



Review argomenti trattati

Docker, K8s, Open-ID

Antonio Sandroni

Workshop on
management of distributed resources for genomic communities

29-30 October 2024



POLICLINICO DI
SANT'ORSOLA



Ministero
dell'Università
e della Ricerca



Virtualization & Containers

Containers



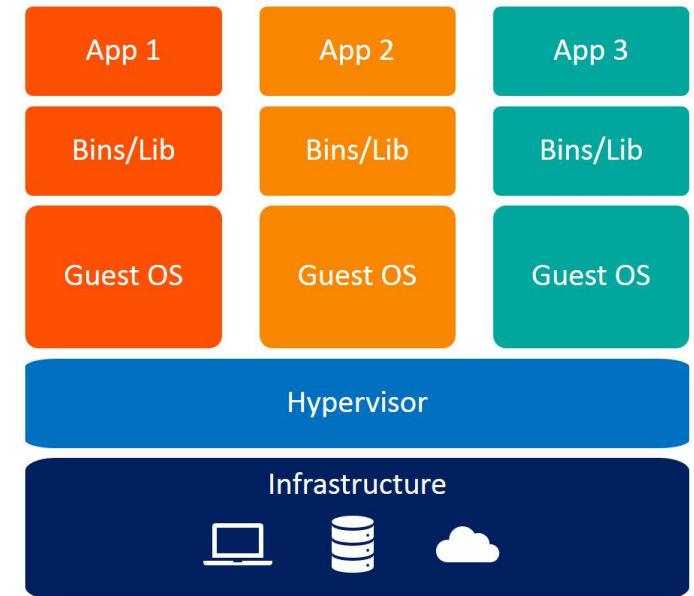
From Virtualization to Containers: why?

Virtualization: creation of a virtual version of "something", like an OS, a storage device, a network

In building a web service on Ubuntu, it works on our machine but not on the remote server.

Reasons: different OS, missing libraries, incompatible version of software.

Each virtualized application includes not only the application but also **an entire operating system**.



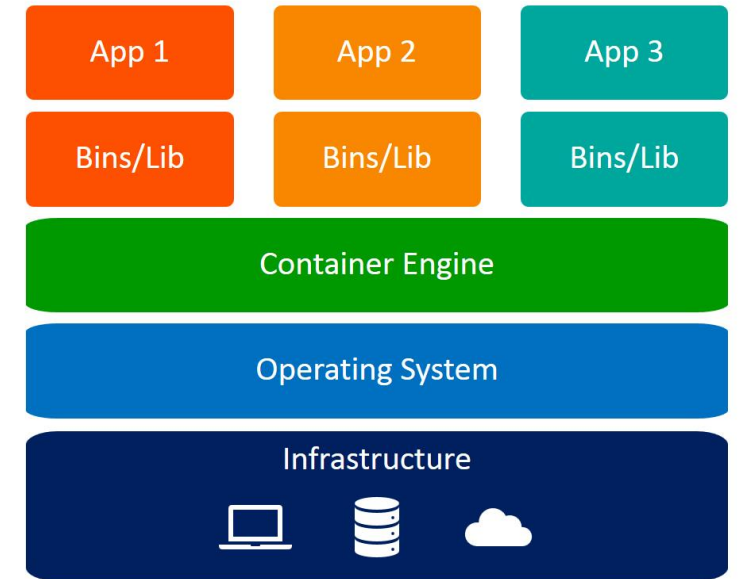
Virtual Machines

Containers

- Docker is a **platform** for running applications in lightweight units called containers.
- The Docker container engine comprises just the **application and its dependencies**. It runs an isolated process on the host operating system.
- It's much more portable, fast and efficient, a container is a lightweight VM.
- This provides enormous simplifications to software development and deployment processes



Workshop on management of distributed resources for
genomic communities



Containers

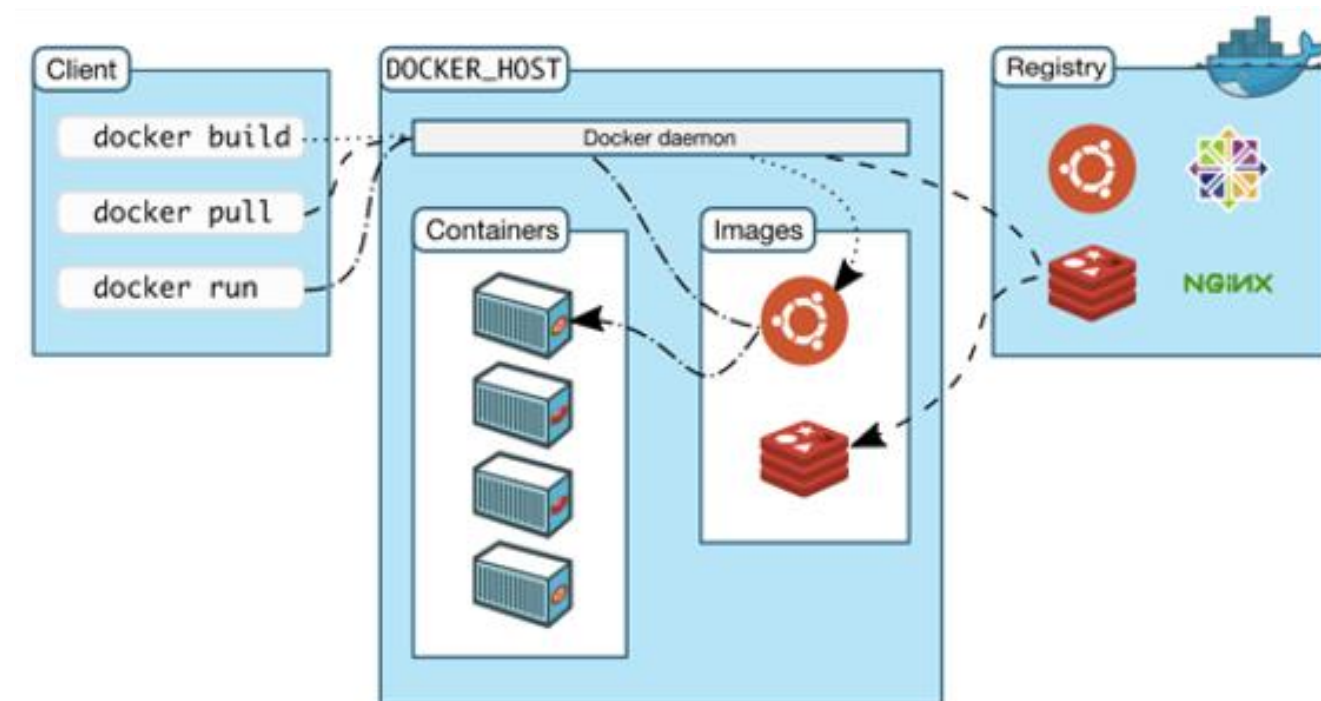


Docker

- Docker is an open-source platform that automates the development and deployment of applications inside portable and self-sufficient software “containers”.

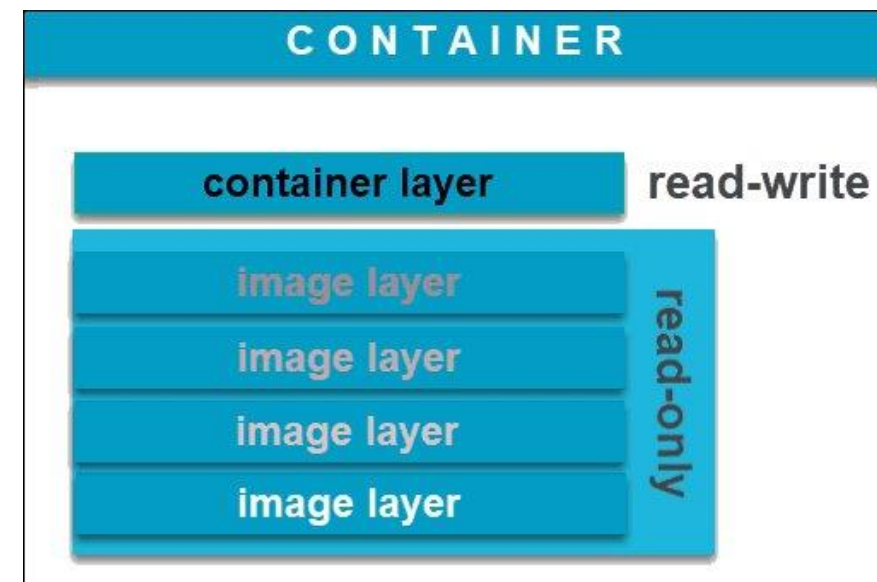
- Main Components:

- **Docker Engine:** the core technology behind Docker that enables building, running, and managing containers.
- **Docker Hub:** is a cloud-based registry that stores and distributes Docker images, allowing users to find, share, and collaborate on containerized applications

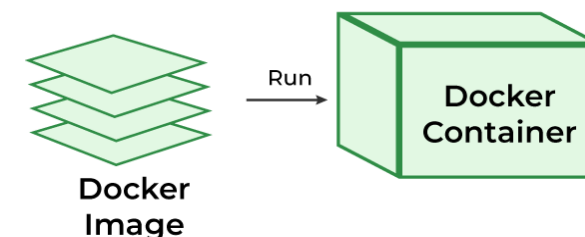


Containers & Images

- A **Docker image** is an unalterable, read-only file with all necessary software components—source code, dependencies, libraries, and tools—to ensure the application operates as intended across various environments.
- A **Container** is the active instance created when a Docker image is deployed, representing a running, isolated environment based on that image.



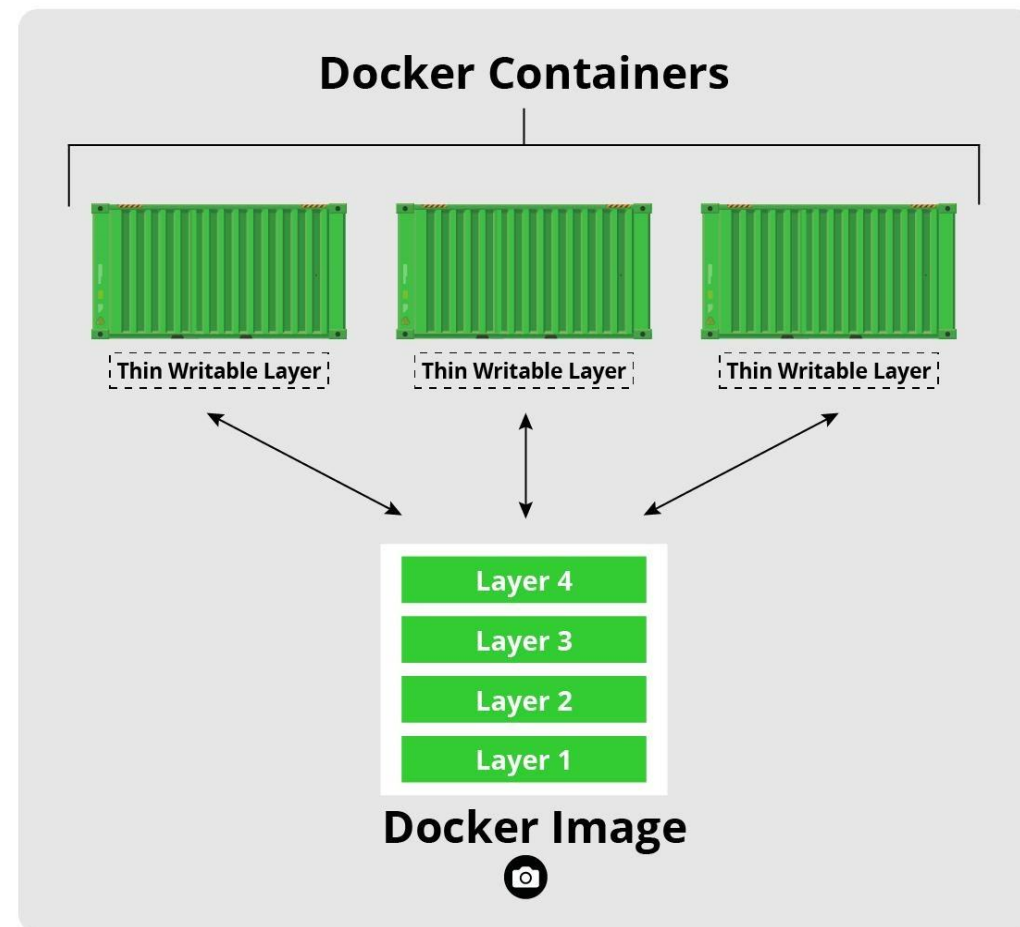
Images can exist without containers, whereas a container needs to run an image to exist



Images blueprint: ex uno plures

A Docker image typically outlines:

- The **base image** to use as a starting point for the container.
- The **commands** that should execute upon container startup.
- Instructions** for setting up the container's file system.
- Configuration details**, such as which ports to expose and how to handle data transfers between the container and the host system.



Docker Images

- Search for an image on Docker Hub:
 - `docker search ubuntu`
- Pull (download) an image
 - `docker pull ubuntu`
- List downloaded images:
 - `docker images`
- Run a container:
 - `docker run -it ubuntu /bin/bash`

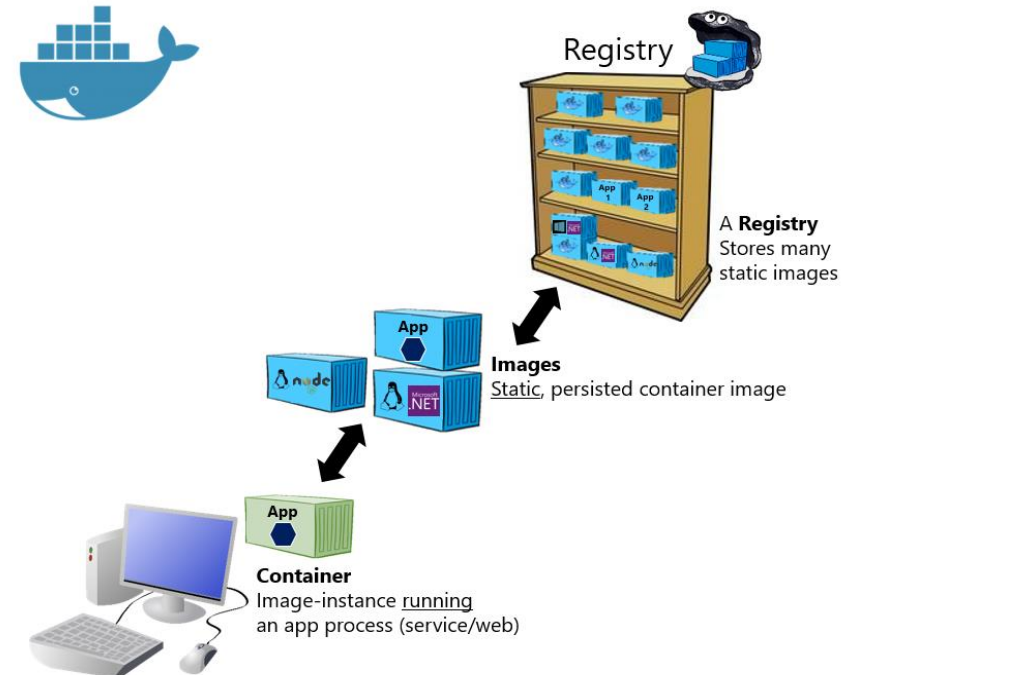
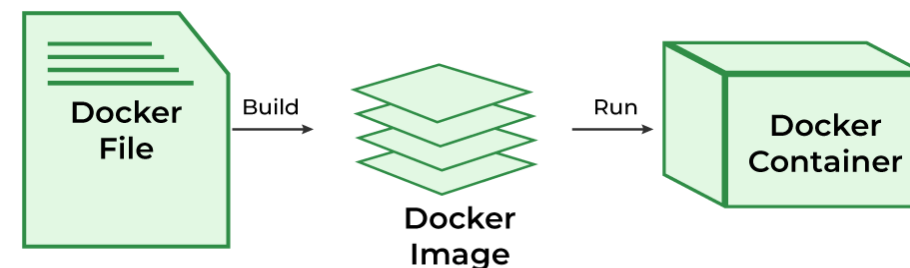


Image extension

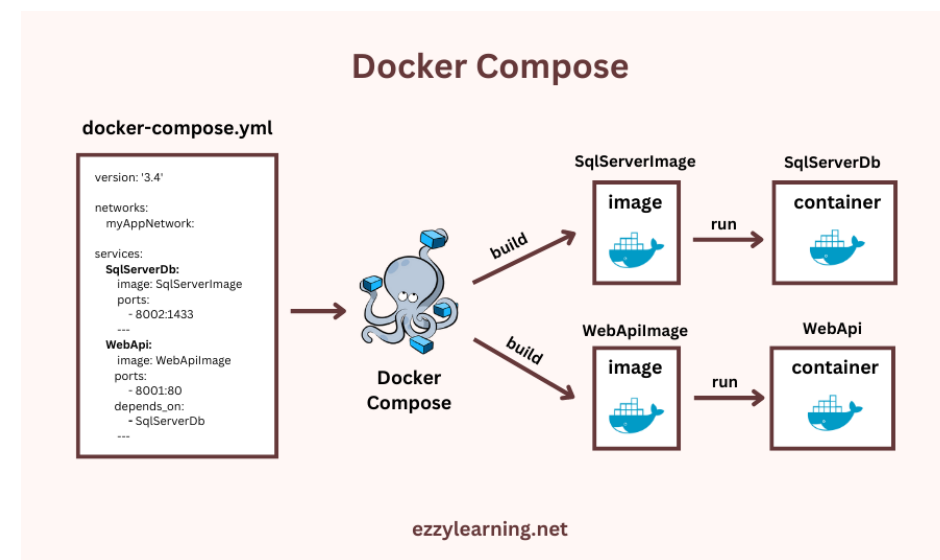
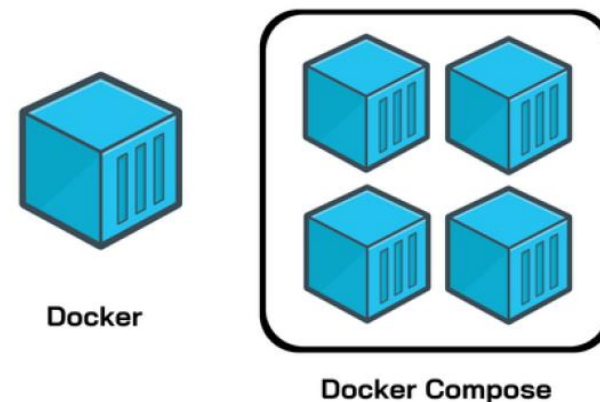
- If a Docker image lacks a command (e.g., ping), install it in the container:
 - `docker run ubuntu /bin/bash -c "apt update; apt -y install inetutils-ping"`
- However, re-running the container will lose the installation, as each docker run starts a new container from the original image.
- Instead of manually modifying a container, you can streamline the process by creating a **Dockerfile**, which contains all necessary commands to build an image.

```
FROM ubuntu
ENV DEBIAN_FRONTEND=noninteractive
RUN apt update
RUN apt install -y inetutils-ping
```



Docker compose

- The next step involves creating **application stacks**, which means linking multiple containers to provide a cohesive multi-container service, all hosted on a single virtual machine.
- This is achieved using the **docker-compose** command.
- **Docker Compose** is a tool for defining and running multi-container Docker applications, using a straightforward YAML syntax to describe the services, networks, and volumes needed for your application.



docker-compose.yml



This builds the container for WordPress,
with both the "backend" and "frontend" networks

Container image for WordPress
(from Docker Hub, latest)

Note that here we refer
to the other container

Port 8080 on the host (VM)
is mapped to port 80 on the
container

This builds the container for the database,
with only the "backend" network

Container image for MySQL
(from Docker Hub)

Configuration variables
for the container software

```
version: '3'
```

```
services:
```

```
  database:
```

```
    image: mysql:5.7
```

```
    environment:
```

- MYSQL_USER=wordpress
- MYSQL_PASSWORD=olss_passwd
- MYSQL_DATABASE=wordpress
- MYSQL_RANDOM_ROOT_PASSWORD=true

```
    networks:
```

- backend

"Obvious" note: although this is just for a demo,
do not use the passwords shown in this screen!

```
  wordpress:
```

```
    image: wordpress
```

```
    depends_on:
```

- database

```
    environment:
```

- WORDPRESS_DB_HOST=database
- WORDPRESS_DB_USER=wordpress
- WORDPRESS_DB_PASSWORD=olss_passwd
- WORDPRESS_DB_NAME=wordpress

```
    ports:
```

- 8080:80

```
    networks:
```

- backend
- frontend

```
networks:
```

```
  backend:
```

```
    driver: bridge
```

```
  frontend:
```

```
    driver: bridge
```

Best practices

- **Use shared base images:** Whenever possible, leverage common base images to minimize size.
- **Limit data in container layers:** Keep the amount of data written to container layers to a minimum.
- **Chain RUN statements:** Combine multiple commands in a single RUN statement to reduce the number of layers created.
- Each container should **host a single application**. For instance, avoid running both an application and its database in the same container.
- Place frequently changing layers **at the bottom of the Dockerfile**.
- Use **proper tagging** for your images to clearly indicate which version of the software they correspond to.
- Do not confuse RUN with CMD

Pros

- Lightweight:** Docker images are much smaller (e.g., 70MB for Ubuntu) compared to full OS distributions, which saves storage.
- Fast Startup:** Containers launch in seconds, far quicker than virtual machines, enhancing productivity.
- Efficient Resource Use:** Containers share the host OS kernel, allowing more efficient use of CPU and memory than VMs.
- Portability:** Docker ensures consistency across environments, making deployment reliable.
- Isolation:** Containers keep apps and dependencies separate, reducing conflicts and simplifying management.

Cons

- Applications that require direct **hardware access** or real-time performance may not perform as well in containers compared to bare metal or VMs.
- Security:** it's not as secure as full virtual machines. Containers share the OS kernel, so a vulnerability in the kernel could potentially affect all containers.
- Persistent Storage:** By default, containers are ephemeral.
- Overhead in Orchestration:** As applications scale and involve multiple services, managing Docker containers can become complex.

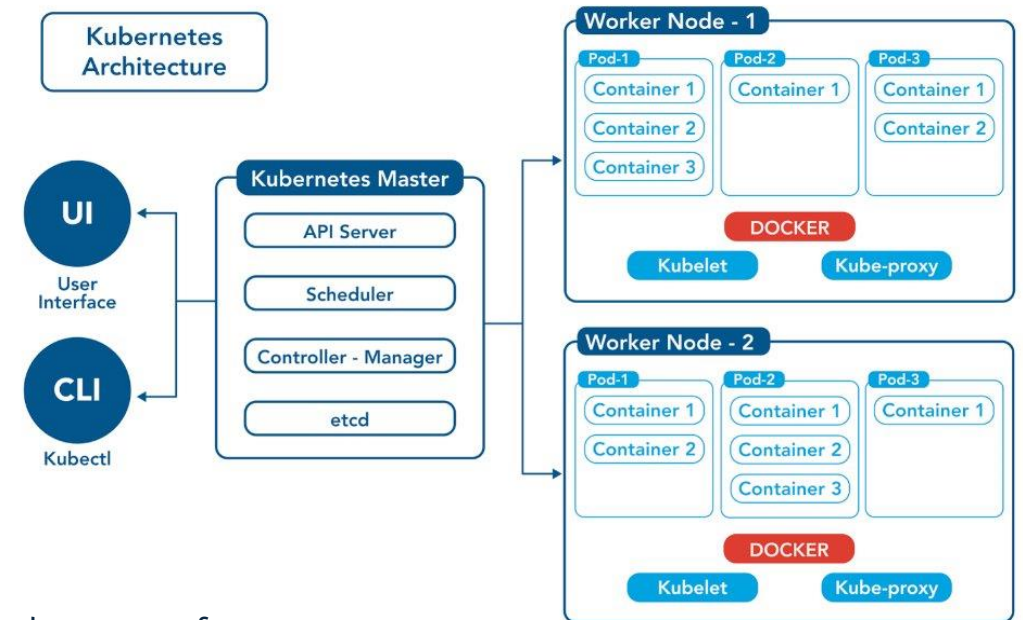
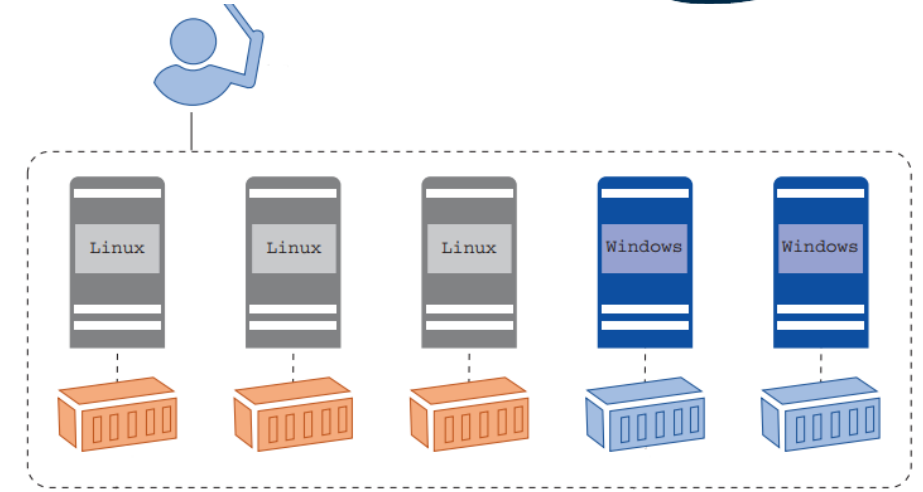
K8s

Kubernetes



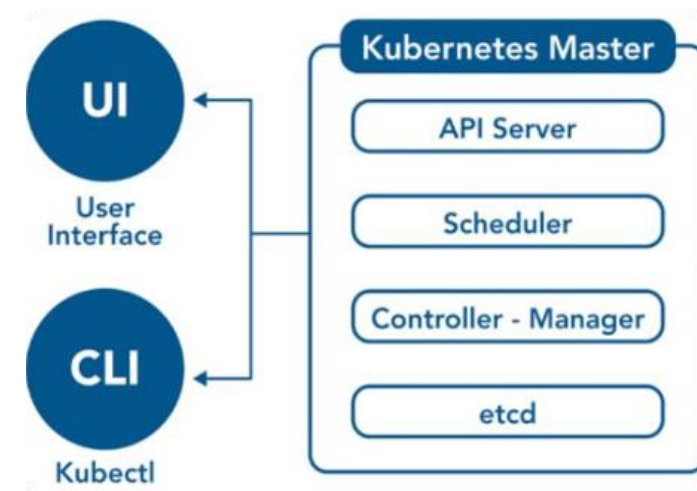
Kubernetes

- **Kubernetes** is a container orchestrator. It makes the changes to **reach the desired state**
- A cluster is a single logical unit composed of many server nodes.
- Each node has a **container runtime** installed, for example Docker.
- **ControlPlane**: is responsible for managing the overall state and behavior of the Kubernetes cluster.
- **Worker**: Worker Nodes are the machines (either physical or virtual) that run containerized applications in Kubernetes.



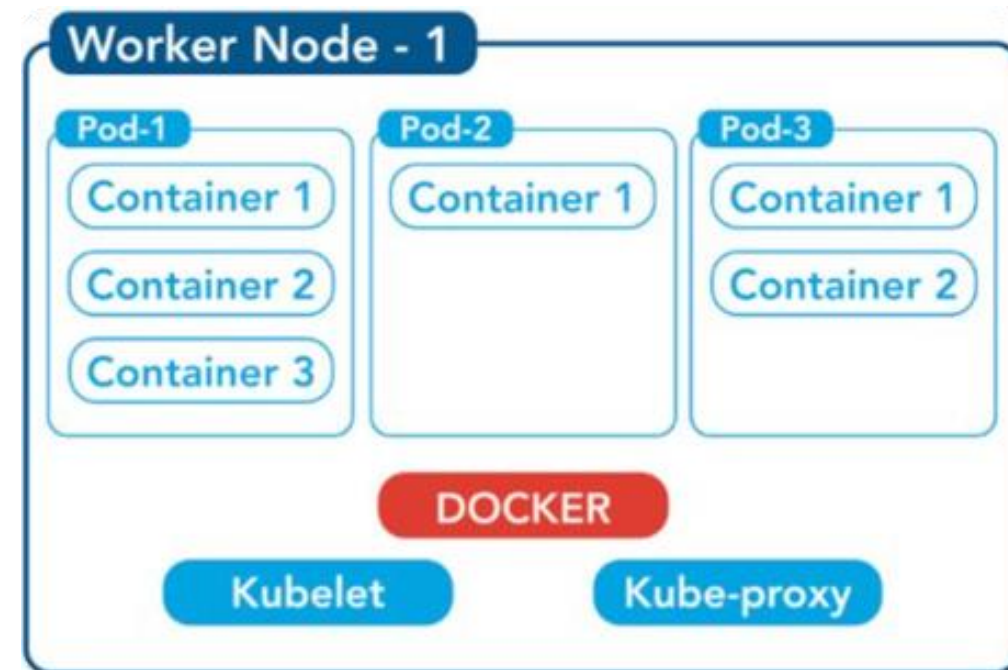
Control Plane

- **API Server:** The Kubernetes API server is the main gateway through which users, services, and components communicate with the cluster.
- **Etcd:** This is the distributed key-value store that keeps all cluster data, storing configuration details, status, and metadata.
- **Controller Manager:** This manages the controllers that monitor the state of the cluster and make adjustments as needed.
- **Scheduler:** The scheduler determines which nodes in the cluster will run each pod, based on resource requirements, workload priorities, and constraints. It ensures efficient resource allocation.



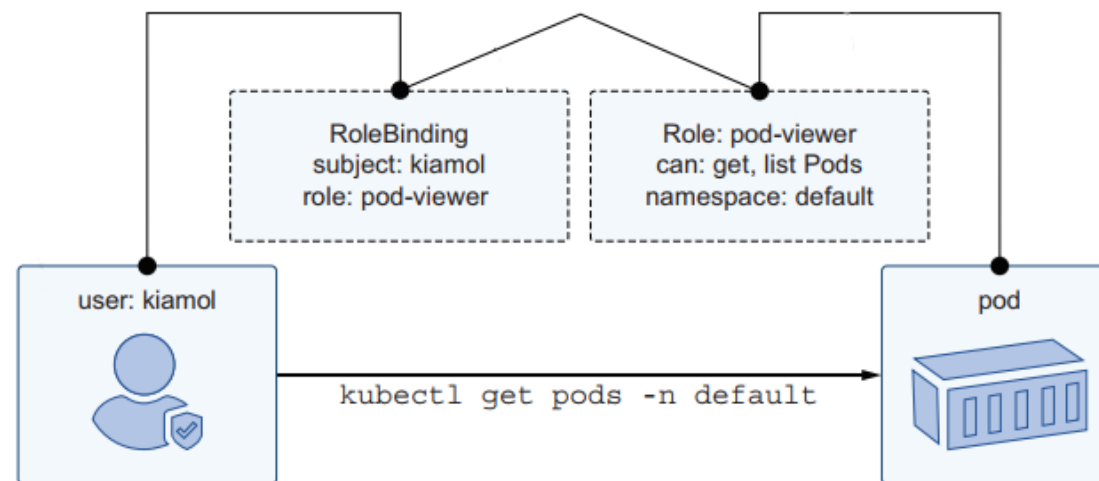
Worker Nodes

- **Kubelet:** This is the agent on each node that communicates with the API server. It receives and executes instructions, ensuring that containers are running as specified in the pod manifest.
- **Container Runtime:** This is the software responsible for running containers
- **Kube-proxy:** A network component that runs on each worker node, managing networking rules and facilitating communication between pods and services both within and outside the cluster.
- **Pods:** The smallest deployable units in Kubernetes. A pod can contain one or more containers



Rules and policies

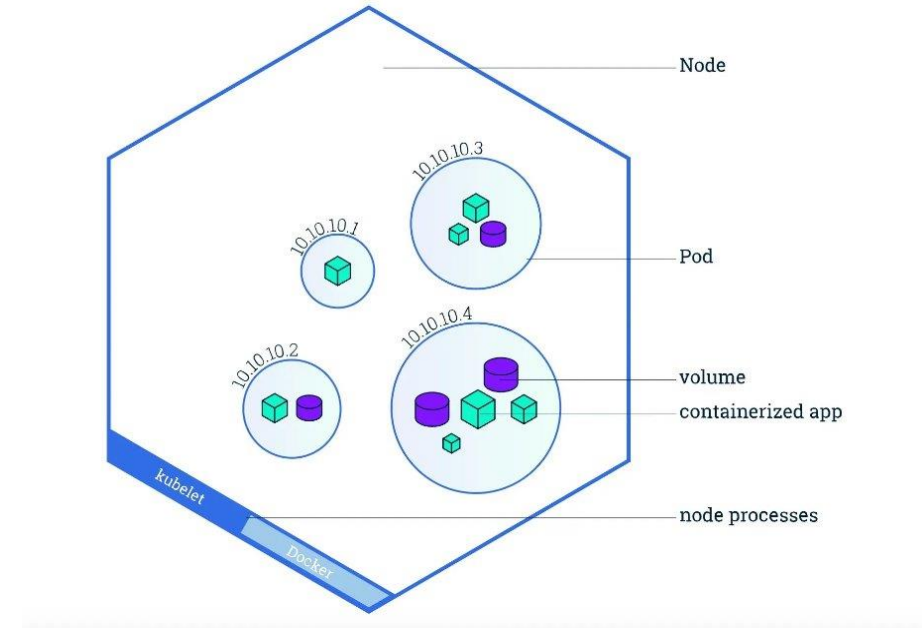
- **kubectl** is the command-line tool used to interact with a Kubernetes cluster
- **RBAC** (Role-Based Access Control) manages user access to resources through roles and bindings.
- A **Role** defines a set of permissions within a specific namespace (e.g., view or edit Pods).
- **RoleBinding** assigns a Role to a user or group within a namespace.



The Pod

- **Pod** is the smallest and most basic deployable unit.
- It represents a single instance of a running application or process within the Kubernetes cluster.
- It is **ephemeral**
- Containers inside a Pod share the **same IP address**
- Lives on one node

```
apiVersion: v1
kind: Pod
metadata:
  name: <nome_Pod>
spec:
  containers:
    - name: <nome_container>
      image: <immagine>
```



Pod specifications

- Each Pod belongs to **one namespace** and cannot exist in more than one. Namespaces are distinct environments, and they can't be nested.
- **Labels** are metadata assigned to Kubernetes objects, such as Pods, for identification and grouping. Labels can be used in selectors to match.
- **Resources** control the CPU and memory (RAM) allocations for Pods, defining the computing power a Pod can request and consume.
- **ConfigMaps** is an API object store non confidential data in key-value format, keeping configuration separate from container images.
- **Secrets** are similar to ConfigMaps but designed to store sensitive information, such as passwords, SSH keys, and certificates.

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx-pod
5    namespace: my-namespace
6    labels:
7      app: nginx
8      environment: production
9  spec:
10   containers:
11     - name: nginx
12       image: nginx:latest
13       resources:
14         requests:
15           memory: "256Mi"
16           cpu: "500m"
17         limits:
18           memory: "512Mi"
19           cpu: "1"
20       env:
21         - name: MY_CONFIG
22           valueFrom:
23             configMapKeyRef:
24               name: my-configmap
25               key: configKey
26         - name: MY_SECRET
27           valueFrom:
28             secretKeyRef:
29               name: my-secret
30               key: secretKey
31
32

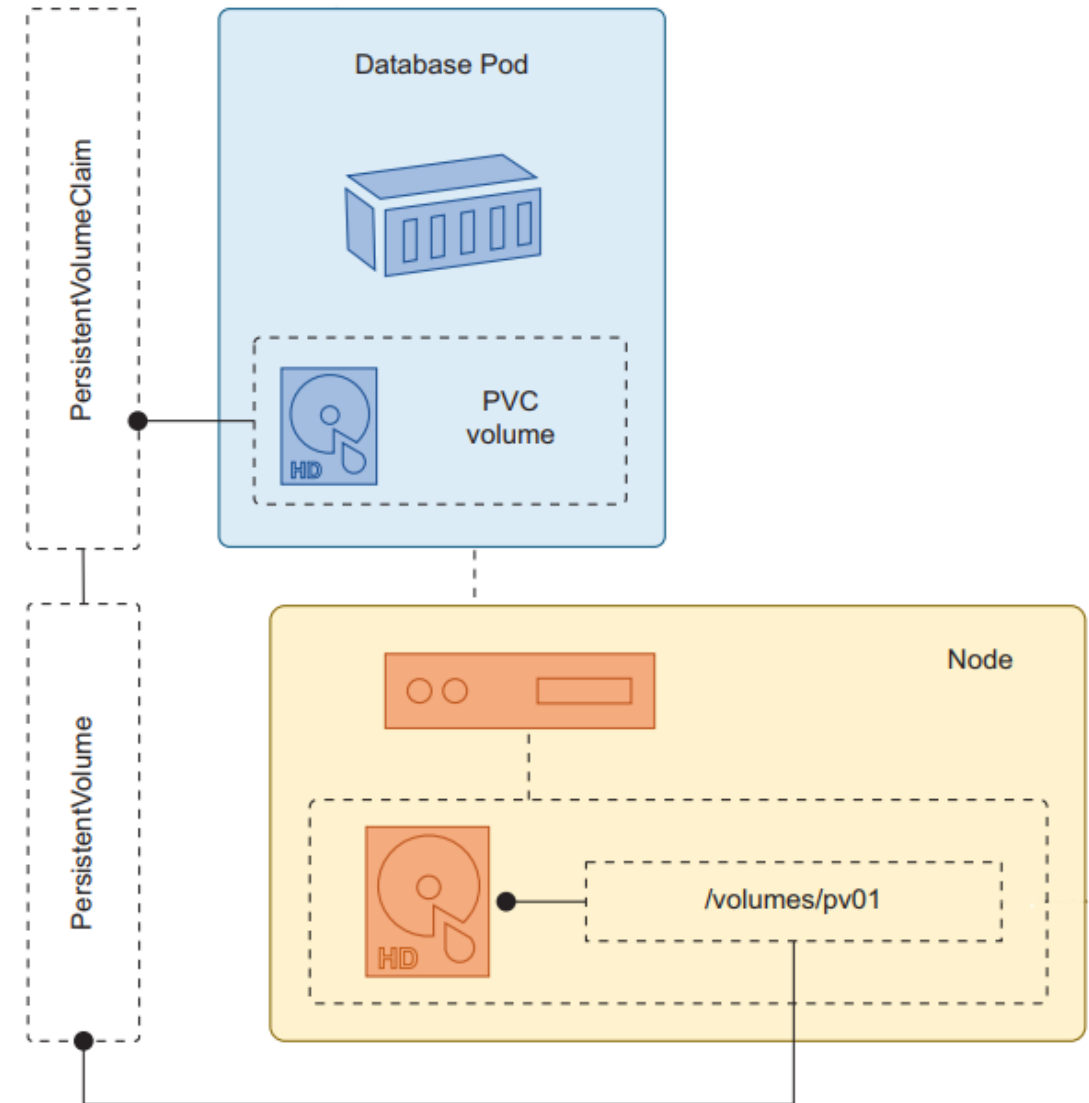
```


Volumes

- Pods are **ephemeral** and so is their filesystem, built by Kubernetes on multiple sources, from the container image to writable layer for the container.
- If a Pod is storing data on a node and suddenly crashes, if it is recreated on another node, how can he **reach his older data**?

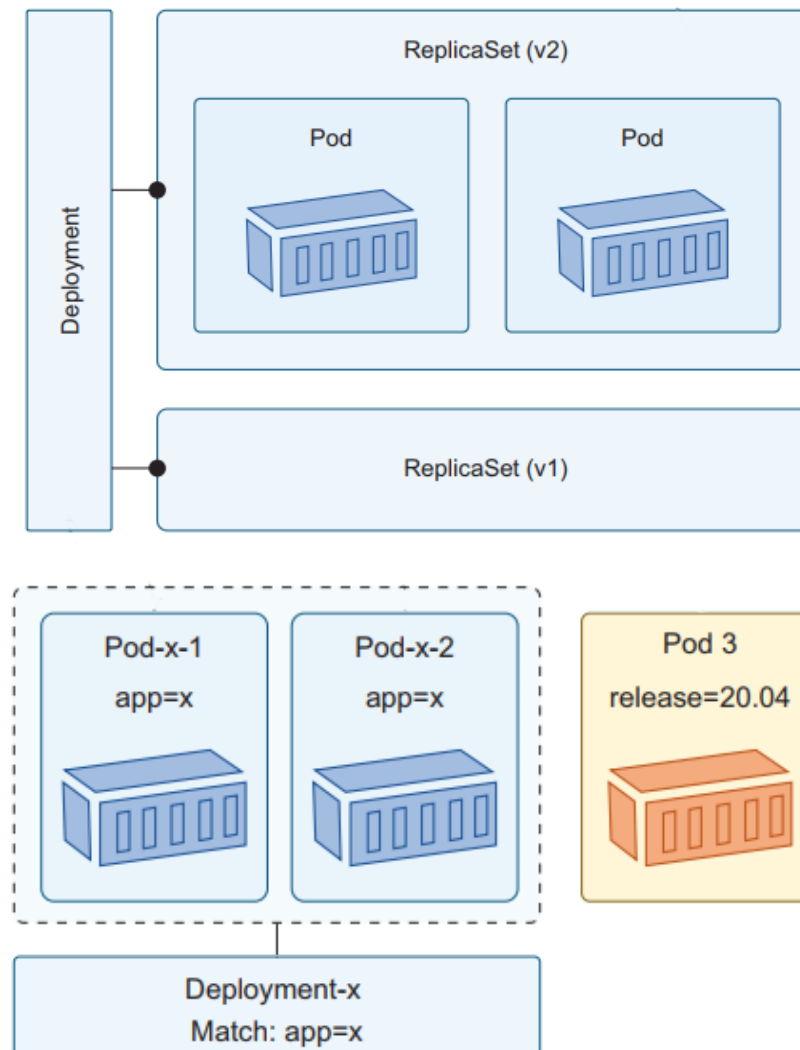
How to **extend the lifetime of data** beyond the life of a Pod?

- We can create PersistentVolumes (PV), an abstraction of piece of storage, and connect them to the Pod through a PersistentVolumeClaim (PVC).



Controllers: managing resources

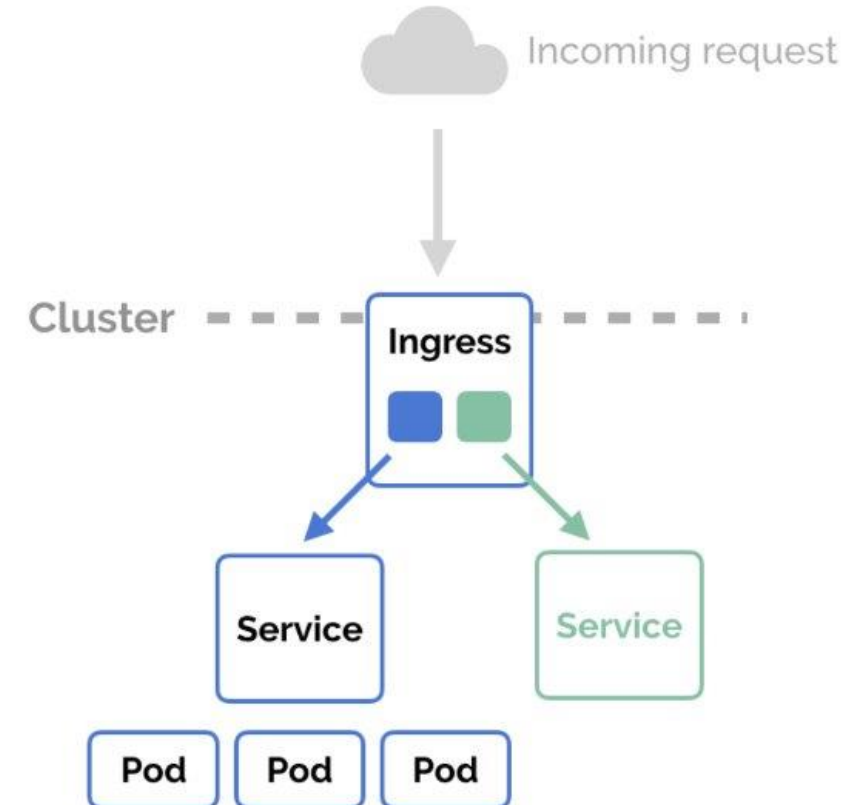
- Pods are too simple to be useful on their own, what happens if is lost?
- We can manage them with **controllers**, resources that manages other resources.
- To scale the app we use **Replicaset**: manages the number of Pods
- On top of ReplicaSets the **Deployment**.
 - Manages the creation of Pods and their number
 - Substitute the old Pods with new one (Rollout)



Pod Networking

Ingress

- Each Pod is assigned a **unique** IP address
 - What if a Pod dies and changes IP? How do I refer to it?
- **Services** are abstraction over network layer that facilitate communication between Pods.
- To handle the fact that Pods may have changing IP addresses, Kubernetes provides **Services**
- **Ingress** is a Kubernetes resource that manages external access to services within a cluster, routing traffic to different Services based on request URL





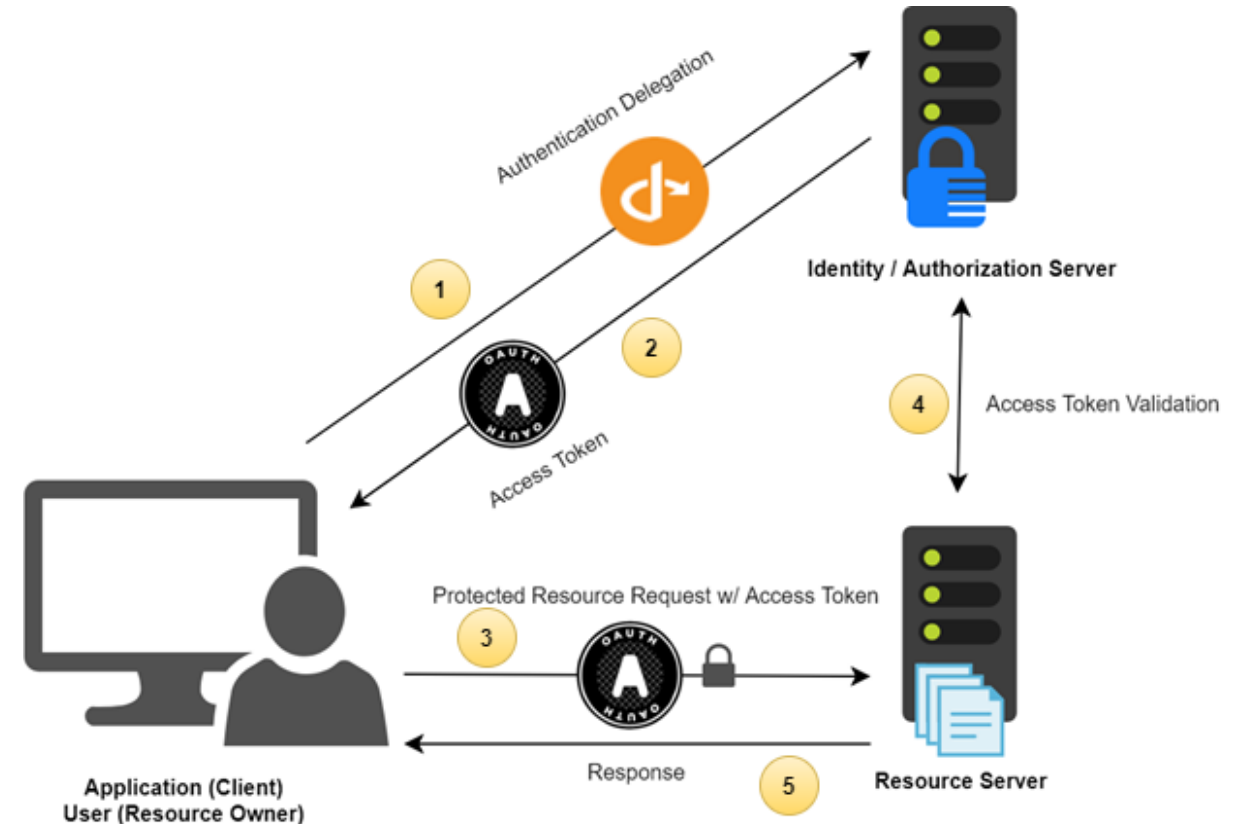
OIDC

An Authentication layer on
top of OAuth2.0



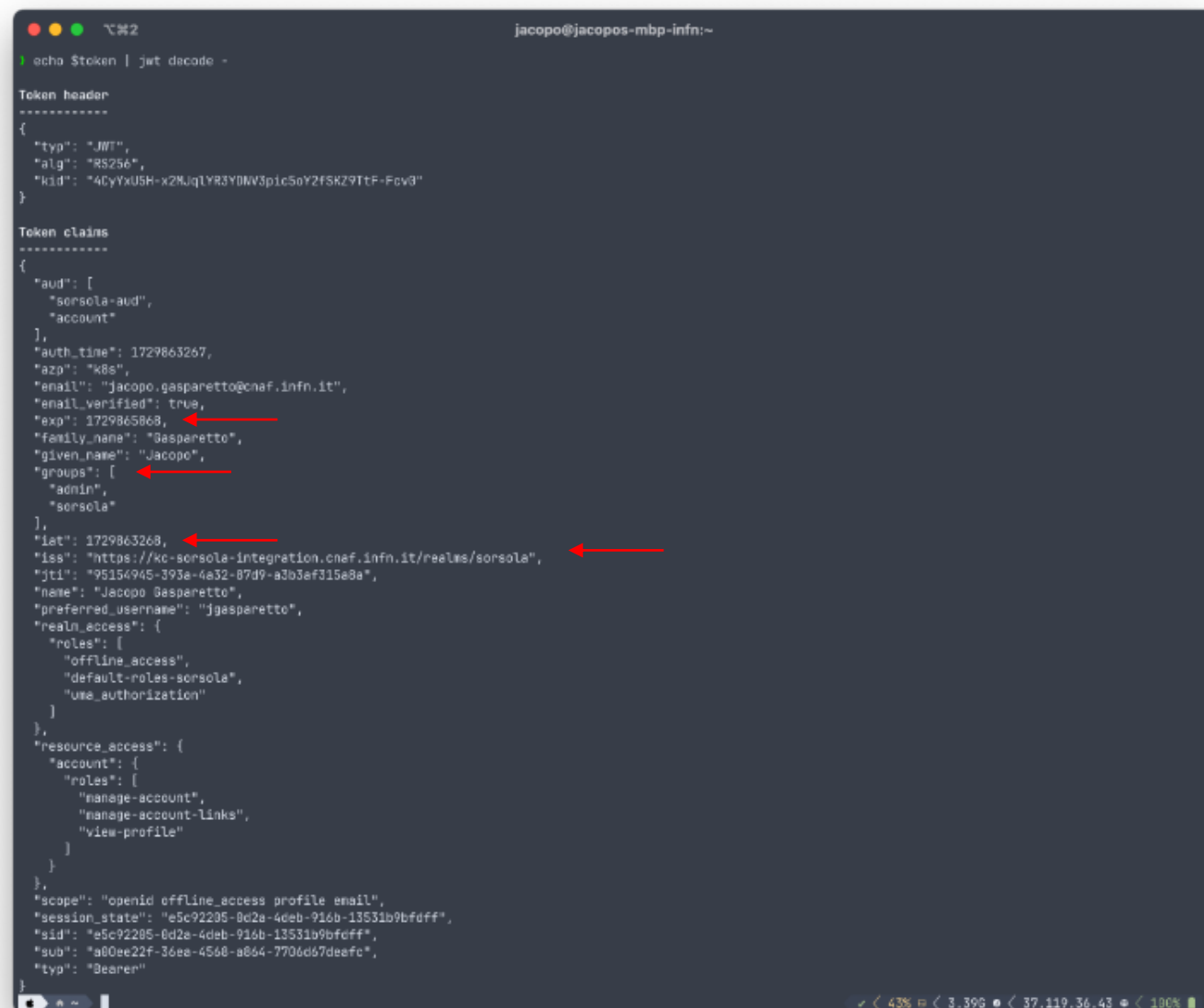
Motivation

- Centralized user management through a centralized Identity Provider (IdP), such as Keycloak.
- Authorization based on Role-Based Access Control (RBAC), assigned according to user group memberships.
- Each group associated with a user is mapped to a specific namespace in Kubernetes to control access accordingly (group mapping).



OAuth2 & OIDC

- OAuth2: Standard framework to delegate authorization.
- OIDC: built on top of OAuth for Authentication
- Based on JWT (Jason We Token)
 - **Header:** Contains basic metadata, such as the hashing algorithm used to sign the token.
 - **Payload:** The core content of the token, detailing specific information about the user or token.
 - **Signature:** A digital signature generated by the Identity Provider (IdP) that issued the token, confirming its authenticity and ensuring it hasn't been tampered with.



```

jaco@jacopo-mbp-infn:~
$ echo $token | jwt decode -

Token header
-----
{
  "typ": "JWT",
  "alg": "RS256",
  "kid": "4CyYxU5H-x2MjqlYR3Y0NV3pic5oY2fSK29ItF-Fcv3"
}

Token claims
-----
{
  "aud": [
    "sorsola-aud",
    "account"
  ],
  "auth_time": 1729863267,
  "azp": "k8s",
  "email": "jacopo.gasparetto@cnaf.infn.it",
  "email_verified": true,
  "exp": 1729865868,
  "family_name": "Gasparetto",
  "given_name": "Jacopo",
  "groups": [
    "admin",
    "sorsola"
  ],
  "iat": 1729863268,
  "iss": "https://kc-sorsola-integration.cnaf.infn.it/realms/sorsola",
  "jti": "95154945-393a-4a32-87d9-a3b3ef315e8a",
  "name": "Jacopo Gasparetto",
  "preferred_username": "jgasparetto",
  "realm_access": {
    "roles": [
      "offline_access",
      "default-roles-sorsola",
      "ume_authorization"
    ]
  },
  "resource_access": {
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links",
        "view-profile"
      ]
    }
  },
  "scope": "openid offline_access profile email",
  "session_state": "e5c92205-8d2a-4deb-916b-13531b9bfeff",
  "sid": "e5c92205-8d2a-4deb-916b-13531b9bfeff",
  "sub": "a00ee22f-36ee-4568-a864-7706d67deefc",
  "typ": "Bearer"
}

```


oidc-agent: token request

- To request an Access Token

```
oidc-token sorsola
```

- To save the token

```
token=$(oidc-token sorsola)
```

- To use the token with Kubernetes:

```
kubectl --token=$token get pod
```

or

```
alias k="kubectl --token=$token"
```

```
k get pod
```

Note: Each time a new token is requested, you'll need to reset the alias.

Best practises

• Login and password

La password:

- **DEVE** avere elevata robustezza (per Epic min 14 caratteri di 3 classi diverse), non essere banale o presente nei dizionari di qualsiasi lingua
- **DEVE** essere modificata con sufficiente frequenza
- **NON DEVE** essere la stessa per più di un servizio
- **NON DEVE** essere salvata in chiaro
- **NON DEVE** essere salvata se il pc è condiviso
- **NON DEVE** essere inserita in pagine web di dubbia provenienza (es: siti che ti calcolano quanto è robusta la tua password)

• Tokens

- **MAI** inviare il proprio token a qualcun altro
- **NON** salvare i token in chiaro

