Containers Orchestration

Lisa Zangrando
(INFN PD)

Outline

- Introduction of the problem
- Containers Orchestration
  - Overview of the major solutions
- Wrap-up

Introduction

- We explored how containers help us to easily create applications that are – as the name says – self-contained.
- We discussed docker applications and explored a bit the docker-compose
- What we need then? we'd explore how to effectively orchestrate many containers across distributed hosts

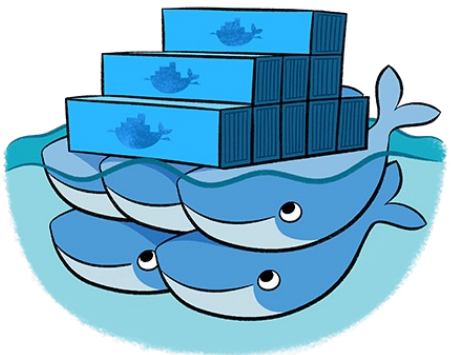
Orchestration

- Orchestration refers to the automated arrangement, coordination, and management of complex software systems, ensuring that all components work together smoothly.

- Key Functions:
  - **Automated Deployment**: ensures that containerized applications are deployed consistently across multiple environments.
  - **Scaling**: automatically adjusts the number of running application instances based on resource demand or defined policies.
  - **Load Balancing**: distributes network traffic efficiently across multiple tasks to ensure high availability and performance.
  - **Self-Healing**: monitors and replaces failed instances, ensuring that applications continue running as expected.
  - **Service Discovery**: helps identify and connect services within the cluster without manual intervention.

- Why it matters:
  - simplifies management of distributed applications.
  - enhances reliability, scalability, and fault tolerance in complex systems.
  - reduces manual intervention, allowing for easier application lifecycle management.

Three major solutions



Docker swarm

Apache Mesos

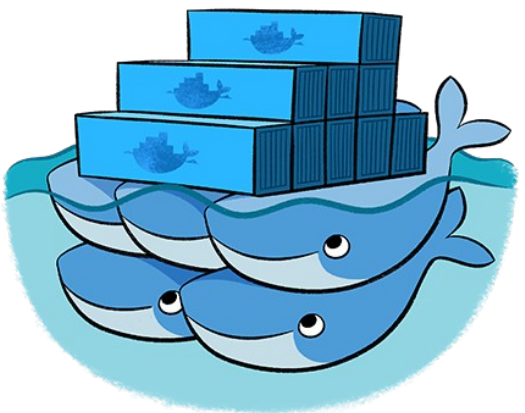
Nearly EOL by now

Kubernetes

Docker swarm

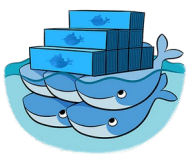
- Docker Swarm is Docker’s native clustering and orchestration tool for managing a cluster of Docker nodes.

- It is straightforward to install, lightweight and easy to use
  - It is embedded in Docker Engine
  - It does not require configuration changes if your system is already running inside Docker
  - it works seamlessly with existing Docker tools such as Docker Compose.



Docker swarm: key features

- **Decentralized design:** this means that any node in a Docker Swarm can assume any role at runtime.
- **Clustering**: groups multiple Docker engines into a single, virtual Docker engine.
- **High Availability:** fault tolerance through manager nodes and replication of services
- **Service Discovery**: automatically assigns a DNS name to services and handles service discovery within the cluster.
- **Load Balancing**: distributes network traffic across containers and nodes to ensure efficient resource utilization.
- **Scaling**: allows easy scaling of services up or down by adding or removing container instances.
- **Rolling Updates**: facilitates updating services without downtime through rolling updates and rollbacks.



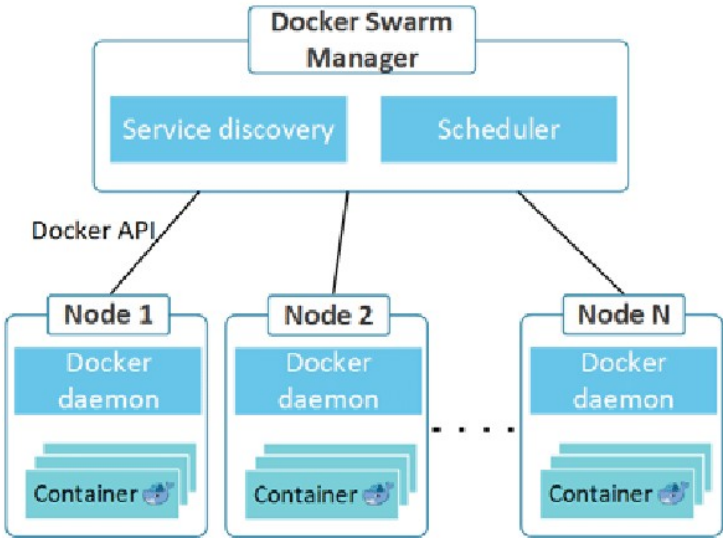
Docker swarm: architecture

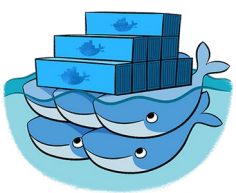
Manager Nodes: Coordinate the cluster and make scheduling decisions. They can also handle requests from users.

Worker Nodes: Execute tasks based on the manager's instructions. They do not make scheduling decisions.

Overlay Network: A network layer that allows containers across different hosts to communicate securely.

A given Docker host can be a manager, a worker, or perform both roles



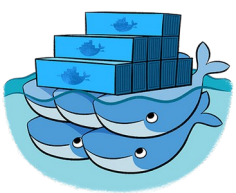


Docker swarm: tutorial

https://docs.docker.com/engine/swarm/swarm-tutorial/

The tutorial guides you through:

- Initializing a cluster of Docker Engines in swarm mode
- Adding nodes to the swarm
- Deploying application services to the swarm
- Managing the swarm once you have everything running



Kubernetes

- Kubernetes is an open-source platform for automating the deployment, scaling, and operation of containerized applications.
- **Developed by Google**: Kubernetes originated as an internal project at Google called Borg, which Google used to manage its own infrastructure.
- **Open-Source**: In 2014, Google open-sourced Kubernetes, combining its years of experience managing containers with contributions from the wider community.
- **Maintained by CNCF**: The Cloud Native Computing Foundation (CNCF) now maintains Kubernetes, supporting its growth and development in the cloud-native ecosystem.

Why Kubernetes?

- **Scalability & Reliability**
  - seamlessly scales applications across thousands of servers.
  - ensures high availability and automatic load balancing.
- **Automation of operations**
  - automatically deploys, scales, and repairs applications.
  - handles tasks like rolling updates, resource management, and self-healing.
- **Portability across environments**
  - supports multi-cloud and hybrid cloud deployments.
  - runs uniformly on public clouds (AWS, Azure, GCP) or on-premises.

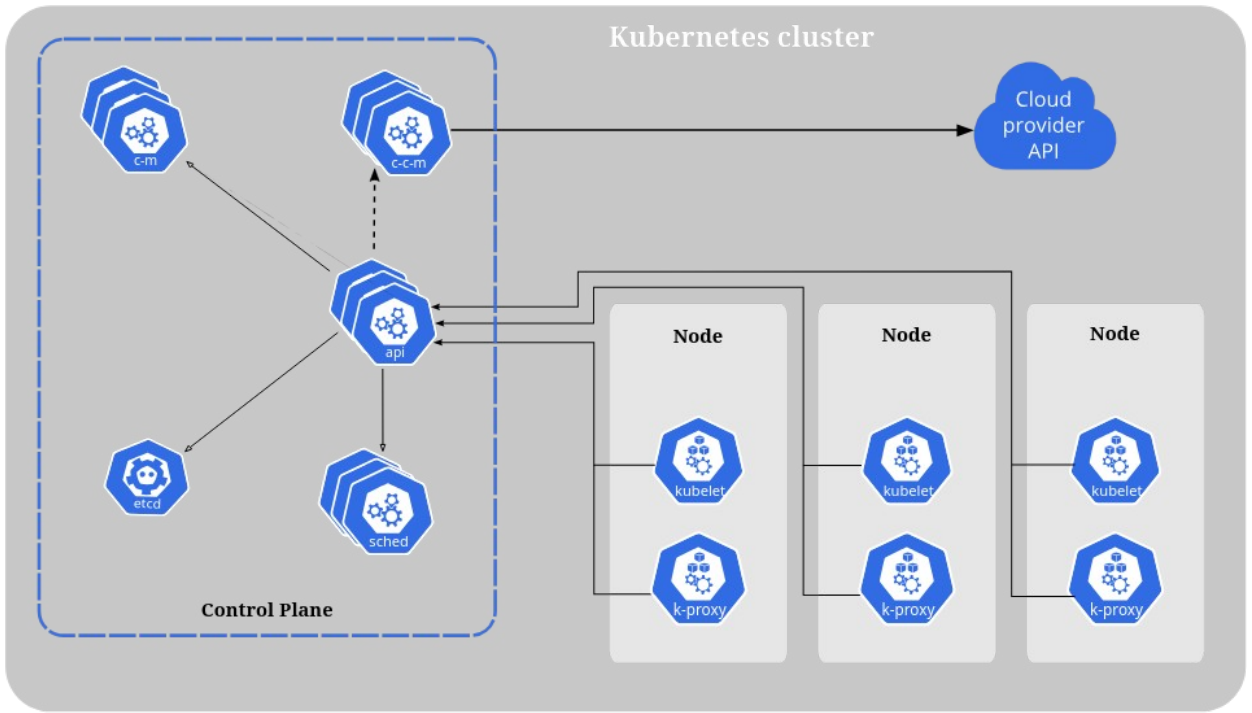
Why Kubernetes? (cont.)

- **Cost efficiency**
  - optimizes resource usage (CPU, memory) and reduces overhead.
  - deploys multiple apps on fewer resources, reducing costs.
- **Vast ecosystem & Community support**
  - backed by a large, active community and leading tech companies.
  - integrates with tools for monitoring, automation, security, and storage.
- **Continuous Innovation**
  - new features introduced regularly, driving efficiency and security.
  - enables businesses to innovate without compromising service stability.

Kubernetes: key features

- **Automated Rollouts & Rollbacks**: safely deploy and update applications with zero downtime.
- **Self-Healing**: automatically restarts failed containers, replaces and reschedules Pods.
- **Horizontal Scaling**: automatically scales applications based on resource usage or custom metrics.
- **Load Balancing**: distributes incoming traffic evenly across Pods for high availability.
- **Storage Orchestration**: manages local or external storage volumes (like AWS EBS or NFS)
- **Automatic Bin Packing**: automatically places containers based on their resource requirements and other constraints
- **IPv4/IPv6 Dual-Stack**:allocation of IPv4 and IPv6 addresses to Pods and Services
- **Batch execution:** Kubernetes can manage your batch and CI workloads, replacing containers that fail, if desired.
- **Designed for extensibility:** add features to your Kubernetes cluster without changing upstream source code.

Kubernetes architecture



Kubernetes architecture

- Control Plane (master node): manages the entire cluster and coordinates its activities.
  - **API Server**: front end of the control plane; exposes the Kubernetes API
  - **Scheduler**: determines which nodes run new Pods based on resource requirements and policies.
  - **Controller Manager**: ensures the cluster is running the desired state by managing controllers like the ReplicaSet, Deployment, etc.
  - **Etcd** (key-value store): stores all cluster data, including configuration and status information, as a key-value store.

- Data Plane (worker nodes): run the containerized applications (Pods).
  - **Kubelet**: ensures that containers are running in Pods by interacting with the container runtime
  - **Container Runtime**: software that runs and manages containers (e.g., Docker, containerd).
  - **Kube-proxy**: manages network communication between Pods and services, handling routing and load balancing.

- NOTE: In Kubernetes, a node is the operational unit that runs your workloads. It may be a **virtual** or **physical machine**, depending on your cluster's configuration. Each node is supervised by the control plane and contains essential services for running pods, the smallest deployable units in Kubernetes.

Kubernetes objects and reconciliation

- **Abstractions in Kubernetes**: Kubernetes operates using a set of abstractions that define the state of the system

- **Kubernetes Objects:**
  - definition: persistent entities representing the desired state of the system.
  - purpose: manage resources declaratively (what should happen, not how).
  - Components:
    - **specification** (desired state)
    - **status** (current state)

Kubernetes objects and reconciliation (cont.)

- **Why objects matter?**
  - simplifies management of complex resources.
  - allows focus on defining desired outcomes.
  - automates tasks like scaling, configuration, and resource allocation.
- **Reconciliation process:**
  - ensures the current state matches the desired state.
  - controllers monitor objects, detect discrepancies, and fix issues.
  - continuous process: auto-corrects application failures, ensures resilience.
- **Declarative & Resilient system:**
  - define "what" should happen; Kubernetes ensures it.
  - self-healing and scalable infrastructure without manual intervention.

Kubernetes objects

```
$ kubectl api-resources
NAME                              SHORTNAMES   APIVERSION                        NAMESPACED   KIND
bindings                                       v1                                true         Binding
componentstatuses                 cs           v1                                false        ComponentStatus
configmaps                        cm           v1                                true         ConfigMap
endpoints                         ep           v1                                true         Endpoints
events                            ev           v1                                true         Event
limitranges                       limits       v1                                true         LimitRange
namespaces                        ns           v1                                false        Namespace
nodes                             no           v1                                false        Node
persistentvolumeclaims            pvc          v1                                true         PersistentVolumeClaim
persistentvolumes                 pv           v1                                false        PersistentVolume
pods                              po           v1                                true         Pod
podtemplates                                   v1                                true         PodTemplate
replicationcontrollers            rc           v1                                true         ReplicationController
resourcequotas                    quota        v1                                true         ResourceQuota
secrets                                        v1                                true         Secret
serviceaccounts                   sa           v1                                true         ServiceAccount
services                          svc          v1                                true         Service
mutatingwebhookconfigurations                  admissionregistration.k8s.io/v1   false        MutatingWebhookConfiguration
validatingadmissionpolicies                    admissionregistration.k8s.io/v1   false        ValidatingAdmissionPolicy
validatingadmissionpolicybindings              admissionregistration.k8s.io/v1   false        ValidatingAdmissionPolicyBinding
validatingwebhookconfigurations                admissionregistration.k8s.io/v1   false        ValidatingWebhookConfiguration
customresourcedefinitions         crd,crds     apiextensions.k8s.io/v1           false        CustomResourceDefinition
apiservices                                    apiregistration.k8s.io/v1         false        APIService
controllerrevisions                            apps/v1                           true         ControllerRevision
daemonsets                        ds           apps/v1                           true         DaemonSet
deployments                       deploy       apps/v1                           true         Deployment
replicasets                       rs           apps/v1                           true         ReplicaSet
statefulsets                      sts          apps/v1                           true         StatefulSet
selfsubjectreviews                             authentication.k8s.io/v1          false        SelfSubjectReview
tokenreviews                                   authentication.k8s.io/v1          false        TokenReview
```

Kubernetes objects (cont.)

```
localsubjectaccessreviews                       authorization.k8s.io/v1           true         LocalSubjectAccessReview
selfsubjectaccessreviews                        authorization.k8s.io/v1           false        SelfSubjectAccessReview
selfsubjectrulesreviews                         authorization.k8s.io/v1           false        SelfSubjectRulesReview
subjectaccessreviews                            authorization.k8s.io/v1           false        SubjectAccessReview
horizontalpodautoscalers          hpa           autoscaling/v2                    true         HorizontalPodAutoscaler
cronjobs                          cj            batch/v1                          true         CronJob
jobs                                            batch/v1                          true         Job
certificatesigningrequests        csr           certificates.k8s.io/v1            false        CertificateSigningRequest
leases                                          coordination.k8s.io/v1            true         Lease
endpointslices                                  discovery.k8s.io/v1               true         EndpointSlice
events                            ev            events.k8s.io/v1                  true         Event
flowschemas                                     flowcontrol.apiserver.k8s.io/v1   false        FlowSchema
prioritylevelconfigurations                     flowcontrol.apiserver.k8s.io/v1   false        PriorityLevelConfiguration
ingressclasses                                  networking.k8s.io/v1              false        IngressClass
ingresses                         ing           networking.k8s.io/v1              true         Ingress
networkpolicies                   netpol        networking.k8s.io/v1              true         NetworkPolicy
runtimeclasses                                  node.k8s.io/v1                    false        RuntimeClass
poddisruptionbudgets              pdb           policy/v1                         true         PodDisruptionBudget
clusterrolebindings                             rbac.authorization.k8s.io/v1      false        ClusterRoleBinding
clusterroles                                    rbac.authorization.k8s.io/v1      false        ClusterRole
rolebindings                                    rbac.authorization.k8s.io/v1      true         RoleBinding
roles                                           rbac.authorization.k8s.io/v1      true         Role
priorityclasses                   pc            scheduling.k8s.io/v1              false        PriorityClass
csidrivers                                      storage.k8s.io/v1                 false        CSIDriver
csinodes                                        storage.k8s.io/v1                 false        CSINode
csistoragecapacities                            storage.k8s.io/v1                 true         CSIStorageCapacity
storageclasses                    sc            storage.k8s.io/v1                 false        StorageClass
volumeattachments                               storage.k8s.io/v1                 false        VolumeAttachment
```

Kubernetes objects: examples

Today we will focus on:

- Pod
- Deployment
- Service
- Volume
- Namespaces

Pod: the basic deployable unit in Kubernetes

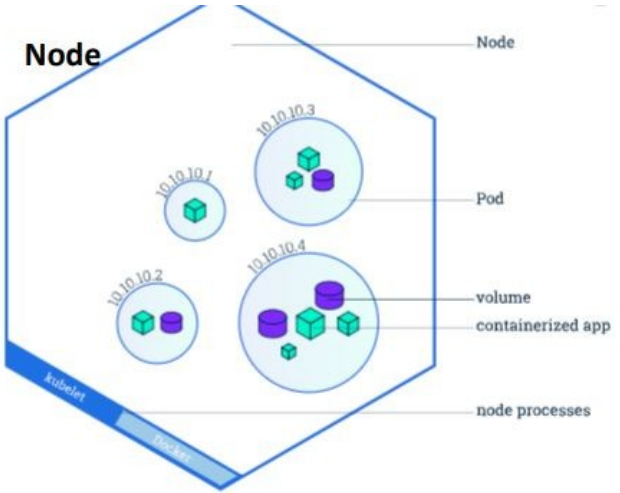
A Pod is the smallest and most basic deployable object in Kubernetes.

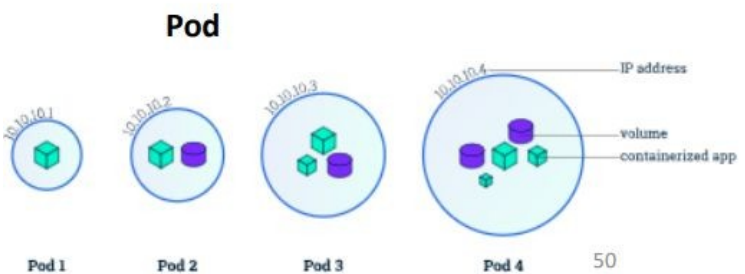
It represents a single instance of a running process in your cluster.

Pods can contain one or more containers, which are tightly coupled and share the same network and storage resources.

In most cases, Pods are used to run a single container, but for applications that need multiple containers working together (e.g., a web server with a sidecar container for logging), they can be grouped into the same Pod.

Pods have an ephemeral nature: if they fail, they can be replaced with a new instance, but the Pod itself is never restarted.





Pod: examples

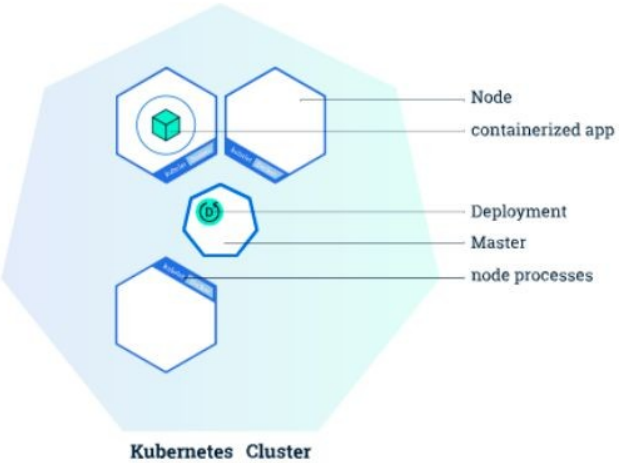
```
apiVersion: v1                # Specifies the version of the Kubernetes API used for the object
kind: Pod                     # Defines the type of Kubernetes object, in this case, a Pod
metadata:
  name: nginx                 # The name of the Pod, which must be unique within the namespace
  labels:
    app: nginx                # Labels used to categorize and organize Pods
spec:
  containers:                 # Defines the containers that will run within the Pod
  - name: nginx               # The name of the container, used within the Pod for identification
    image: nginx:latest       # Specifies the container image to use
    resources:
      requests:               # Minimum resources the container is guaranteed to have
        memory: "128Mi"       # Requests 128 MiB of memory
        cpu: "250m"           # Requests 250 milliCPU (0.25 of a CPU core)
      limits:                 # Maximum resources the container is allowed to use
        memory: "256Mi"       # Limits memory usage to 256 MiB
        cpu: "500m"           # Limits CPU usage to 500 milliCPU (0.5 of a CPU core)
    ports:
    - containerPort: 80       # Exposes port 80 inside the container,
```

Deployment: managing Applications and Scaling

A Deployment is a higher-level abstraction that manages a group of Pods and ensures that the right number of them are running at any given time.

It provides declarative updates, allowing you to define the desired state of your application (like how many replicas you need, or what version to run), and Kubernetes handles the rest.

Deployments are ideal for stateless applications where you want to scale up or down or roll out updates. If anything happens to the Pods (like failures), the Deployment controller automatically replaces them, ensuring the application remains available and healthy.



Deployment: example

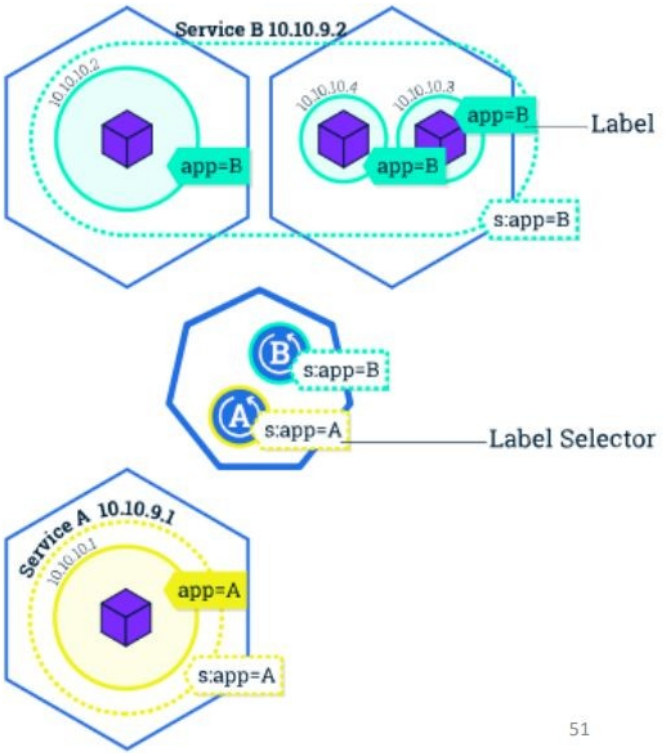
```
apiVersion: apps/v1                 # Specifies the version of the Kubernetes API for deployments
kind: Deployment                    # Defines the type of Kubernetes object: Deployment
metadata:
  name: nginx-deployment            # The name of the Deployment object
  labels:
    app: nginx                      # Labels used to categorize and identify the Deployment
spec:
  replicas: 3                       # Specifies the desired number of Pod replicas (3 Pods)
  selector:
    matchLabels:
      app: nginx                    # Defines a label selector
  template:
    metadata:
      labels:
        app: nginx                  # Labels for the Pods created by the Deployment
    spec:
      containers:
      - name: nginx                 # The name of the container running in each Pod
        image: nginx:latest         # Specifies the container image to use
        ports:
        - containerPort: 80         # Exposes port 80 within the container for HTTP traffic
```

Service: ensuring stable access to Pods

A Service in Kubernetes acts as a stable entry point for a group of Pods. Since Pods are ephemeral and can change IP addresses when they are recreated, a Service ensures that you have a consistent way to reach the Pods, whether from inside the cluster or externally.

Services distribute traffic to Pods using a built-in load balancer, and they can operate in various modes, such as **ClusterIP** (for internal communication), **NodePort** (for external access), or **LoadBalancer** (for cloud providers).

The service ensures that applications remain accessible even as Pods are dynamically created and destroyed.



Service: example

```
apiVersion: v1            # Specifies the version of the Kubernetes API
kind: Service             # Defines the type of Kubernetes object, in this case, a Service
metadata:
  name: nginx-service     # The name of the Service object
  labels:
    app: nginx            # Labels used to identify the Service
spec:
  type: NodePort          # The Service type; NodePort exposes the service on a static port
  selector:
    app: nginx            # Defines a label selector that matches Pods
  ports:
    - port: 80            # The port that the Service exposes inside the cluster
      targetPort: 80      # The port on the Pod that the Service will forward traffic to
      nodePort: 30007     # The static port on the node where the service will be exposed
                          # externally (in this case, port 30007 on each node in the cluster)
```

Volume: persistent storage for containers

A Volume in Kubernetes provides storage that can be attached to a Pod.

Unlike container storage, which is temporary and lost when a container stops, Volumes persist data beyond the lifecycle of a single container.

Kubernetes supports different types of volumes, including **hostPath** (using the node's local storage), **PersistentVolume** (abstracting storage from the infrastructure), and **cloud-provider-specific volumes** (e.g., AWS EBS, Google Cloud Disks).

Volumes are key for applications that require stable, long-term data storage.

Volume: example

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-volume
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
    volumeMounts:
    - mountPath: /usr/share/nginx/html   # Mounts a volume inside the container at this path
      name: hostpath-volume              # Refers to the volume named 'hostpath-volume'
  volumes:
  - name: hostpath-volume                # Defines a volume named 'hostpath-volume'
    hostPath:
      path: /mnt/data       # The path on the host machine that is mounted into the container
      type: Directory       # Specifies that the volume is a directory on the host filesystem
```

Namespace: organizing and isolating resources

Namespaces provide a way to divide a single Kubernetes cluster into multiple virtual clusters.

They are especially useful in environments with many users or teams, helping to isolate resources and avoid conflicts.

For example, different development teams can have their applications deployed in separate namespaces, ensuring that resources like Pods, Services, and volumes are organized and managed independently.

Namespaces also allow for resource quotas and access control, giving teams better governance over what they can use and access in the cluster.

Kubernetes-as-a-Service

Deploying and managing a Kubernetes cluster is generally not trivial (that's why Minikube was introduced), since it requires effort and several skills.

It would be nice to automatize this part as well, and focus just on deploying our containers on a Kubernetes cluster that somebody else instantiates for us.

Kubernetes-as-a-Service (KaaS) is a cloud-based solution that allows organizations to leverage the full power of Kubernetes without the complexities of managing it themselves. Instead of setting up, configuring, and maintaining your own Kubernetes clusters, you can use a managed service offered by cloud providers.

Popular KaaS providers:

- Google Kubernetes Engine (GKE)
- Amazon Elastic Kubernetes Service (EKS)
- Azure Kubernetes Service (AKS)
- **INFN Cloud**

Benefits of KaaS: it reduces operational overhead, ensures a high level of availability, and allows developers to focus on building applications rather than managing infrastructure.

Kubernetes vs Docker swarm

Kubernetes and Docker Swarm are designed to manage and scale containerized applications, they differ in architecture, complexity, and feature sets.

Kubernetes: best for complex, large-scale applications requiring advanced orchestration, scalability, and high availability.

Docker Swarm: ideal for simpler, smaller deployments, or teams looking for a fast, easy-to-learn solution.

Feature	Kubernetes	Docker Swarm
Ease of Setup	More complex setup; requires detailed configuration	Easier to set up and configure, less learning curve
Scaling	Advanced auto-scaling features based on resource metrics	Manual scaling; basic auto-scaling options
Load Balancing	Built-in load balancing across Pods and services	Built-in but simpler load balancing
Networking	Rich networking model with namespaces, service discovery, and overlay networks	Simpler networking, limited overlay networks
State Management	Ensures desired state with Controllers (Deployments, ReplicaSets)	Less robust state management; focuses on desired state only during the lifetime of services
High Availability	Advanced fault tolerance and high availability with automatic recovery	Basic fault tolerance, less sophisticated recovery mechanisms
Ecosystem & Extensibility	Extensive ecosystem, integrates with many tools (Prometheus, Helm, etc.)	Smaller ecosystem, mainly Docker-native tools
Rolling Updates & Rollbacks	Built-in support for rolling updates, rollbacks, and canary deployments	Basic rolling updates and rollbacks
Learning Curve	Steeper learning curve due to advanced features and complexity	Simpler, quicker to learn and adopt