or we need to have a dynamic storage provisioner which is explained in another section.

## Dynamic storage provisioner

A Dynamic storage provisioner automatically creates a volume when a volume claim is deployed.

There are multiple options for this:

* NFS
* Ceph + Rook

# Webhooks

Webhook is a tool used for validation or modification of other resources during their creation or update.

They are configured using the MutatingWebhookConfiguration and ValidatingWebhookConfiguration resources.

The Webhook server usually runs inside of its own Pod.

Always when we create a resource in Kubernetes, Kubernetes API server checks if there are any configurations for that resource (MutatingWebhookConfiguration and ValidatingWebhookConfiguration).

If Webhook configurations are found for a resource, then Kubernetes API server communicates with Webhook using HTTPS through the Webhook’s service.