**K8s notes**

**K8s** is an open source container orchestration tool (initially developed by Google)

**Rancher** isa platform for managing multiple Kubernetes clusters

**Kubeadm** isa command-line tool for bootstrapping a Kubernetes cluster

**Minikube** is atool to run a k8s cluster locally

**K8s features:**

1. High availability (no downtime)
2. Scalability (high performance)
3. Disaster recovery (backup and restore)

**K8s components:**

1. Node 🡪 a simple server (physical or virtual machine). We use multiple nodes for replication
2. Pod 🡪 the smallest unit of k8s (an **abstraction layer** over a container. It creates the running environment for containers) 🡪 pod is mainly used to run **one** application inside it.

Each pod gets an internal IP address and communicates with each other.

If a pod crashes, it gets recreated and gets a new IP address

1. Service 🡪 a static (permanent) IP address with a **DNS name**, that can be attached to each pod. The lifecycle of service and pod is not connected. If a pod crashes, the service stays the same (IP address).

It is also a load balancer, meaning that the service will actually catch the request and forward it to the least busier pod. By default, the services are **internal (type: ClusterIP)**.

External service 🡪 a service that opens the communication from external sources (access to outer world)

1. Ingress 🡪 It is used to route traffic into the cluster. Instead of a service, the request goes first to the ingress and this does the forwarding to the service (uses service name instead of IP and https instead of http)
2. ConfigMap 🡪 an external configuration for the application (it contains configuration data like URLs of databases). No need to build new images if a url changes. Do not put credentials into ConfigMap
3. Secret 🡪 it is like ConfigMap but it is used to store secret data credentials and it is stored not in a plain text format, but in base64 format
4. Volumes 🡪 it attaches a physical storage on a hard drive to the pod. The storage can be local, on the same server node where the pod is running, or a remote storage (outside of the k8s cluster). K8s does not manage data persistence!
5. Deployment 🡪 a blueprint for pods, where I specify how many replicas of pods I need. It is an abstraction over pods. We cannot replicate a database via a deployment because a database has a **state** (all pods use the same storage)
6. StatefulSet 🡪 it is specifically meant for stateful applications (like databases). It takes care of replication and scale up or down, but avoids data inconsistencies. It is more difficult to deploy them than deployments. It is best to keep the databases **outside** of the k8s cluster (external)

**K8s architecture**

1. Worker servers (nodes) 🡪 they are the cluster servers that actually do the work. Each node will have multiple application pods with containers. K8s uses 3 processes that must be installed on every node that are used to schedule and manage those pods.
   1. Container runtime 🡪 docker or another technology. It needs to be installed to run the containers that are inside the pods.
   2. Kubelet 🡪 a process of k8s that has interface with both container runtime and the node itself (the machine). It is responsible for taking the configuration and running a pod with a container inside, and then assigning resources from that node to the container, like CPU, RAM, storage

(Services 🡪 the way of communication between nodes. It acts as a load balancer that catches the request directed to the pod and then forwards it to the respective pod)

* 1. Kubeproxy 🡪 it has intelligent forwarding logic inside that makes sure that the communication works in a performant way (for example if a pod sends a request to the db, kubeproxy will send it to the pod in the same node, avoiding the network overhead of sending it to another node)

1. Master servers (master nodes) 🡪 the managing servers that control the cluster state and the worker nodes. They have the following processes:
   1. API server 🡪 the main entry point into the cluster. When I want to deploy a new application in the cluster, I interact with the API server using a client (like UI (like k8s dashboard), or CLI tool like kubelet or a k8s API). It acts as a **cluster gateway** which gets the request for changes into the cluster. It also acts as a gatekeeper for authentication. When I want to schedule new pods, or create new services etc, I have to talk to the API server (with authentication). Also if I want to query the status of cluster health etc, I make a request to the API server. The API server is load balanced.
   2. Scheduler 🡪 After the request to start a new pod passes through the API server, it is handed to the scheduler in order to start the application pod on one of the worker nodes (in an intelligent way, in the node with least usage). Kubelet gets the request from the scheduler and executes the request on that node
   3. Controller manager 🡪 it detects state changes like crashing of pods. If a pod died, it detects it and tries to recover the cluster state as soon as possible. It makes requests to the scheduler to reschedule the dead pods, which then decides where to restart those pods (in what worker node)
   4. etcd 🡪 A key value store of the cluster state (the cluster brain). Every change in the cluster (when a new pod is scheduled or another dies), is saved or updated into this key value store. All of the information about the cluster is stored in the etcd cluster. Application data is not stored inside there. The etcd forms a distributed storage across all master nodes.

**What is Minikube**

It is atool to run a k8s cluster locally. It is a 1 node k8s cluster, where the master processes and worker processes both run on **one machine (node)**. This node has docker container runtime pre-installed. It runs through a virtual box or another hypervisor. It is used for testing purposes.

**What is kubectl**

It is a CLI tool for k8s cluster. It is the way to interact with a k8s cluster (**API server**). Alternatives to kubectl are the k8s UI, the k8s API, but kubectl is the most powerful.

**Creating a deployment**

Deployments manage pods 🡪 I do not create pods, I create deployments which is an abstraction over pods

When I create a deployment, it has all the information (blueprint) for creating the pod

A deployment manages a replicaset 🡪 a replicaset manages all the replicas of a pod 🡪 a pod is an abstraction of containers

Replicaset 🡪 it is managing the replicas of a pod. I usually do not create/delete/modify replica sets

I can create deployments with kubectl (kubectl create deployment –name –image –option1 –option2 etc), but this is impractical. I can better do it with k8s configuration files.

**Kubectl apply -f <filename> (-f means file)**

It takes a file and applies it, runs all configuration described there

**K8s configuration file**

Each configuration file has 3 parts

1. kind 🡪 deployment, service etc
2. metadata 🡪 it contains **labels**
3. spec (specification) 🡪 Attributes of spec are specific to the kind (for example deployment, service). It contains **selectors**.

Inside the spec, I can have a template 🡪 another config file inside the config file. It has its own metadata and spec section. This configuration applies to the pod (this is the blueprint of the pod)

1. status (automatically generated and added by k8s) 🡪 K8s compares the desired and actual state of the component. If they do not match, k8s tries to fix it (self-healing feature). K8s updates the status continuously. This information comes from the etcd, which holds the current status of any k8s component

**Connecting Deployments to Pods**

With labels, we tell the **deployment** to match all the **components** with a specific label, which is matched by the selector of the pod.

**Connecting Services to Deployments**

The deployment has its own label, which is used by the service selector, which makes a connection between the service and the deployment

**Ports in service and Pod**

Service has a port where it is accessible at. The service needs to know in which port to forward the request (targetPort). This should match the container port

**Secrets**

They live in k8s, not in the repository. When creating secrets files, I save the credentials in base64 format, not plain text.

Secrets must be created **before** the deployment to avoid errors.

**External Service**

This is only for testing a service and not for production.

I declare the service as usual, and to make it external I set the **type: LoadBalancer**, and in ports I define a **nodePort**. (the external port accessible from outside)

NodePort has a specific range (30000-32767)

In minikube, when I create an external service, it does not have an external-IP yet (it is in status <pending>). In order to assign the external service a public IP address, I have to run it from terminal.

**Namespaces 🡪** a virtual cluster inside a cluster

Resources can be organized in namespaces. There can be multiple namespaces in a cluster.

For example, I can group my resources to the following namespaces:

Database

Monitoring

Nginx-Ingress Controller

ELK

Backend

Some components cannot be created in namespaces (for example volumes and nodes)

**Resource Quotas** 🡪 they limit how many resources a namespace can use

**Ingress Controller**

It is the entry point to the cluster. It evaluates all the rules defined in the cluster and it manages all the redirections.

I have to consider the environment where the cluster is running. Load balancer (entrypoint) is managed in cloud deployments, but I have to take care of it in on-premise installations. So I need an entrypoint set up on physical or virtual server.

**Helm**

It is a package manager for k8s (like apt/yum/homebrew). It is a convenient way for packaging collections of k8s yaml files and distributing them in public and private repos.

**Helm Charts**

1. A collection of yaml files. I can create my own charts and push them to a helm repository (a private one), or I can download already deployed charts (I can reuse the existing configuration).
2. A templating engine. When I have deployment files that are similar, I can create a template file (a common blueprint and use placeholders for dynamic values).
   1. Values is an object that is defined in a yaml file (or via terminal)

It is useful when I deploy the same applications across different environments. I can create a chart that has all the necessary yaml files that it needs and then use it to redeploy the same application in different environments.

**Helm Chart Structure**

Top level <mychart> folder

1. Chart.yaml 🡪 meta information about the chart
2. Values.yaml 🡪 values for the template files (default values) 🡪 I can override it
3. Charts/ 🡪 chart dependencies (if it depends on other charts)
4. Templates/ 🡪 the actual template files

I run it with: **helm install <chartname>**

**Helm Release Management**

In Helm version 2, whenever I deploy a helm chart, helm client will send the yaml files to **Tiller** that runs in a k8s cluster. Tiller executes this request and creates components from yaml files. Whenever I create or change a deployment, Tiller will store a copy of each configuration the client sent for future reference (creating a history of chart executions). When I run **helm upgrade <chartname>**, the changes are applied to existing deployment instead of creating a new one. If upgrade goes wrong, I can rollback the upgrade.

Tiller has too much permissions and makes it a security issue, so it was removed in Helm version 3.

**Data Persistence**

1. I need a storage that doesn’t depend on the pod lifecycle.
2. Storage must be available on all nodes.
3. Storage needs to survive even if cluster crashes.

**Persistent Volume**

A cluster resource that is used to store data. It gets created using k8s yaml file, where we specify the kind, capacity etc. This takes the storage from the actual physical storage (local disk/nfs server/cloud storage). K8s does not interfere with the storage. Storage is an **external** **plugin** to the cluster. Persistent volumes live outside of any namespaces (they are available for all namespaces).

Local Volume Type 🡪 it is tied to each node. It does not survive cluster crashes.

Remote Storage 🡪 for DB persistence use remote storage.

**Persistent Volume Claim**

It claims a volume and whatever persistent volume satisfies this claim, will be used. It is used inside the Pod yaml file. It Is the way that the pod finds and accesses the persistent volume it needs. Claims must exist in the namespace that the pod exists (while persistent volumes live outside of it). When it finds the pv, the volume in mounted into the pod and then into the container.

I just create the pvc and assume that the cluster already provides this pv.

**Storage Class**

It creates and provisions persistent volumes dynamically, whenever pvc claims it. StorageClass is created using yaml configuration file. It is requested by a pvc.

**StatefulSet**

It is a k8s component used for specifically stateful applications (DB, or any application that stores data).

Stateless applications don’t keep record of state and each request is completely new.

Stateless applications are deployed using **Deployment** (allows to replicate the application)

Stateful applications are deployed using **StatefulSet** component (this also allows to replicate the stateful app parts). Scaling/replicating stateful applications is more difficult.

Stateful pod replicas can’t be created/deleted at the same time and can’t be randomly addressed. The replica pods are not identical (they have a **pod additional identity**).

**Pod Identity**

1. It creates a sticky identity for each pod (db-0, db-1, db-2) 🡪 pod identifiers
2. It is created from the same spec file but it is not interchangeable
3. It has a persistent identifier across any re-scheduling (if db-2 dies, it gets replaced with the same identity, db2)
4. Each statefulSet pod has an individual dns name (different from deployment pods)

Pod identity is important to avoid data inconsistencies. Only one pod is allowed to insert/update the data that is shared (master), while others can read from db (slaves/workers).

These pods don’t use the same physical storage of the data (while the data is still the same!). They each have their own replicas of the storage that each one can access. Each pod replica must have the same data at all times (continuously synchronize the data).

**K8s Services**

1. ClusterIP (default, accessible within the cluster) 🡪endpoints that communicate with containers. No external traffic
2. Headless 🡪 when I want to communicate to a specific pod with dns lookup (I set clusterIP: None which returns pod IP address instead of service IP address)
3. NodePort 🡪 accessible on a static port on each worker node in the cluster (instead of ingress). When created, a ClusterIP is also created internally. It is **not** very **efficient** and not **secure** (only for testing).
4. LoadBalancer 🡪 a service becomes accessible externally through a cloud provider’s LoadBalancer. When created, a NodePort and ClusterIP Service are created automatically from k8s. The entry point becomes the LoadBalancer 🡪 NodePort 🡪 ClusterIP.