Docker e i container: cosa sono, perché esistono e quando è utile usarli

5 Agosto 2021 - 10 min. read

Containers Docker

Se lavori nel settore dell'Information Technology, indipendentemente dalla tua posizione lavorativa, avrai sicuramente sentito parlare di *container software*. Magari ci lavori tutti giorni, magari ci hai lavorato qualche volta, o magari li hai solamente sentiti nominare.

Si sente parlare di questa tecnologia da non più di 10 anni, ma il concetto di container è ben più datato.

Lo scopo di questo articolo è di ripercorrere la storia dei container, capire da dove nascono fino ad arrivare ai giorni nostri. Discuteremo dello standard principalmente utilizzato (Open Container Initiative) e faremo chiarezza tra le diverse componenti tecnologiche (immagine, container, runtime). Infine, vedremo possibili alternative e alcune applicazioni nel mondo AWS.

Cos'è un container?

Non è facile dare una risposta semplice a questa domanda, iniziamo con quanto riportato nella homepage di Docker:

Un container software è un unità software che racchiude codice e tutte le sue dipendenze, affinchè l'applicazione possa essere eseguita nella stessa maniera in ambienti computazionali diversi.

Proviamo ad analizzare meglio la risposta. Quando un'applicazione viene pacchettizzata in un container, può, appunto, essere eseguita su diverse macchine con la certezza che verrà eseguita alla stessa maniera. Da questo punto di vista, può sembrare molto simile ad una Virtual Machine.

Ma il container è molto più che questo.

Iniziamo dall'etimologia della parola *container*. Tutti noi conosciamo i container utilizzati su navi, treni e camion per il trasporto di merci. Immaginiamo quindi i *container software* come una sorta di involucro per la nostra applicazione, con una propria grandezza, uno standard, e dei meccanismi di locking verso il mondo esterno. Qualsiasi container, grazie alla sua forma, può essere spostato da una parte all'altra senza grossi problemi, indipendentemente dal suo contenuto.

Ma qual'è il contenuto dei container software? E' un'insieme di dipendenze, librerie, unitamente al codice applicativo, salvato sotto forma di immagine (*container image*) che può essere eseguito su qualsiasi macchina supporti l'esecuzione dei container.Ne consegue che lo stesso software, contenuto in un container, si comporterà alla stessa maniera sia che venga eseguito sulla macchina dello sviluppatore, sul server on premise o sulla virtual machine in cloud.

In generale, qualsiasi applicazione necessita anche delle sue dipendenze, che possono variare da macchina a macchina (a causa della diversa architettura hardware, diverso sistema operativo). Per avere coerenza, il codice dovrebbe essere preparato insieme alle sue dipendenze.

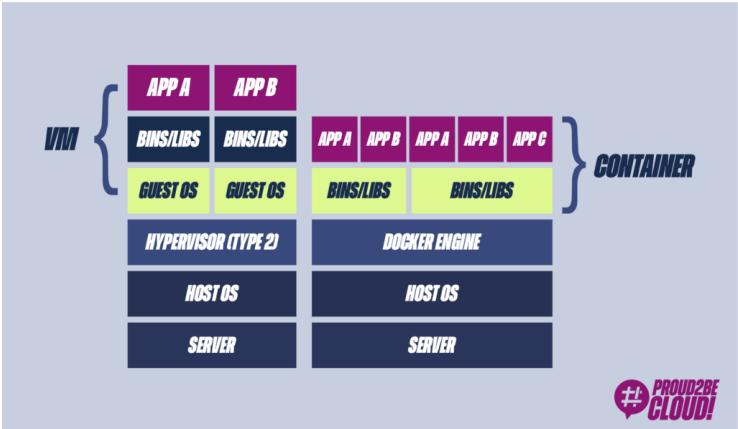
Nell'industria del software si sono infatti utilizzate per anni le Virtual Machines (VM). La Virtual machine non è altro che un'emulazione di un computer, compreso di sistema operativo, su cui è possibile installare tutte le tue dipendenze necessarie, rendendola indipendente dalla macchina host (codice, dipendenze e sistema operativo). Diventa però difficile utilizzare le VM come strumento di delivery del software, in quanto dovresti utilizzarne l'intera immagine, compreso il sistema operativo. Immagino sia capitato a tutti di dover creare una virtual machine sul proprio computer, e di riscontrarne la lentezza (provare per credere!)

Un altro problema riguardo le VM, è che rimaniamo legati ad un virtual hardware (attraverso hypervisor). Uno sviluppatore non dovrebbe preoccuparsi dello storage, del networking o del tipo di processore (o almeno, non a basso livello). Tornando

quindi alla metafora iniziale, il container dovrebbe essere indipendente da chi lo trasporta (nave, treno, truck ecc).

L'abbiamo già detto, ma un altro problema delle virtual machine sono appunto le performance, oltre a richiedere molte risorse hardware, soffrono generalmente di tempi di boot elevati. Una virtual machine, per quanto virtuale, rimane comunque una macchina completa, non è adatta di per sé al delivery di un software.

Riassumendo, possiamo quindi dire che per rilasciare del software affidabile e ripetibile su diversi computer, necessitano di un box ermetico in cui mettere il nostro codice. Questo box dovrebbe essere agnostico dal sistema su cui gira, in modo che lo sviluppatore possa concentrarsi sullo sviluppo del software e delle sue strette dipendenze, senza includere i dettagli della macchina. Inoltre, deve essere più performante di una virtual machine.



La storia dei container

Ripercorriamo brevemente la storia dei container, tra funzionalità del Linux kernel, primi progetti open source e prime aziende che hanno visto la potenzialità di questa tecnologia, fino alla loro ampia diffusione grazie a Docker.

1979: Unix V7

In quest'anno viene rilasciata una feature del Linux kernel, *chroot*, che permette di cambiare la root directory di un processo. Questo risultato è solo l'inizio verso l'isolamento dei processi, meccanismo necessario nei container odierni.

2001: Linux VServer

Nasce quello che viene definito *meccanismo jail*, una sorta di virtualizzazione a livello di sistema operativo. Linux VServer è tra i primi software di questo tipo che permette di isolare a partizionare risorse su una macchina

2002: Namespaces

Il kernel Linux rilascia una nuova funzionalità: i *namespaces*. I namespaces permettono di partizionare risorse hardware tra un set processi, limitando la loro visibilità sul resto del sistema.

2001-2007:

Molte aziende iniziano ad investire in questa tecnologia: Solaris container (Oracle), Open VZ

2007: Cgroups

Altra funzionalità del kernel Linux. Infatti, il *cgroup* (abbreviazione di control groups), è una funzionalità del kernel Linux che limita le risorse (CPU, memoria, disk I/O, network ecc.) dei processi.

2008: LXC

LXC (Linux Containers) fu il primo esempio di container engine moderno. Sfrutta infatti feature del kernel Linux utilizzate dai più recenti container engine (cgroups, namespaces)

2013: Docker

Partendo da LXC, nel 2013 nasce Docker come software open source per eseguire container. Da allora, il mondo dei container è destinato a cambiare.

Nel corso del tempo ha implementato il proprio container manager utilizzando librerie proprie (libcontainer).

2013 - Today

Dal 2013, le tecnologie si sono consolidate, sono stati creati standard (OCI), orchestratori di container (Kubernetes, Docker Swarm) e diverse alternative (micro Virtual Machines)

Docker e la popolarità dei container

Molto spesso i termini *Docker* e *container* vengono confusi, ma come mai?
Semplicemente perchè è grazie a Docker che i container sono diventati così popolari.

Docker è un insieme di prodotti PaaS (Platform as as Service) che permettono lo sviluppo e il delivery di software in container.

dotCloud, l'attuale Docker Inc, rilasciò Docker per agevolare lo sviluppo software creando degli "standard box". Partendo proprio dalle funzionalità del kernel Linux viste precedentemente Docker volle agevolare l'utilizzo di queste funzionalità a basso livello, fornendo interfacce dal facile utilizzo.

Docker rese open-source tre aspetti chiave che facilitarono l'utilizzo dei container, favorendone quindi un ampio utilizzo:

- Uno standard per il formato dei container
- Strumenti per pacchettizzare (build) i container
- Strumenti per lanciare ed eseguirei container

Come abbiamo già visto, l'immagine di un container non è altro che un pacchetto software autocontenuto con codice sorgente, librerie e relative dipendenze.

Una volta create, queste immagini sono standard, statiche, non cambiano più (proprio come il contenuto dei container che vengono spediti da una parte all'altra del mondo). Non è scopo di questo articolo scendere nel dettaglio dell'immagine di Docker, possiamo però dire che il container, una volta che viene eseguito, potrà scrivere su un layer *writable* che è separato dal layer in cui è salvata l'immagine del nostro container. Di per se, infatti, i dati di un container sono effimeri.

Ma come possiamo creare l'immagine di un container con Docker? Semplice, scrivendo una "ricetta", chiamata *Dockerfile*: un file testuale che consiste in istruzioni ben precise su come pacchettizzare il nostro software standalone. Gli sviluppatori, guardando il Dockerfile, possono concludere a colpo d'occhio quale sarà il contenuto finale dell'immagine e averne pieno controllo.

Di seguito è riportato un esempio di Dockerfile che usa Ubuntu 18 come immagine di partenza (FROM), copia il contenuto della directory in cui si trova il Dockerfile nella cartella *app* (COPY), esegue un comando durante la creazione dell'immagine (RUN) e definisce il comando che dovrà essere eseguito allo startup del container (CMD)

```
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

Semplice, vero?

Ma le funzionalità di Docker non finiscono qui. Una volta creata l'immagine, infatti, gli sviluppatori o altri membri del team possono scambiarla tra di loro ed eseguirla sul proprio computer, on-premise, o sul cloud senza dover installare altro oltre all'engine di Docker, usando una comoda, e di facile utilizzo, Command Line Interface (CLI). Il container runtime di Docker non fa altro che leggere il contenuto dell'immagine ed eseguirlo. Grazie ai Linux namespaces, ogni container verrà eseguito in un ambiente separato dagli altri.

Avete visto i vantaggi di Docker? Potete capire il perchè della sua popolarità?

Docker non ha stravolto il concetto di container (portabilità, migliori prestazioni rispetto a tradizionali VM, possibilità di modularizzare applicazioni monoliti, offrire maggiore scalabilità), ma è stata la prima azienda ad offrire gli strumenti giusti per il delivery del software, velocizzando il processo. Non a caso, questi sono anche gli anni della rivoluzione DevOps (agilità, flessibilità, scalabilità nel delivery di un software)

Da Docker allo standard OCI (Open Container Initiative)

Dalle prime release di Docker, una grande quantità di persone iniziò a utilizzare i container come unità standard per il delivery del proprio software. Anche le aziende iniziarono ad utilizzare Docker, e il suo team non riuscì sempre a rispondere a tutte le esigenze tecniche e di business che il mercato richiedeva. In risposta a questo, dalla community sono nati diversi runtime, con diverse implementazioni e capabilities, unitamente a nuovi tool per creare e lanciare container.

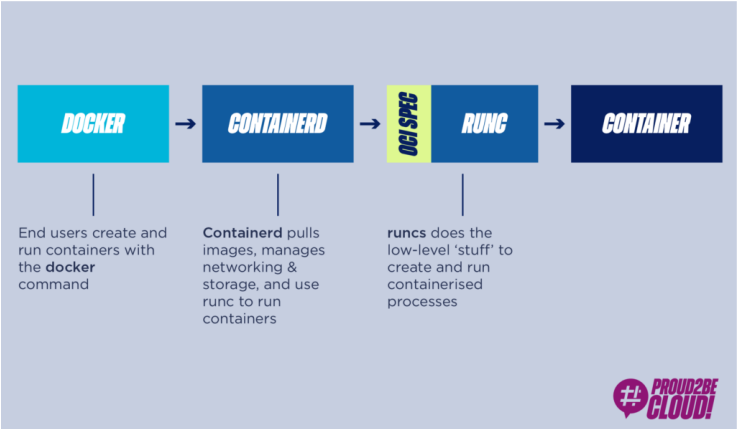
Per essere sicuri che diversi container runtime potessero lanciare immagini prodotti da diversi tool, Docker e altri membri della community fondarono nel 2015 la Open Container Initiative (OCI) per definire uno standard attorno al mondo dei container.

Le immagini originali d Docker sono quindi diventate l'OCI image specifications.

Fornita un'immagine conforme allo standard OCI, qualsiasi container runtime che supporti l'OCI runtime può leggere il contenuto di un'immagine ed eseguirlo in un ambiente isolato.

Docker non ha donato solo lo standard di creazioni dell'immagine, ma anche il suo container runtime, runc, e il demone containerd, creando così un ecosistema completamente modulare, come riportato di seguito:

- **Docker**: La suite di Docker include la CLI per utilizzare i comandi con cui creare e gestire i container
- **Containerd**: un demone che gestisce il lifecycle dei container. Salva e scarica le immagini, gestisce memoria e networking
- **Runc**: il low-level container runtime, colui che effettivamente crea ed esegue i container. E' scritto in Go



OCI è quindi un gruppo di tech companies che mantiene una specifica per l'immagine dei container e su come essi debbano essere eseguiti. Nasce dalla volontà di poter scegliere tra diversi runtime che sono conformi alle stesse specifiche, avendo implementazioni a basso livello diverse

E Windows OS?

Fino ad ora abbiamo parlato di container strettamente legati al mondo Linux. Ma in tutto questo, Windows dove si posiziona? Proviamo a rispondere.

Su windows, hai sostanzialmente due modi per eseguire i container:

- Direttamente sulla macchina host, con quello che Microsoft chiama Windows Server Container. Proprio come i container descritti fino ad ora, infatti, condividono un kernel unico.
- Utilizzando una macchina virtuale gestita da Hyper-V, un sistema di hypervisor di Microsoft. Ogni Hyper-v contianer è una macchina virtuale totalmente isolata, ciascuna con un proprio kernel

Nel mondo Linux, Docker è quasi completamente open source e supporta tutte le distro Linux. Per poter utilizzare i container anche su questo sistema operativo, Docker e Windows lavorano a stretto contratto per portare e mantere i container su questo sistema operativo.

Container Non-OCI e alternative

Ma una domanda sorge spontanea: esistono alternative allo standard OCI? Certamente! Vediamo alcuni esempi:

- **Crun**: Un container runtime scritto in C (a differenza di *runc*, che è scritto in Go).
- **Kata-runtime**: nasce dal progetto katacontainers. Rispecchia in parte lo standard OCI, con la differenza che ad essere eseguiti non sono dei container, ma delle micro Virtual Machines (le tratteremo tra poco!).
- **gVisor**: progetto opensource di Google per creare container con un proprio Kernel. Implementano solamente il runtime container OCI (chiamato *runsc*). Possiamo considerarli come una via di mezzo tra container (in termini di Linux namespaces) e micro VMs. Le applicazioni che girano dentro il sistema gVisor puntano ad essere più sicure, partendo dal fatto che dialogano di meno con il kernel Linux sottostante, riducendo quindi la superficie di attacco da workload indesiderati.
- **Firecracker**: un runtime ottimizzato per workload serverless. E' la tecnologia utilizzata da AWS come runtime di Lambda e Fargate. Firecracker fa girare applicazioni containerizzate dentro una micro VM. Leggere micro VMs ottimizzate per singole applicazioni. Se siete curiosi, il progetto è disponibile su Github!

Tra le alternative non conformi allo standard OCI, esistono quindi soluzioni diverse ai container, parliamo soprattutto di micro VM. Stiamo facendo un salto nel passato quindi? Assolutamente no! Generalizzando, possiamo dire che le micro VM sono delle VM leggerissime, con una propria versione ridotta del kernel, che offrono livelli di isolamento e sicurezza dei processi maggiori rispetto ai container. Si spiega perché, ad esempio, AWS abbia scelto questa modalità per offrire maggiore sicurezza ai proprio clienti che scelgono potenza computazionale di tipo serverless (Lambda e Fargate).

Container e Cloud Computing

I container vengono ormai utilizzati ampiamente come strumento di delivery del software, ma è possibile farlo anche sul cloud? Certamente!

AWS offre servizi come ECS (Elastic Container Service), già discusso in molti articoli sul nostro blog, e EKS (Elastic Kubernetes Service) come orchestratori di container. Nulla vieta, inoltre, di poter orchestratore in autonomia i proprio container su macchine EC2, magari in autoscaling, ma dove sarebbe il vantaggio?

Personalmente, in architetture basate su container, vedo questi ultimi come l'unità infrastrutturale base da gestire, dimenticando quindi la manutenzione dei server sottostanti. Se ci pensiamo bene, è proprio la filosofia che sta dietro ai container, ovvero avere degli elementi software autocontenuti, indipendenti dall'host su cui girano (in termini di hardware e OS). Per questo motivo, quando uso ECS lo scelgo in modalità Fargate: costruiamo, gestiamo e manteniamo container, non server!

Per concludere

In questo articolo abbiamo ripercorso la storia dei container, da quando sono nati fino alle loro applicazioni odierne. Abbiamo trattato i vantaggi e le differenze principali rispetto ad una classica architettura basata su macchine virtuali.

Siete interessati ad approfondire l'argomento? Fatecelo sapere nei commenti!

PS: Sapevate che dall'ultimo AWS re:Invent è possibile eseguire i propri container su Lambda?

Alessandro Bertini

DevOps Engineer @ beSharp, mi occupo di sviluppo software Cloud-native, fortemente orientato al paradigma Serverless!Appassionato di giochi da tavolo e videogame (come ogni buon smanettone!)
