INTRODUZIONE AI CONTAINER

Docker

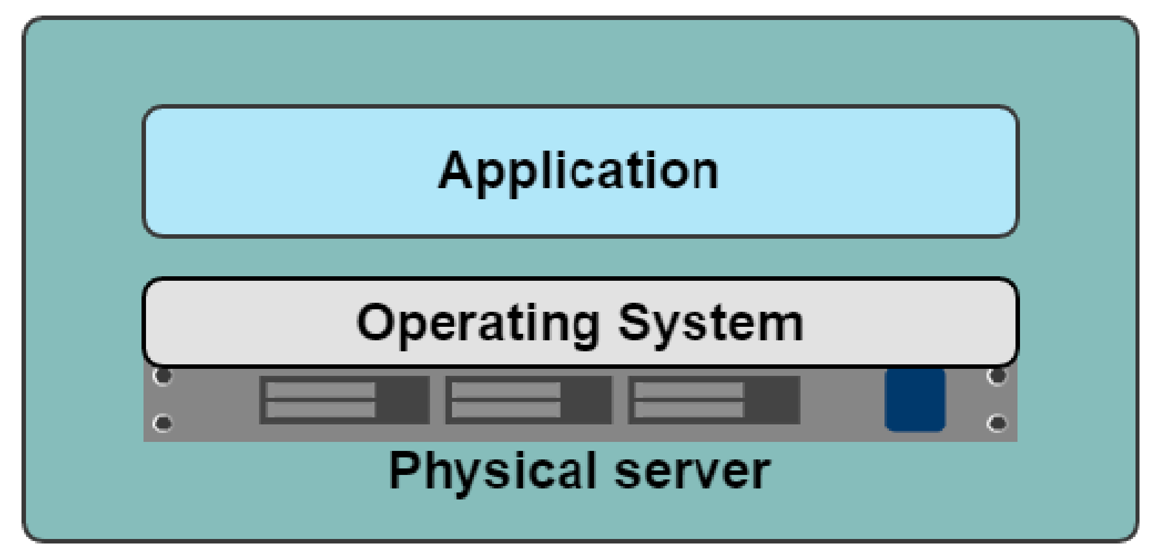
Le ere della virtualizzazione

L’era delle macchine fisiche

Ogni applicazione viveva su un server fisico dedicato.

Limitazioni:

- Tempi di deploy molto lenti
- **Costi molto alti**
- **Molte risorse sprecate**
- Difficoltà nello scalare le applicazioni
- Difficoltà nel migrare le applicazioni
- Difficoltà nel ospitare più applicazioni sullo stesso sistema operativo



Virtualizzazione

- Processo di disaccoppiamento tra i servizi di infrastruttura e l’hardware sottostante (allocazione e disallocazione rapida di risorse)

- Il servizio (e.g. risorsa computazionale, rete, storage) è gestito tramite software ed espone delle API, abilitando operazioni CRUD per i clienti

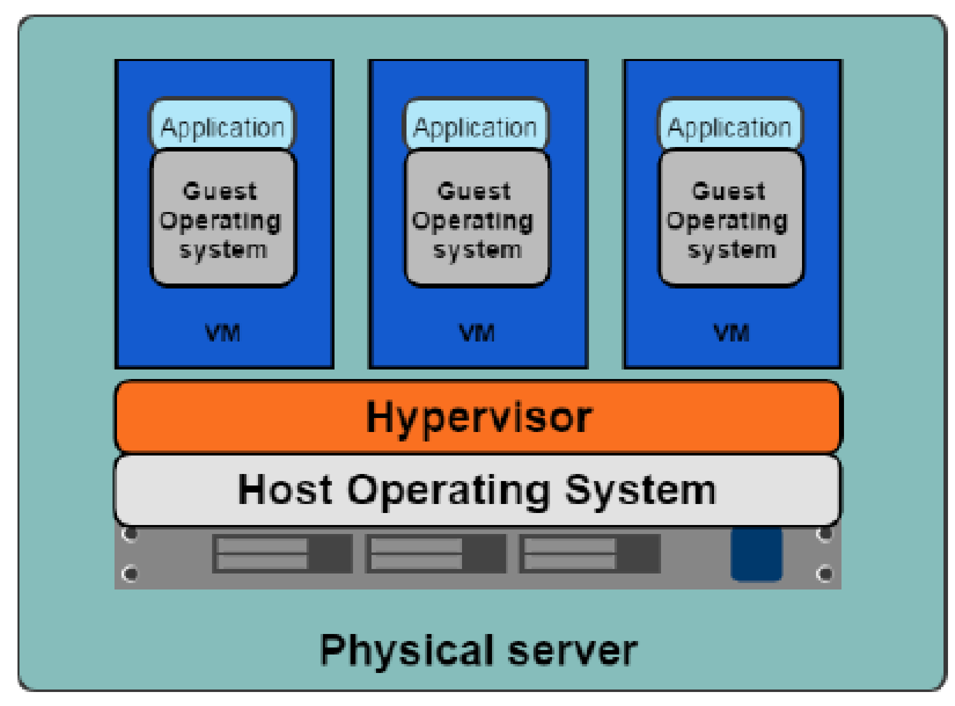
- Nel cloud tutti i servizi di infrastruttura sono virtualizzati
    - Compute virtualization
    - Network virtualization
    - Storage virtualization

L’era delle macchine virtuali

In questa era si sviluppano le Virtual machine che permettono ai server fisici di **contenere più di un’applicazione**.

Anche la **scalabilità** ne ottiene dei benefici, così come la possibilità di gestire meglio le risorse fisiche **riducendo i costi**.

La nascita **dell’IaaS** permette di avere Macchine Virtuali nel cloud.



L’era delle macchine virtuali

Limitazioni delle macchine virtuali:

- Ogni Virtual Machine necessita di molte risorse: CPU, Spazio disco, RAM ed **un intero sistema operativo completo**
- Un sistema operativo completo implica molte risorse sprecate
- La portabilità non è garantita

Importanza della virtualizzazione per il Cloud

- Consente di gestire **pool** di risorse computazionali fisiche in cloud
- Uso efficiente di tutte le risorse fisiche disponibili, **riduzione di costi** (sia per il provider che per il consumer)
- Allocazione dinamica delle risorse (compute, network, storage) sulla base delle richieste dei clienti
- Elasticità rapida

Virtualizzazione delle risorse computazionali

- Astrarre l'hardware rispetto al sistema operativo
- Consente a più sistemi operativi ed applicazioni di girare simultaneamente sullo stesso server fisico, cioè si crea un ambiente di esecuzione **isolato (virtual machine)**
- La virtualizzazione consente di far apparire un unico sistema fisico come se fossero molti sistemi
- Le applicazioni non possono accedere a risorse di altre VM se non tramite interfacce di rete (isolamento = maggiore sicurezza)
- Eventuali patch di OS vanno applicate a tutte le VM

HyperVisor

- Disaccoppia l’hardware fisico dalla virtual machine, cioè emula l’hardware fisico in software (a.k.a. Virtual Machine Monitor)
- Si trova tra l’hardware e il S.O. della virtual machine (usa la tecnica dell’emulazione)
- Fornisce ambienti di esecuzione isolati per le VMs
- Esempi: VMWare ESX Server, Linux KVM, Hyper-V, XenServer

Funzioni dell’HyperVisor

- Responsabile primario di:
    - Creare ed eseguire VM isolate
    - Partizionare e condividere dinamicamente le risorse fisiche (e.g. CPU, RAM, I/O, storage) dell’hardware sottostante tra le diverse VM

Tipologie di HyperVisor

- Due tipi di HyperVisor
    - Bare-metal HV, a.k.a. Type 1
        - Gira direttamente sull'hardware e ospita le VM
        - Non richiede un S.O. installato sull'hardware ospitante
        - Esempio: VMWare ESX
    - Type 2
        - Gira direttamente sul S.O.
        - Esempio: VMWare Fusion, Oracle VirtualBox
    - KVM (Kernel Virtual Machine) usa il kernel di Linux come hypervisor
        - Possiede aspetti sia di Type 1 che di Type 2

Host

- La macchina fisica sulla quale avviene la virtualizzazione
- Tipicamente un Intel x86 o un AMD
- Vi gira il software HV che crea le virtual machine
- Alcuni stack cloud lo definiscono come **compute node**

Guest

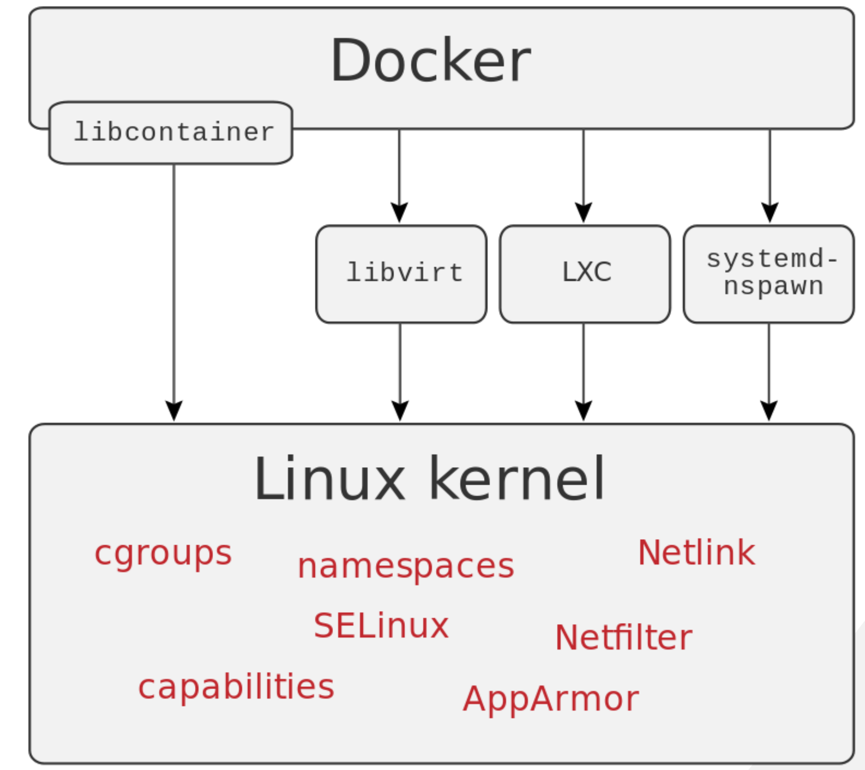
- La macchina virtuale stessa
- Ci girano
  - Un sistema operativo completo (incluse dipendenze e patches)
  - Le applicazioni utente
- E’ memorizzata sull’hardware fisico come insieme di files
  - Disco virtuale, file di configurazione, file di paging, file del BIOS, file di log, etc …
- Viene riferita come istanza nei sistemi Cloud (e.g. OpenStack, AWS, GCE)

L'era dei container

L’idea di base dei container è quella di utilizzare il kernel del sistema operativo della macchina host per eseguire tanti ***root file system***.

Ogni root file system è chiamato container e su ogni container abbiamo:

- Processi (di cui uno principale)
- memoria
- stack di rete



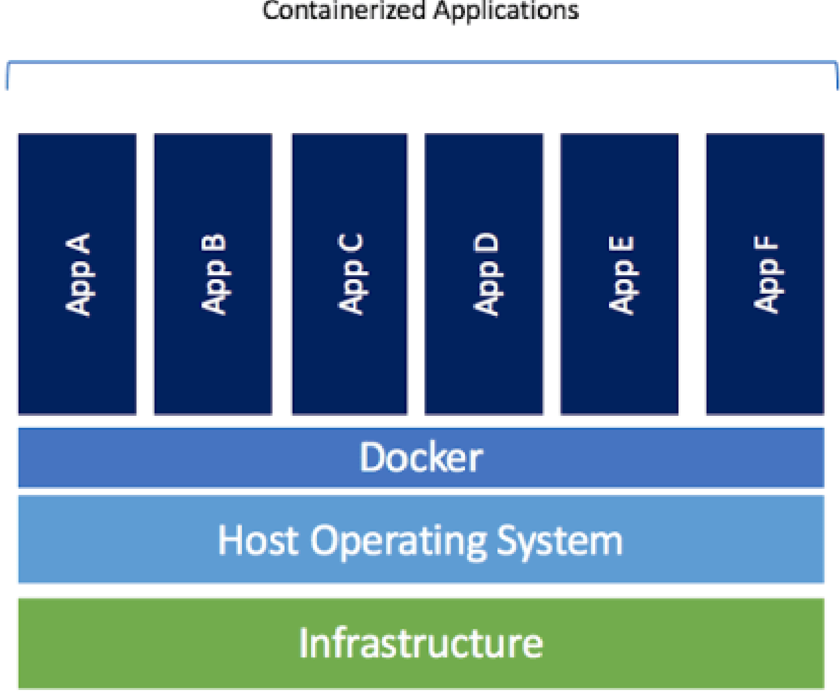
Cos’è un container

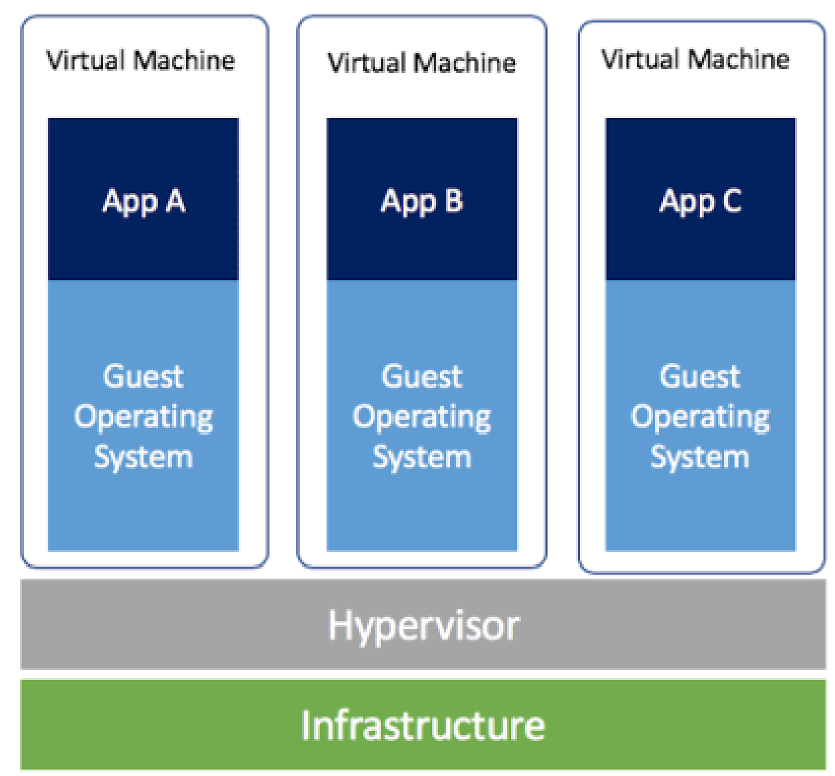
- Applicazione completamente isolata che gira nativamente nel kernel della macchina host
  - Completo isolamento da altri container e dalla macchina host. Un container può accedere ai file ed alle porte di rete della macchina host solo se configurato per poterlo fare
  - Esegue applicazioni direttamente nel kernel dell’host. Un container non ha bisogno di un Hypervisor
- I container che girano sull’host condividono il kernel dell’host
  - Virtualizzazione del S.O. e non dell’hardware
- Ogni container esegue un singolo componente software
  - E.g. un container che esegue Apache HTTP Server

Come si crea un container

- I containers sono creati a partire da immagini
- Un immagine contiene informazioni circa:
  - Il codice eseguibile
    - E.g. codice Python o Java
  - Ambiente necessario per eseguire il codice
    - Un runtime come un interprete Python o una JVM
    - Librerie utilizzate dal codice
    - Variabili di environment
    - File di configurazione
- Un container engine (e.g. Docker) esegue l’immagine direttamente nel kernel della macchina host e crea il container

L'era dei container





Container VS Virtual Machine

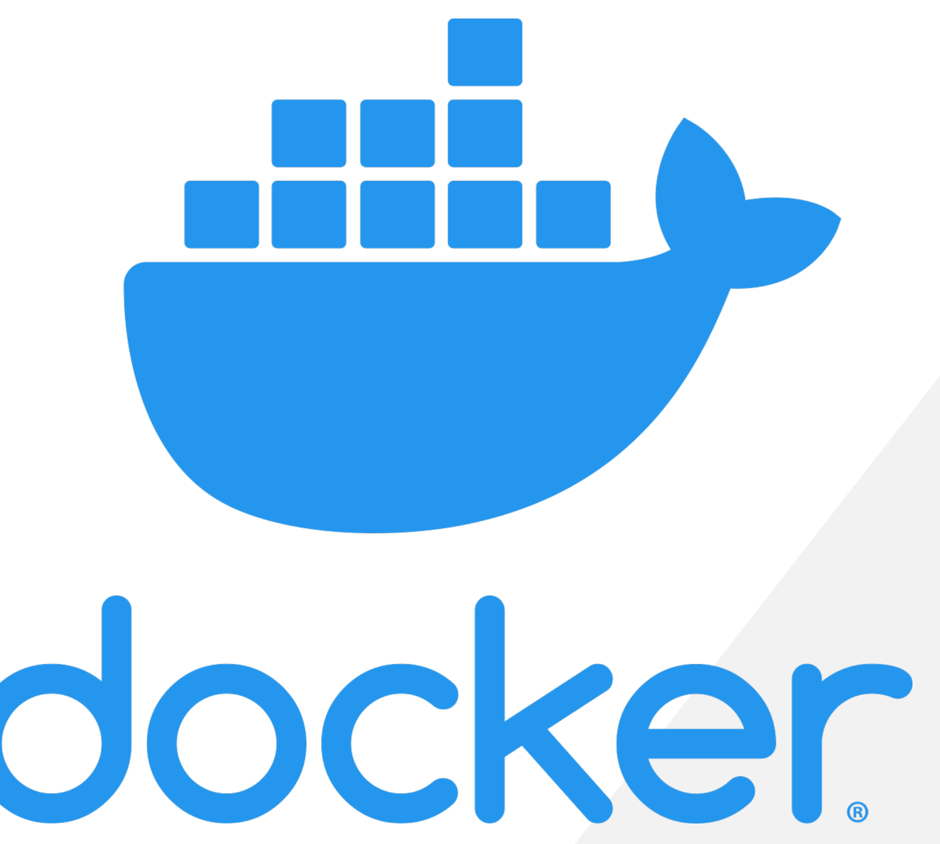
Vantaggi rispetto alle VM:

- Velocità e leggerezza
- Non necessitano di un sistema operativo dedicato
- Necessitano di meno risorse: RAM, CPU, Spazio disco
- A parità di condizioni, su un host, è possibile lanciare più container di macchine virtuali
- Grande portabilità
- Sono più semplici da gestire
- Sono la soluzione più efficace per sviluppare e rilasciare microservizi

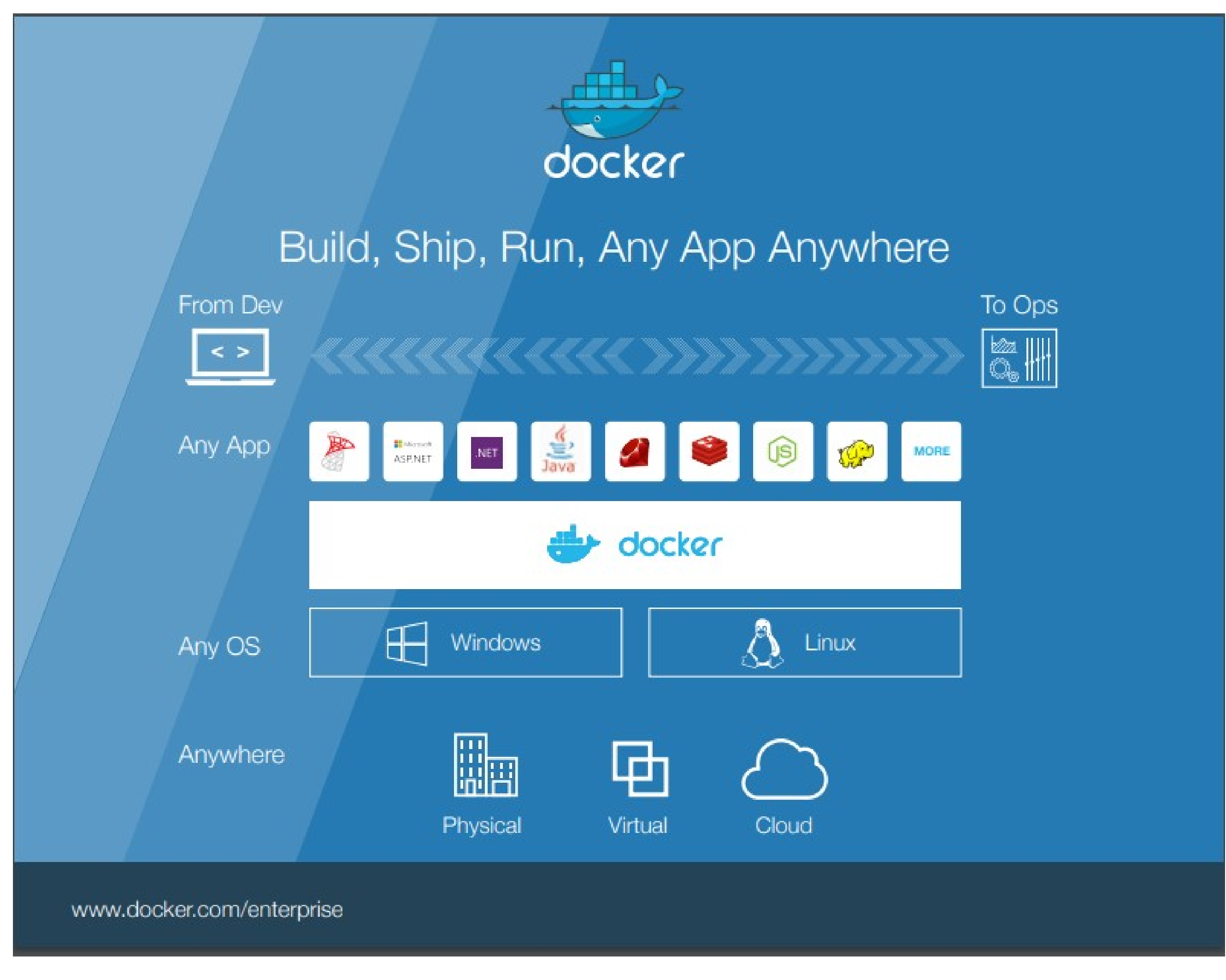
Docker

Cos'è Docker?

Docker is an **open platform** for **developing**, **shipping**, and **running applications**. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can **manage your infrastructure in the same ways you manage your applications**. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly **reduce the delay between writing code and running it in production**.



Docker: Build, Ship and run



Docker Platform

Docker provides the ability to **package** and **run** an application in a **loosely isolated environment** called a container. The isolation and security allow you to **run many containers simultaneously on a given host**. Containers are **lightweight** because they don’t need the extra load of a hypervisor, but r**un directly within the host machine’s kernel.** This means you can run more containers on a given hardware combination than if you were using virtual machines. You can even run Docker containers within host machines that are actually virtual machines!

Docker Platform

Docker provides tooling and a platform to manage the lifecycle of your containers:

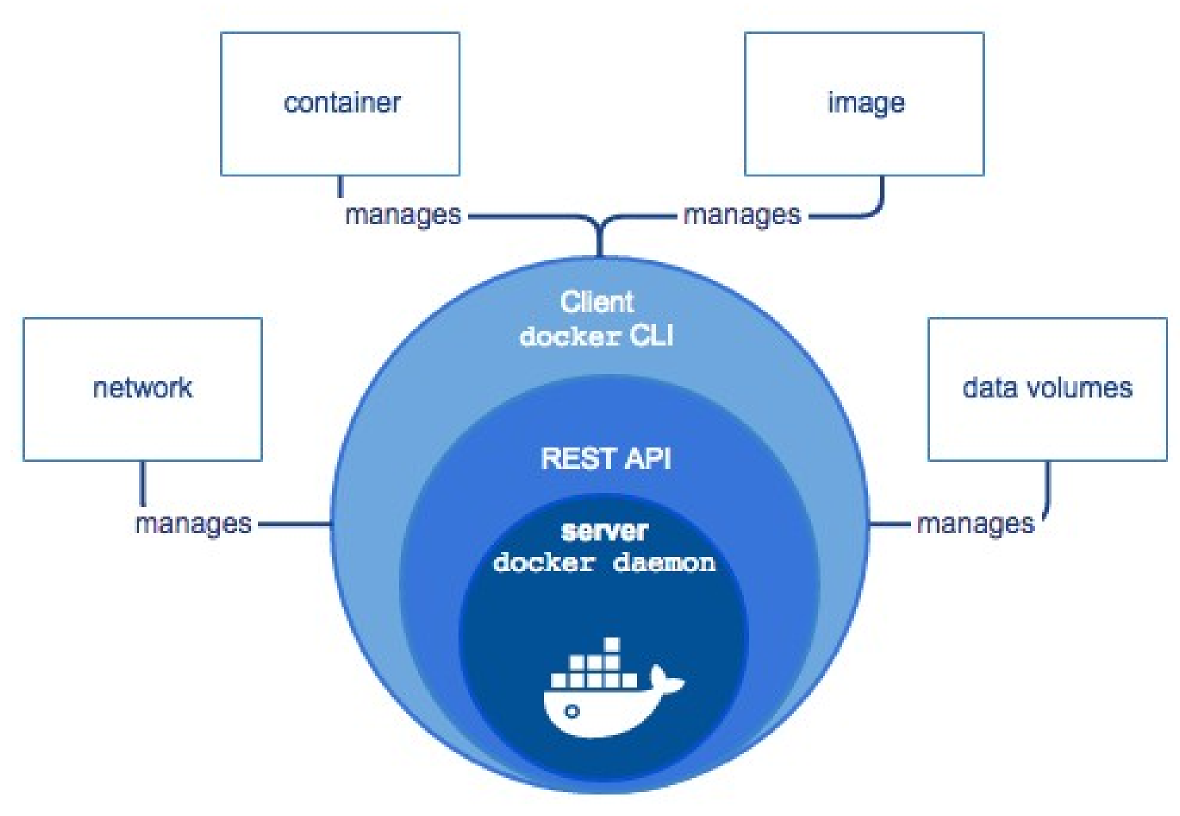
- Develop your application and its supporting components using containers.
- The **container** becomes the **unit for distributing** and testing your application.
- When you're ready, deploy your application into your production environment, as a container or an orchestrated service. **This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.**

Docker Engine

È un applicazione client/server che permette di distribuire ed eseguire i container

È composto da:

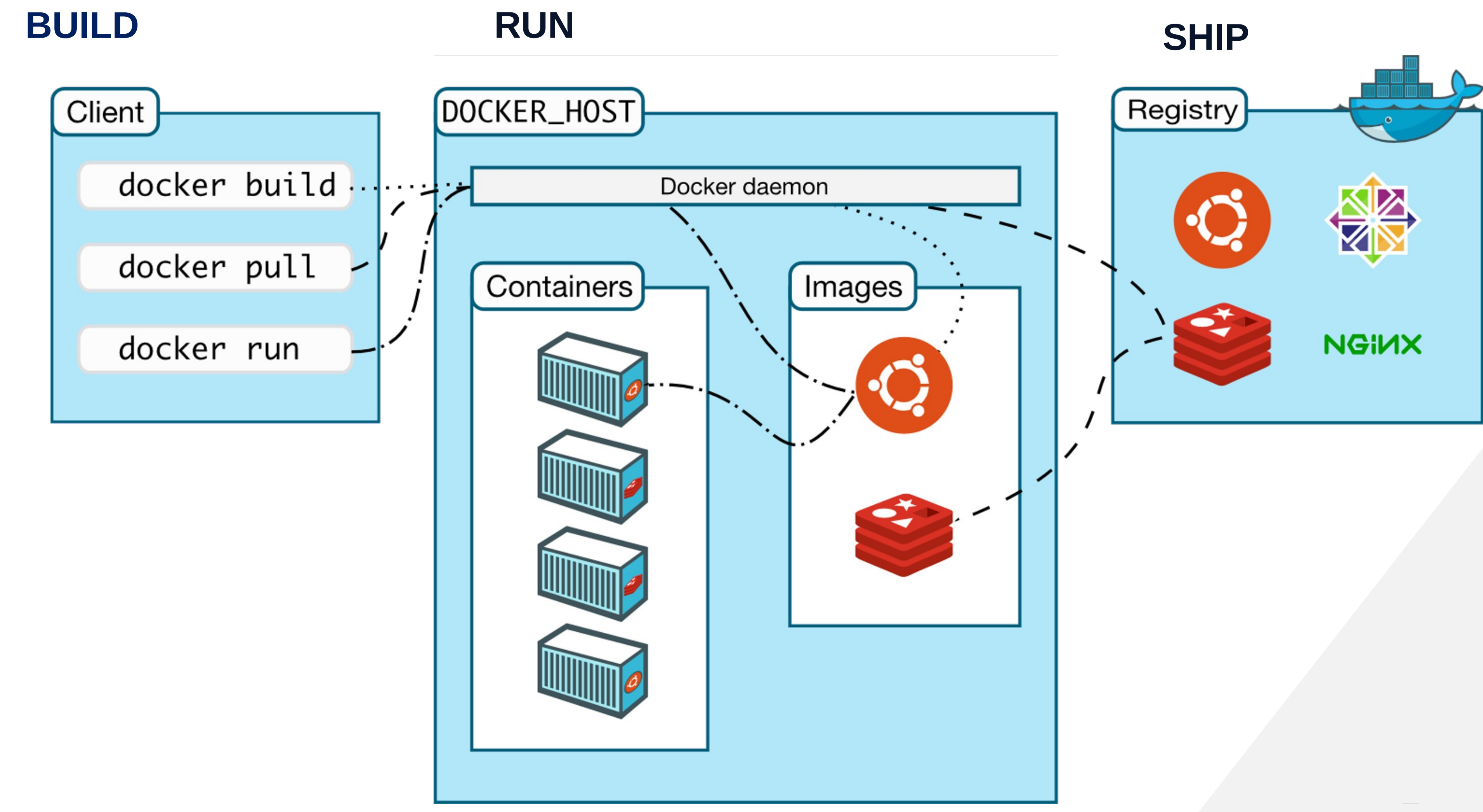
- **Server** (dockerd) che utilizza i namespaces del Kernel Linux e i control group per isolare l'esecuzione del container
- **APIREST** che permettono ai client di interfacciarsi con il Server e gestire i container
- **Client** (docker) CLI per interagire con il Server attraverso le API REST



Come sono isolati i container

- Docker utilizza due tecnologie contenute nel kernel di Linux per isolare i container
  - Namespaces
    - Forniscono un ambiente isolato per un container
    - Quando si esegue un processo in un namespace, l’accesso è limitato a quel namespace
    - Docker isola ogni aspetto di un container utilizzando namespaces quali Process ID (PID), rete, IPC, mount per hostname e domain-name
  - Control Groups (cgroups)
    - Utilizzati per limitare le risorse (e.g. memoria) disponibili per un container

Architettura Docker



Concetti Docker

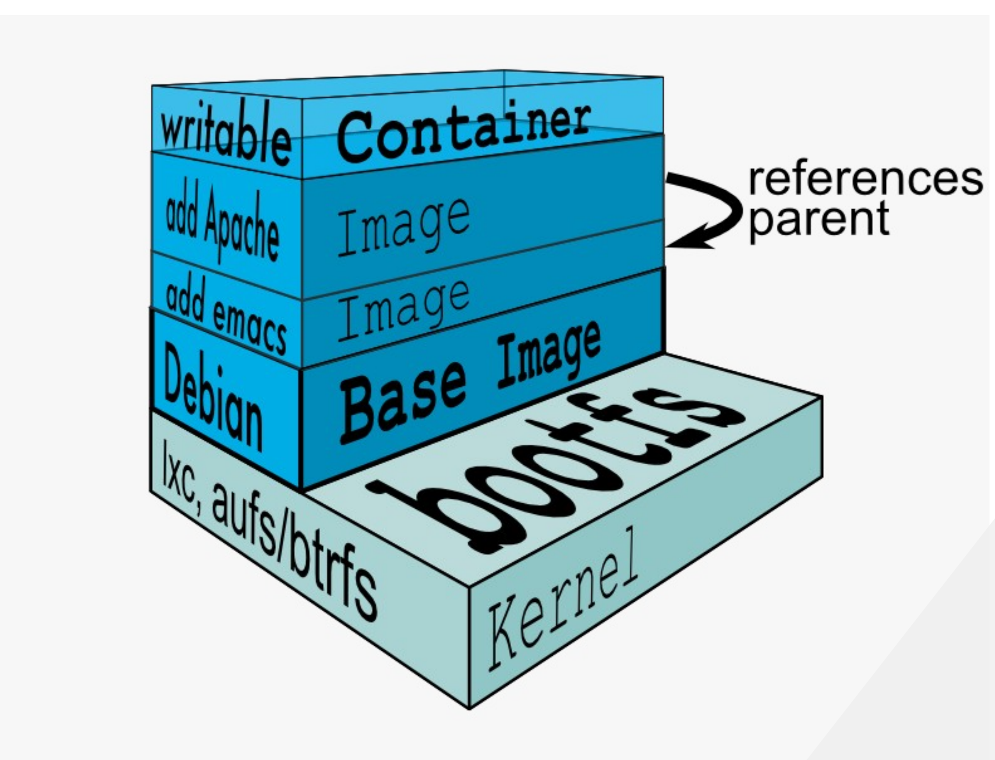
Docker Image

La base per costruire un container. **Rappresenta l'intera applicazione**.

Un'immagine è un **file immutabile** che rappresenta un'istantanea di un container.

Le immagini sono costituite da **strati di altre immagini** e vengono create con l'operazione di build a partire da un file descrittore chiamato **DockerFile**.

Possono essere condivise e scaricate da un registry



Dockerfile

11 lines (6 sloc) | 231 Bytes

Raw | Blame | History

```
FROM tiangolo/meinheld-gunicorn-flask:python3.7

COPY /dist/flaskr-1.0.0-py2.py3-none-any.whl /app

RUN pip install /app/flaskr-1.0.0-py2.py3-none-any.whl

ENV FLASK_APP=flaskr

RUN flask init-db

ENV APP_MODULE flaskr:create_app()
```

Docker Container

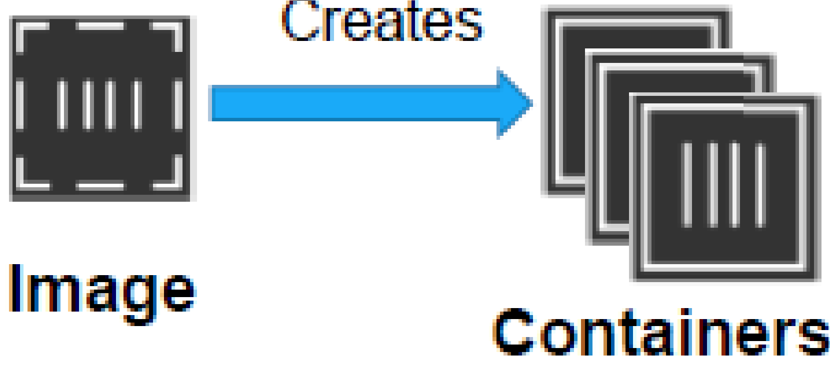
L'unità standard in cui risiede ed esegue il servizio dell'applicazione

L'applicazione esegue in modo isolato

Contiene tutto il necessario per eseguire l'applicazione

Possibilità di gestire:

- Volumi esterni
- Tipo di rete e porte utilizzate



Docker Registry

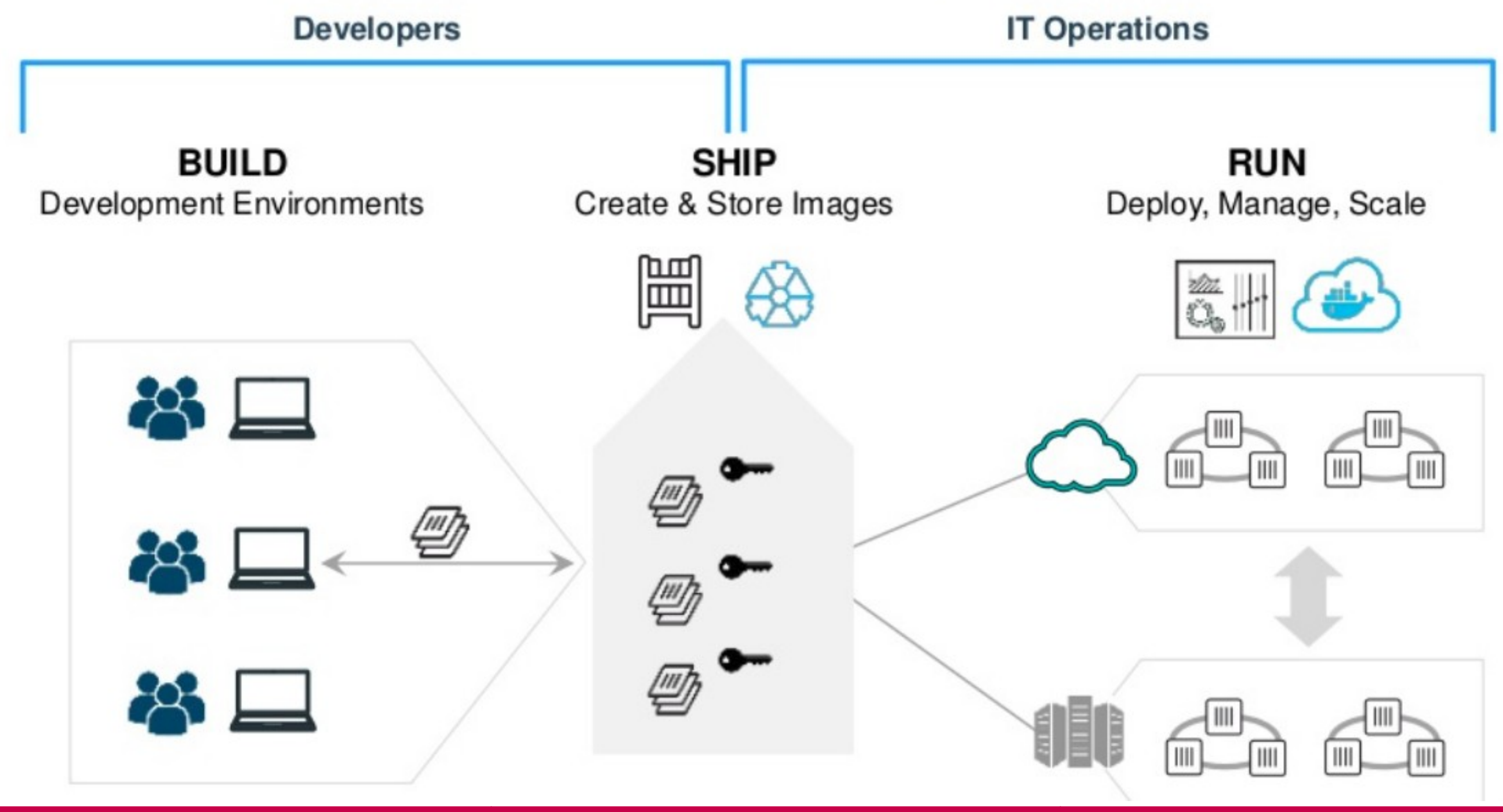
Cloud or server based storage and distribution service for your image.

Il registry Cloud più conosciuto è Docker Hub

- È possibile scaricare tutte le immagini pubbliche
- È possibile creare un account personale per pubblicare le proprie immagini

È consigliato avere un Trusted Registry privato dove salvare ed utilizzare le immagini certificate

Visione generale



Docker Networking

I container docker sono collegati in rete attraverso network drivers

Docker è distribuito con alcuni network drivers, abilitando per default il bridging e l'overlay

I container sono spesso collegati in rete attraverso bridging e overlay

Il networking di Docker può essere esteso grazie a plugins

- Usano le API del progetto open source LibNetwork
  - e.g. Kuryr network plugin, Cisco Contiv

Docker Networking

Ogni container può avere più interfacce di rete

Un container usa l'interfaccia di rete per comunicare su una rete specifica

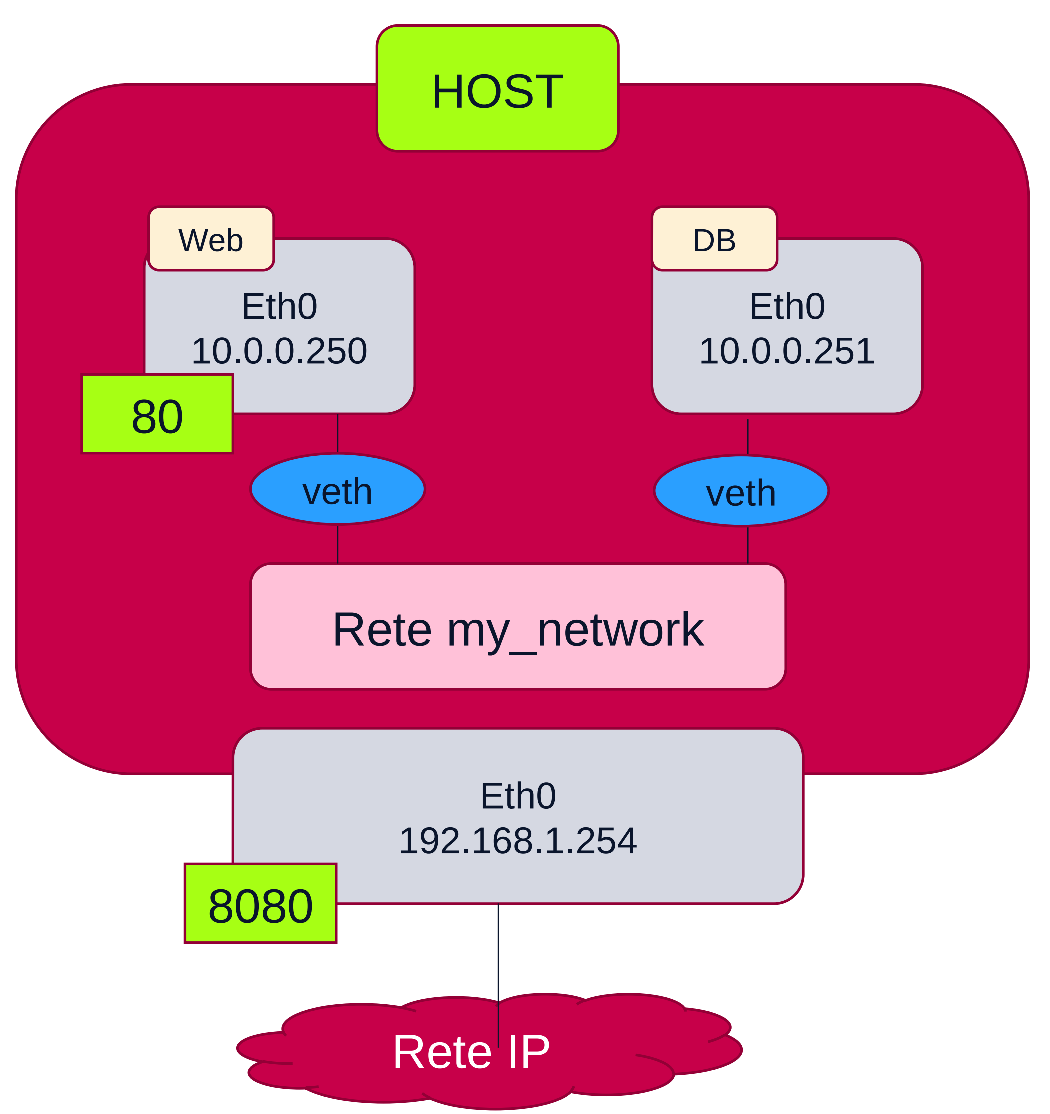
Una rete è un dominio di broadcast al livello 2

- Container sulla stessa rete comunicano liberamente
- Container su reti diverse sono isolati

Ogni container ha un indirizzo IP assegnato da una subnet privata da Docker

- Questa subnet è separata dall'hardware di rete sottostante

Bridge Networking



In Docker gira un DNS privato

Non è l’approccio suggerito

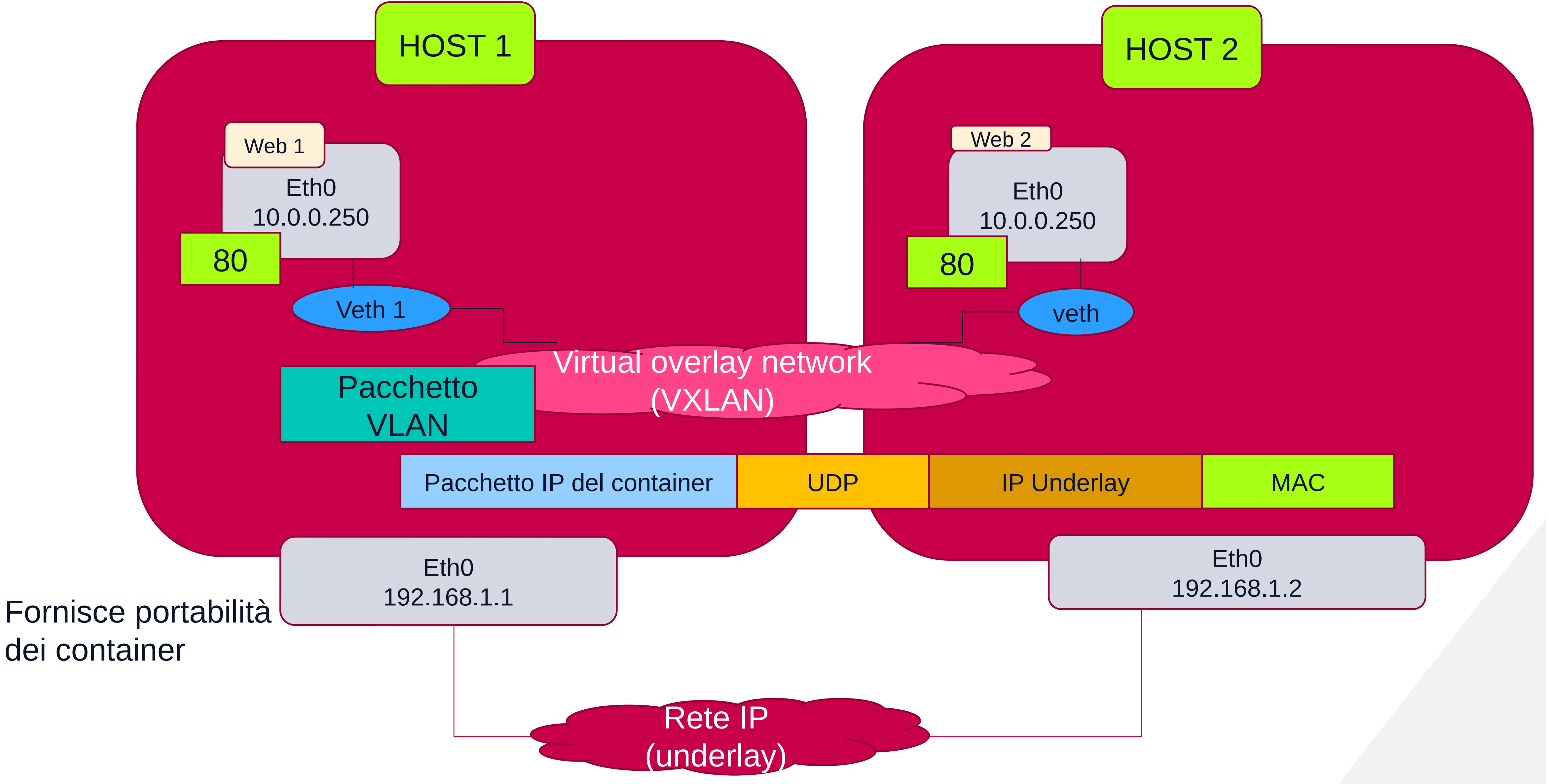
Docker network create

Veth operano da link

Per far raggiungere le applicazioni dall’esterno bisogna mappare le porte (e.g. 8080 -> 80)

La pubblicazione della porta avviene solo a runtime

Overlay Networking



Fornisce portabilità
dei container

Orchestration

What and why of orchestration

- Esistono molti tipi di orchestratori
- Prendono decisioni su **quando** e **dove eseguire attività**
- Sono stati utilizzati fin dagli albori dell'informatica: programmazione di Mainframe, Puppet, Terraform, AWS, Mesos, Hadoop, ecc.
- Dal 2014 è nata la necessità di creare nuovi progetti di orchestrazione:
  1. Popolarità del calcolo distribuito
  2. Diffusione dei Docker per distribuzione ed esecuzione di applicazioni in modo isolato
- Necessità di avere "molti server da gestire come uno solo per eseguire molti containers"
- Nascita dei **Container orchestrator**

Container orchestrator

Molti progetti (open source) sono stati creati negli ultimi 5 anni per:

- *Schedule running of containers on servers*
- *Dispatch them across many nodes*
- *Monitor and react to container and server health*
- *Provide storage, networking, proxy, security, and logging features*
- *Do all this in a declarative way, rather than imperative*
- *Provide API's to allow extensibility and management*

Principali progetti di container orchestration

- Docker Swarm
- Kubernetes, aka K8s
- Apache Mesos/Marathon
- Cloud Foundry (sviluppata da VMware)
- Amazon ECS

Cloud platforms for Kubernates

- Alibaba Cloud *(Aliyun)* **ACK**
- Amazon Web Services *(AWS)* **EKS**
- DigitalOcean *(DO)* **Kubernetes**
- Google Cloud Platform *(GCP)* **GKE**
- Microsoft Azure **AKS**
- Oracle Cloud Infrastructure *(OCI)* **OKE**

Docker swarm e compose

Docker Compose

Docker Compose è un tool che permette di **definire ed eseguire applicazioni multi-container.**

Con Docker Compose è quindi possibile ricreare **l'intero ambiente di esecuzione** di un'applicazione che prevede **più di un container** definendoli come servizi all'interno di un file yml nel quale vengono specificate tutti gli attributi necessari nei comandi di run dei container.

Docker Compose utilizza il nome del progetto per definire un ambiente completamente isolato dagli altri gestendo in modo autonomo reti e volumi.

docker-compose.yml

Esempio di file docker-compose

[Docker Compose ambiente laboratorio]

Docker Swarm

Uno swarm ("sciame" in italiano) consiste in un insieme di nodi Docker che si comportano da manager, per la gestione del cluster stesso, e worker, che eseguono servizi.

Swarm si proponeva quindi come lo **strumento di default** per la gestione di container in un cluster di nodi Docker.

Docker Swarm

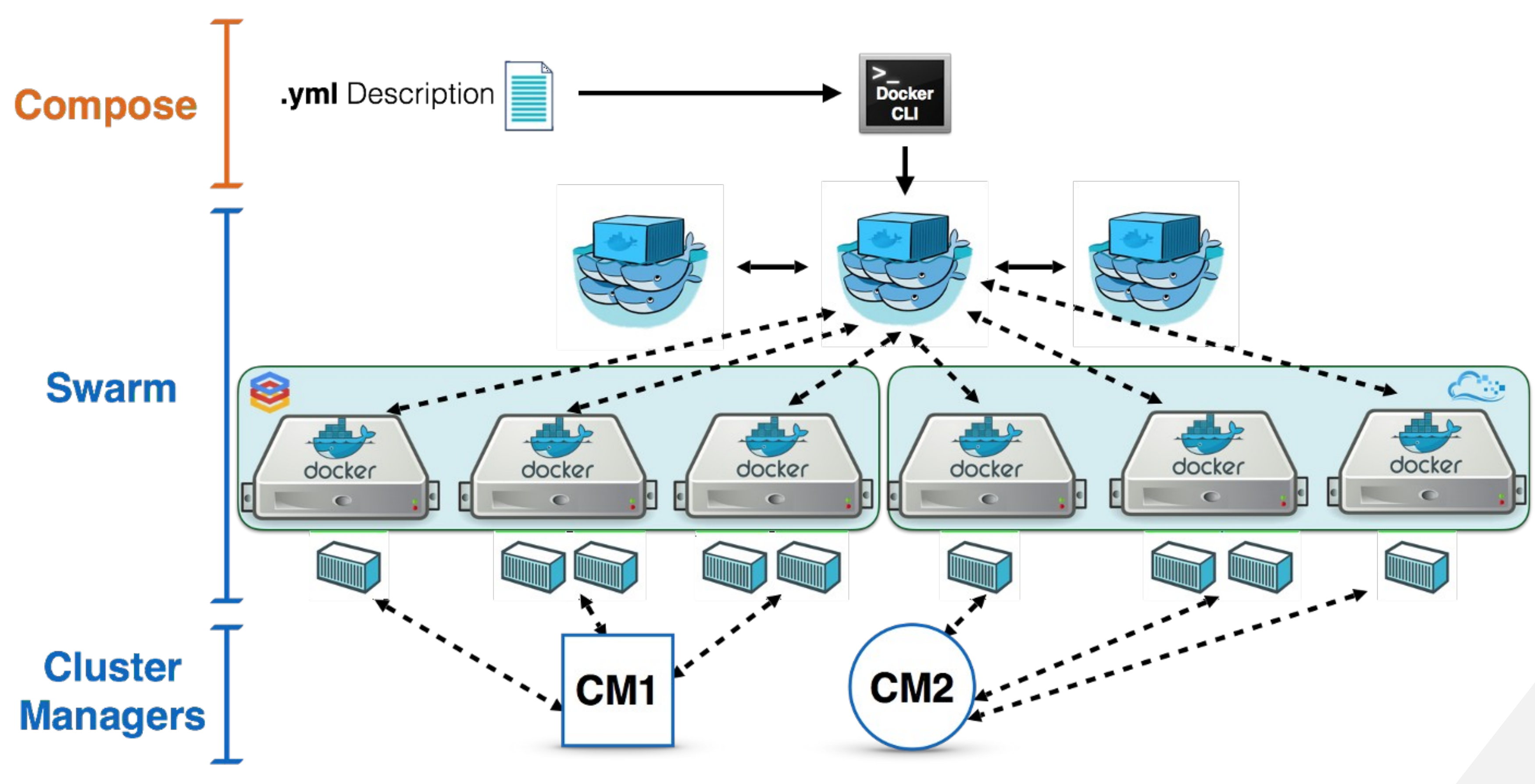
Nodi

- si tratta di **istanze di Docker** che partecipano al cluster. Solitamente, ogni istanza del Docker Engine risiede su *una* macchina fisica o virtuale che sia.
- possono essere di tipo ***manager*** o ***worker*** (o **entrambi**): più nodi possono avere il ruolo di master, ma solo uno è eletto in un certo momento a coordinare i servizi e a mantenere lo stato del cluster.
- Il **deploy** di una applicazione **passa sempre da un nodo master** che **delega un *task* ad un nodo worker** (o un altro master, o a se stesso) in modo che *esegua il servizio richiesto*. I nodi del cluster informano sempre il nodo master sullo stato dei task assegnati, in modo da mantenere sempre lo stato dei servizi come richiesto.

task e ***servizi***:

- un nodo master assegna il compito (*task* appunto) di eseguire un certo *servizio* su un nodo del cluster (possibilmente worker), ovvero avviare una o più istanze di un container a partire da un'immagine.
- l’interazione tra il nodo master e la CLI è **asincrona**: da qua si evince che c’è qualcuno che sta facendo qualcosa (task) dopo che abbiamo eseguito un comando.

Architettura Docker Swarm



Kubernates

Kubernates

Kubernetes is an open source system for managing containerized applications across multiple hosts. It provides basic mechanisms for deployment, maintenance, and scaling of applications.

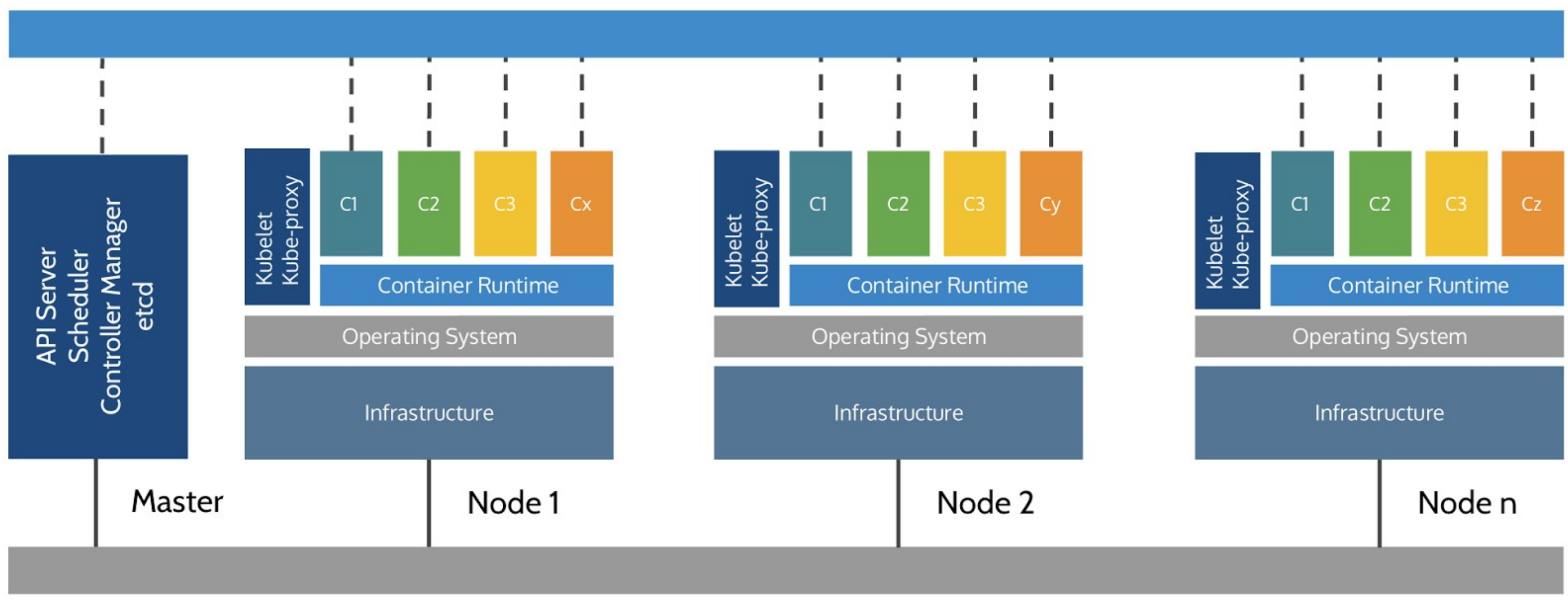
Kubernetes builds upon a decade and a half of experience at Google running production workloads at scale using a system called Borg, combined with best-of-breed ideas and practices from the community.

Kubernetes is hosted by the Cloud Native Computing Foundation (CNCF). If your company wants to help shape the evolution of technologies that are container-packaged, dynamically scheduled, and microservices-oriented, consider joining the CNCF. For details about who's involved and how Kubernetes plays a role, read the CNCF announcement.

Funzionalità di Kubernates

- Basic autoscaling
- Blue/green deployment, canary deployment
- Long running services, but also batch (one-off) and CRON-like jobs
- Overcommit our cluster and evict low-priority jobs
- Run services with stateful data (databases etc.)
- Fine-grained access control defining what can be done by whom on which resources
- Integrating third party services (service catalog)
- Automating complex tasks (operators)

Kubernates Architecture



Pod

Pods are the **smallest deployable units** of computing that can be created and managed in Kubernetes.

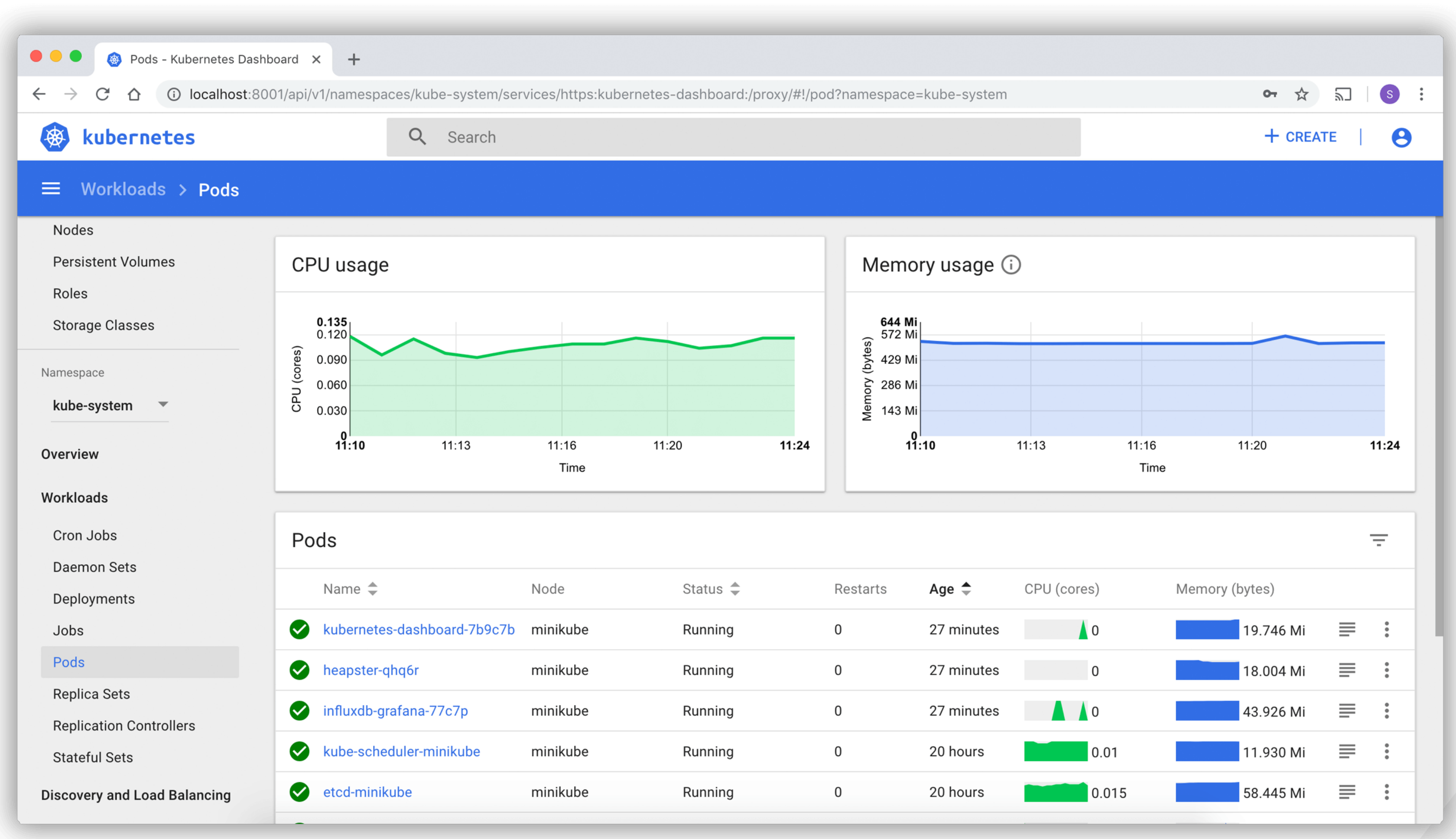
A *Pod* (as in a pod of whales or pea pod) is a group of **one or more** containers (such as Docker containers), with shared storage/network, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host" - it contains one or more application containers which are relatively tightly coupled

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: hello-world
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
      - name: hello-world-server
        image: gcr.io/megangcp/helloworld:v0.0.1
        ports:
        - containerPort: 8080
```

Pod

Docker Image

Kubernate Web UI



Riferimenti

https://www.slideshare.net/Docker/docker-101-nov-2016

https://www.slideshare.net/MeganOKeefe1/kubernetes-a-short-introduction-2019

https://www.docker.com/sites/default/files/Infographic_OneDocker_09.20.2016.pdf

https://docs.docker.com/get-started/

https://www.docker.com/blog/containers-replacing-virtual-machines/

https://it.wikipedia.org/wiki/Docker

https://docs.docker.com/engine/reference/commandline/docker/

https://slides.kubernetesmastery.com/

https://codingjam.it/da-docker-compose-a-docker-swarm-viaggio-di-sola-andata/

https://github.com/kubernetes/kubernetes

https://medium.com/containermind/a-reference-architecture-for-deploying-wso2-middleware-on-kubernetes-d4dee7601e8e

https://docs.docker.com/compose

https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/