

CONTAINER E ORCHESTRAZIONE DI CONTAINER



(CON DOCKER E KUBERNETES)

Francesco Patanè - Corso di Laurea Magistrale in Ingegneria Informatica – Big Data

Matricola: 530HHHINGINFOR

SOMMARIO

INTRODUZIONE	3
VIRTUALIZZAZIONE CON DOCKER.....	5
DOCKER IN AZIONE	9
ORCHESTRAZIONE DI CONTAINER CON KUBERNETES	18
KUBERNETES IN AZIONE	22
CONCLUSIONI.....	29
BIBLIOGRAFIA.....	30

INTRODUZIONE

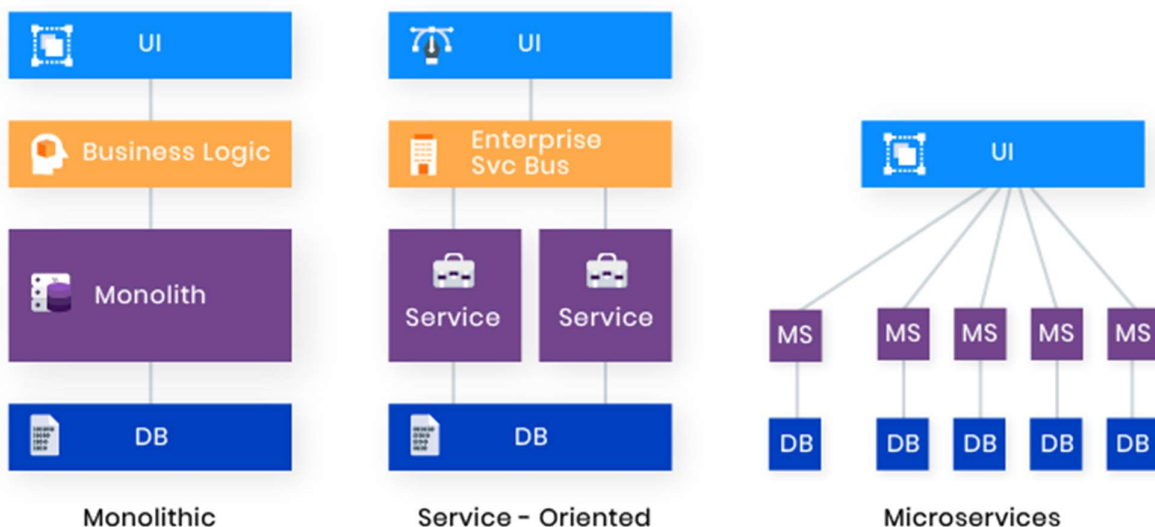
Come sappiamo, nel tempo recente la tecnologia, in particolare la sfera relativa all'informazione e alla comunicazione, si è diffusa in modo capillare in tutti gli aspetti della nostra vita e società e il suo utilizzo è diventato appannaggio delle masse, in un processo senza prospettive di rallentamento nel futuro concepibile.

Questo cambiamento è causato dalla rapida evoluzione del contesto tecnologico, e allo stesso tempo lo alimenta. Di innovazione in innovazione, si ottengono nuove possibilità e capacità, necessità vengono soddisfatte lasciando il passo a richieste e problematiche ancor più complesse.

Ne sono ben consapevoli i professionisti del settore IT. Negli ultimi anni sono state sviluppati o hanno preso piede una miriade di strumenti, tecnologie, linguaggi, approcci architetturali... l'aumento della complessità e le evoluzioni volte a contenerla sono particolarmente evidenti nello sviluppo software.

Nel momento in cui le necessità dei contesti di business hanno portato gli applicativi monolitici alla massa critica oltre la quale manutenibilità, scalabilità ed evoluzione non erano più ottenibili in maniera efficiente, si è passati a un approccio modulare allo sviluppo software, strutturando le applicazioni in moduli differenti, più facili da gestire e riutilizzare. Da questo concetto sono sorte le architetture Service Oriented, il cui esempio più famoso sono probabilmente gli applicativi Java Enterprise eseguiti all'interno di un application server, ancora largamente rilevanti e diffuse.

Anche questo approccio non è stato sufficiente. L'innovazione dell'architettura software è giunta all'approccio detto "a micro-servizi": Da un'applicazione composta da vari moduli, a un sistema composto da varie applicazioni che collaborano tra loro. Questa è la più recente tipologia architetturale affermata in modo stabile (in questa trattazione non si farà riferimento al paradigma serverless, nel quale la novità risiede nei modelli di esecuzione offerti tramite cloud computing).



Con questo tipo di architettura si ottengono importanti vantaggi: il codice viene distribuito in applicazioni più piccole, più semplici da gestire separatamente; la scalabilità diventa più efficiente potendo essere incentrata sui servizi che più ne hanno bisogno; si ottiene un disaccoppiamento molto maggiore tra le varie componenti.

Tuttavia questi benefici vengono ottenuti senza alcuna riduzione della complessità: si introducono problematiche non banali relative in particolare alla comunicazione e coordinazione tra i vari micro-servizi, al loro monitoraggio e messa in esercizio.

Per far fronte a questi aspetti, si è cercato di spostare parte della complessità dal livello software a quello infrastrutturale. Proprio qui entrano in gioco l'utilizzo dei container, ovvero delle virtualizzazioni interne al sistema operativo, e la loro gestione automatica, detta orchestrazione, che si stanno dimostrando un supporto fondamentale all'architettura a micro-servizi e più in generale uno dei capisaldi delle ultime innovazioni nel campo dello sviluppo software ed erogazione di servizi ad esso collegati.

In questa trattazione verranno illustrati i concetti di container e loro orchestrazione, quelle che al momento presente sono le principali tecnologie a tal riguardo, ovvero Docker e Kubernetes, e verranno mostrati degli esempi di utilizzo nello sviluppo applicativo.

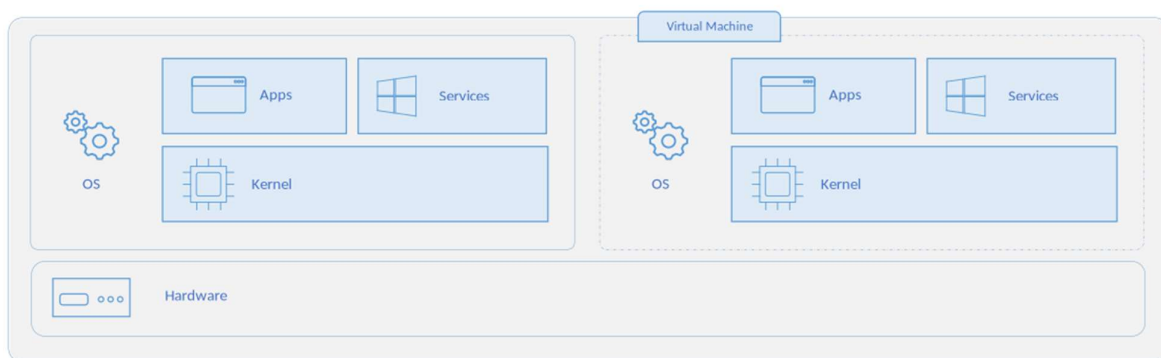
VIRTUALIZZAZIONE CON DOCKER

Concetto di container

Come già delineato nell'introduzione, l'architettura a micro-servizi prevede la creazione di sistemi complessi il cui funzionamento consiste nell'interazione di svariate applicazioni, ciascuna incentrata sul fornire delle funzionalità ben delineate relative a uno specifico dominio applicativo e idealmente caratterizzata da un'alta coesione e uno scarso accoppiamento con le altre applicazioni del sistema (i cosiddetti micro-servizi, per l'appunto, anche se l'aggettivo micro non è da considerarsi in modo letterale).

Volendo mettere in pratica i principi di questo tipo di architettura ci si scontra con la complessità inerente alla gestione di un così elevato numero di individuali applicazioni, che non si limitano al puro software di business, ma include anche eventuali elementi complementari come i DBMS.

L'approccio tradizionale consiste nell'eseguire il software, monolitico o SOA che sia, internamente a una macchina virtuale, ovvero una replica virtuale di un intero sistema operativo. Tuttavia, quando si parla di svariate applicazioni, ognuna con un proprio ciclo di vita, che interagiscono tra loro, la macchina virtuale si rivela uno strumento non sufficientemente flessibile. Ogni servizio includerebbe non solo le funzionalità del dominio di business, ma un intero sistema operativo. Non sarebbe possibile effettuare operazioni come la terminazione e la creazione di una nuova virtual machine in modo sufficientemente rapido. Inoltre il funzionamento del software potrebbe riscontrare delle differenze o anomalie da una macchina virtuale all'altra.

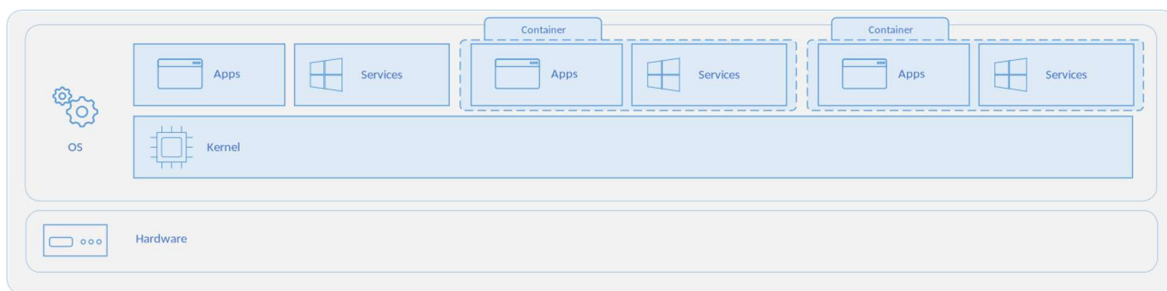


(Architettura macchina virtuale)

Quelli elencati sono alcune delle inadeguatezze delle macchine virtuali quando le si adoperano in un contesto a micro-servizi. A queste lacune si è colmato adottando le tecnologie di virtualizzazione tramite container.

Il concetto di container è figlio del paradigma di “virtualizzazione a livello del sistema operativo”, in cui il kernel permette l’esistenza di multiple istanze isolate di spazi utente dette, per l’appunto, container.

Per comprendere la differenza tra un container e un normale sistema operativo è utile assumere il punto di vista del software. Un programma in esecuzione all’interno di un sistema operativo ordinario ha visibilità su tutte le sue risorse. Al contrario, per un programma in esecuzione all’interno di un container, le uniche risorse visibili sono quelle assegnate al container stesso. Si ottiene così un isolamento internamente al sistema operativo ospite.

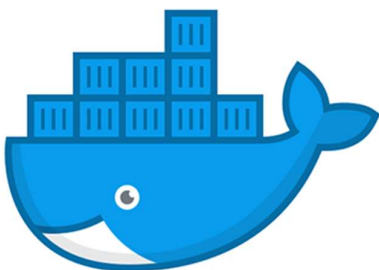


(Architettura container)

Si intuisce già come la virtualizzazione in container sia ortogonale ai moderni approcci di sviluppo software a micro-servizi: un contenitore isolato all’interno del quale è possibile eseguire applicazioni ben si presta al concetto di realizzare multiple applicazioni ognuna con un ben definito dominio applicativo.

Cominciamo ora a considerare l’aspetto puramente tecnologico esaminando il primo dei due strumenti oggetto di questo documento: docker.

Docker

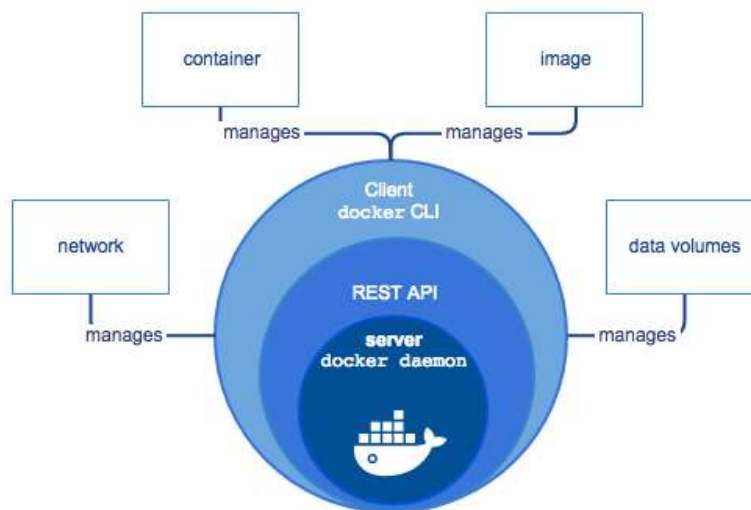


Docker è una piattaforma open source per lo sviluppo, la distribuzione e l’esecuzione di applicazioni contenute in container, sviluppata da Docker Inc. a partire dal 2010 e rilasciato per la prima volta nel 2013. Permette la gestione dell’infrastruttura oltre che delle applicazioni e consente una notevole diminuzione del tempo che intercorre tra la scrittura del codice e la sua messa in produzione.

Tramite Docker è possibile eseguire svariati container in uno stesso kernel, e vengono messi a disposizione strumenti per una loro esecuzione orchestrata.

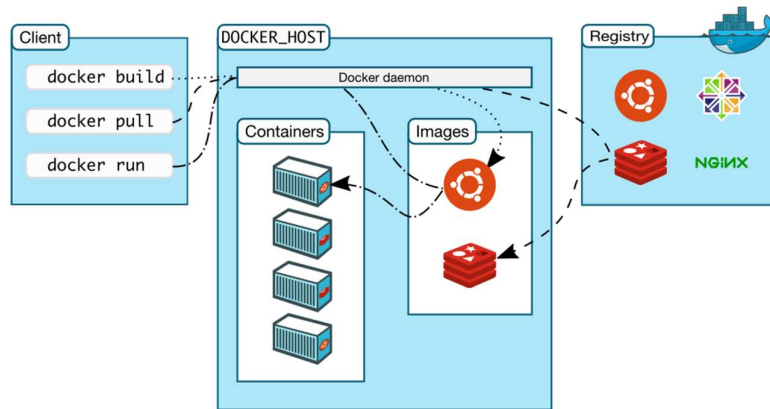
Il Docker Engine è strutturato come un'applicazione client-server con i seguenti componenti principali:

- Un server costituito da un processo demone in background.
- Un'API REST che specifica le interfacce attraverso le quali i programmi possono comunicare con il demone.
- Un client con interfaccia a riga di comando (CLI).



Il demone crea e gestisce varie tipologie di oggetti docker, tra i quali container, immagini, reti, servizi, volumi. Di seguito una breve descrizione dei principali:

- **Immagine:** template read-only per la creazione di contenitori. Contenute in un registro.
- **Container:** istanza eseguibile di una immagine.
- **Service:** definisce lo stato desiderato di un servizio in ogni momento, e come può essere acceduto (es: tramite load balancing).
- **Network:** rete virtuale a cui i container possono essere assegnati.
- **Volume:** oggetto che indica una porzione di file system in cui i dati prodotti da un determinato container vengono persistiti fisicamente.



Vediamo ora alcuni vantaggi nell'adozione di container docker rispetto alle macchine virtuali, relativamente allo sviluppo software.

- **Leggerezza:** un container non necessita di inglobare un sistema operativo dedicato.
- **Isolamento:** un container è un elemento completamente isolato dal sistema ospite, assicurando un funzionamento coerente indipendentemente dalla piattaforma.
- **Distribuzione semplificata:** i container permettono di distribuire applicazioni in modo unitario, minimizzando gli impatti di una singola distribuzione rispetto al sistema nel suo insieme.
- **Disponibilità rapida:** la leggerezza dei container ne riduce notevolmente i tempi di avvio, rendendo routine le operazioni di spegnimento e avvio.
- **Portabilità:** forniscono un modo rapido per la distribuzione in vari ambienti minimizzando le configurazioni necessarie.
- **Granularità:** la capacità di inglobare al loro interno tanto un'intera applicazione quanto un singolo componente permette una suddivisione logica dei vari componenti dell'architettura.

Nel prossimo capitolo verrà riportata un'applicazione pratica della tecnologia relativamente alla realizzazione di un banale software di esempio.

DOCKER IN AZIONE

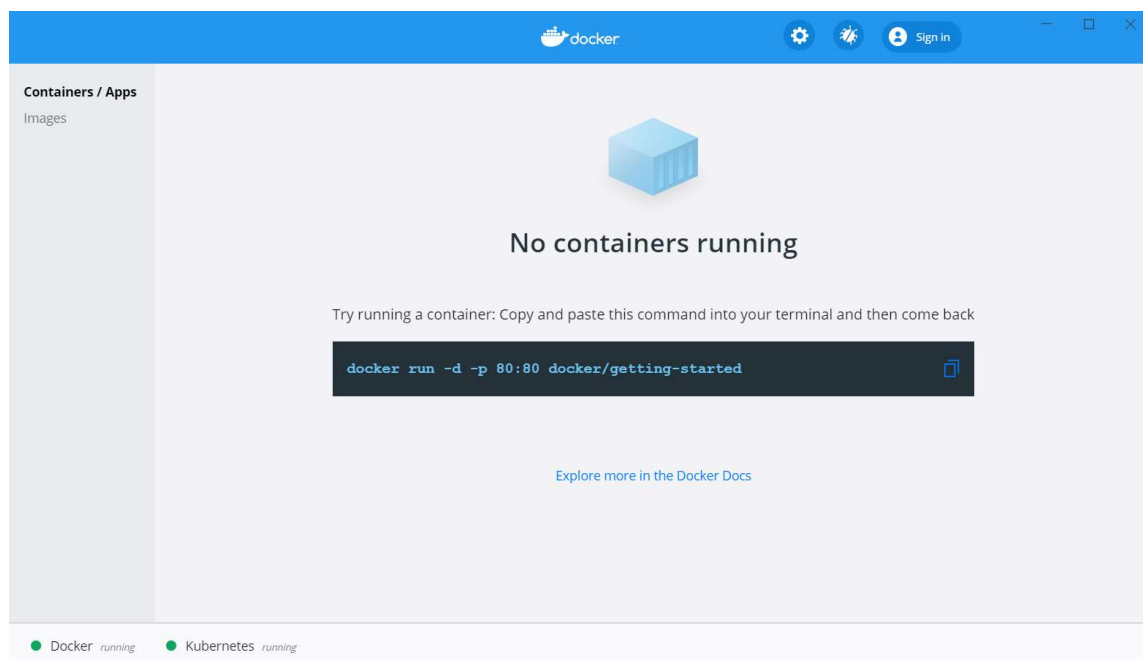
In questo capitolo verrà illustrato un semplice esempio pratico per dimostrare alcune delle caratteristiche di Docker relativamente in particolar modo allo sviluppo software. L'attività svolta è strutturata come segue:

1. Installazione di docker su una macchina Windows locale.
2. Creazione ed esecuzione di container contenente un DBMS PostgreSQL.
3. Connessione e azione sul DBMS esposto tramite container.
4. Creazione di una immagine contenente una semplice applicazione Spring Boot.
5. Eseguire l'applicazione Spring Boot in un container e farla comunicare con il DB.

Installazione

Sebbene Docker si accoppi più naturalmente con un sistema operativo Unix, ci sono vari modi di installarlo su una macchina Windows. Per il caso in esame si è scelto di adottare Docker Desktop (<https://docs.docker.com/docker-for-windows/install-windows-home/>), che non solo fornisce un ambiente completo per lo sviluppo di container, ma include anche la possibilità di eseguire un server Kubernetes, di cui faremo uso nella seconda parte della nostra trattazione. Oltre ad esso, sarà necessario installare anche WSL 2 (<https://docs.docker.com/docker-for-windows/wsl/>), ovvero Windows Subsystem for Linux 2, una estensione che permette di replicare il kernel Linux su una macchina Windows.

Una volta completata l'installazione, oltre alla CLI, sarà disponibile anche un'interfaccia grafica.



Al momento non viene visualizzato alcun container in esecuzione.

Esecuzione Dbms in container

Passiamo ora all'attivazione di una istanza PostgreSQL internamente a un container. Per svariate applicazioni esistono già delle immagini preimpostate, rese accessibili pubblicamente nel Docker Hub, un repository pubblico dove è possibile caricare o scaricare immagini. Postgres è una delle applicazioni per cui sono fornite delle immagini già pronte.

Di default, se si richiede a Docker di scaricare una nuova immagine, la cercherà su Docker Hub.



Come già accennato, il principale metodo di interazione con Docker è tramite linea di comando. È possibile eseguire comandi Docker tramite il prefisso `docker`. Con il comando

> `docker pull postgres`

Si richiede lo scaricamento dell'immagine postgres dal repository.

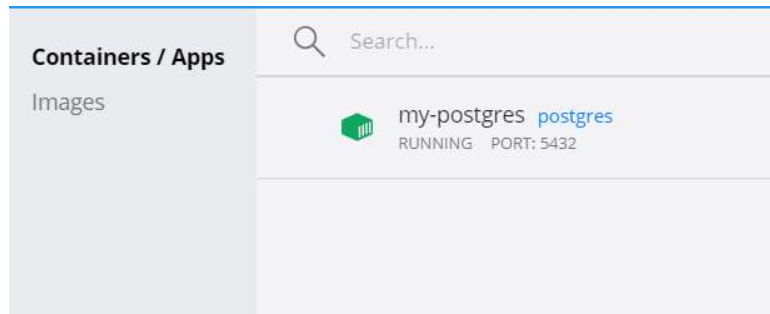
Ottenuta l'immagine, è necessario creare ed eseguire un contenitore a essa relativo. Con il comando

> `docker run -d -p 5432:5432 -name my-postgres -e POSTGRES_PASSWORD=postgres -v /var/lib/postgresql/data postgres`

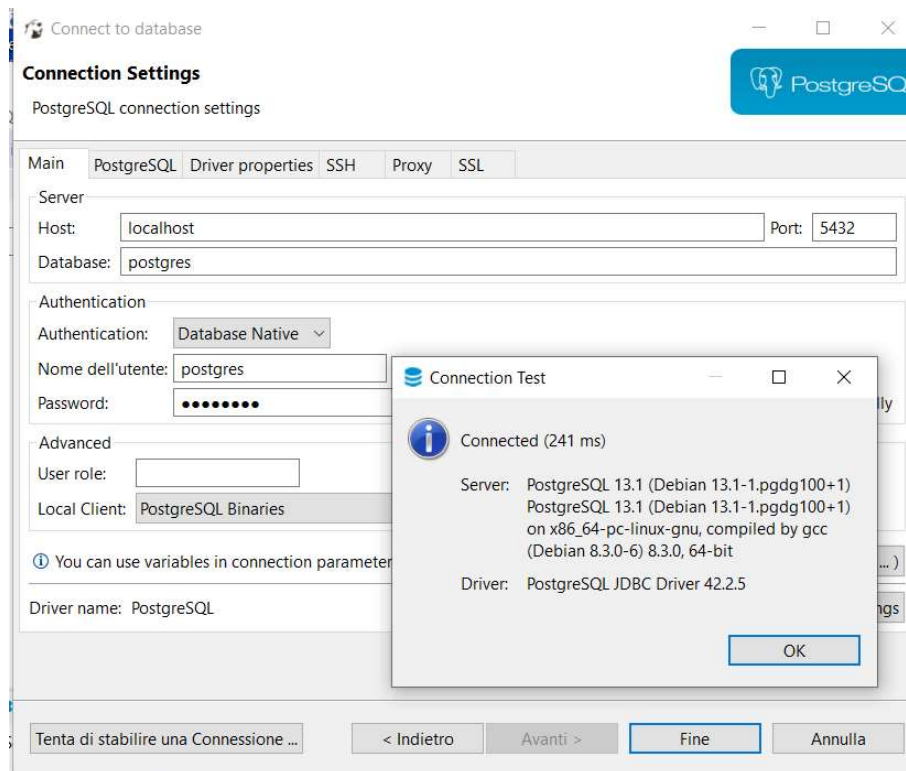
Viene avviato un nuovo container contenente un'immagine postgres. Esaminiamo i parametri:

- **run:** richiede la creazione e l'esecuzione di un processo container.
- **-d:** esegue il processo in background (detached mode).
- **-p 5432:5432:** mappa la porta 5432 interna al container con la porta 5432 del sistema host.
- **-name my-postgres:** assegna il nome my-postgres al nuovo container.
- **-e POSTGRES_PASSWORD=postgres:** imposta la password per la connessione al db.
- **-v /var/lib/postgresql/data:** imposta la persistenza fisica nel volume dal path indicato.

Controllando l'interfaccia grafica di docker-desktop, possiamo verificare la presenza del container appena avviato.



È ora possibile connettersi all'istanza di postgres come faremmo con un qualsiasi DBMS. Ad esempio, tramite un client come DBeaver.



Infine, creiamo una semplice tabella di esempio contenente dati utente.

usr_user				
	user_id	email	name	surname
1	1	mrossi@gmail.com	Mario	Rossi
2	2	sbianchi@gmail.com	Stefano	Bianchi

Esecuzione applicazione in container

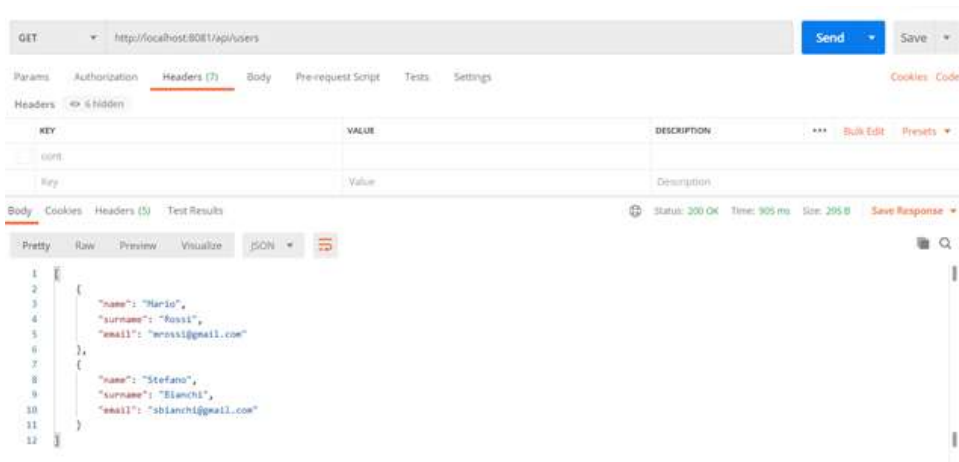
Fino a questo punto abbiamo utilizzato un'immagine fornita da terzi, ma come è possibile creare un'immagine di un software realizzato da noi, in modo da poterlo eseguire internamente a un container? Nei prossimi paragrafi sarà illustrato come creare un'immagine relativa a una semplice applicazione Java.

L'applicazione consisterà in un semplice servizio REST che legga i dati degli utenti inseriti nel database nel paragrafo precedente. Per questo esempio ci avvarremo del framework Spring Boot.

In questa trattazione non verranno illustrati i dettagli implementativi della banale applicazione Spring. È sufficiente considerare che, avviando in locale l'applicazione con le seguenti proprietà

```
server.port=8081
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=postgres
```

essa riesce a connettersi all'istanza Postgres containerizzata come ad un qualsiasi database. Di seguito il risultato della chiamata REST.



Approntata l'applicazione, vediamo come creare un'immagine a partire da essa per eseguirla in un contenitore.



Docker costruisce le immagini automaticamente, a partire dalle specifiche contenute nel *Dockerfile*, un file di testo contenente i comandi da eseguire per la costruzione di un'immagine, espressi secondo una precisa struttura.

Per la nostra applicazione il Docker file conterrà le seguenti informazioni:

```
FROM openjdk:8-jdk-alpine
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} kubedemo-0.0.1-SNAPSHOT.jar
ENTRYPOINT ["java", "-jar", "/kubedemo-0.0.1-SNAPSHOT.jar"]
```

Vediamo il significato delle varie istruzioni:

- **FROM:** specifica l'immagine base su cui la nostra viene costruita, nel nostro caso l'immagine alpine della jdk 8.
- **ARG JAR_FILE:** dichiara una variabile JAR_FILE che contiene il path del jar di cui verrà creata un'immagine. Nel nostro caso un file jar sotto la cartella /target.
- **COPY \${JAR_FILE}:** specifica come copiare il jar file nel filesystem del contenitore.
- **ENTRYPOINT:** specifica come eseguire l'applicazione.

Con il comando

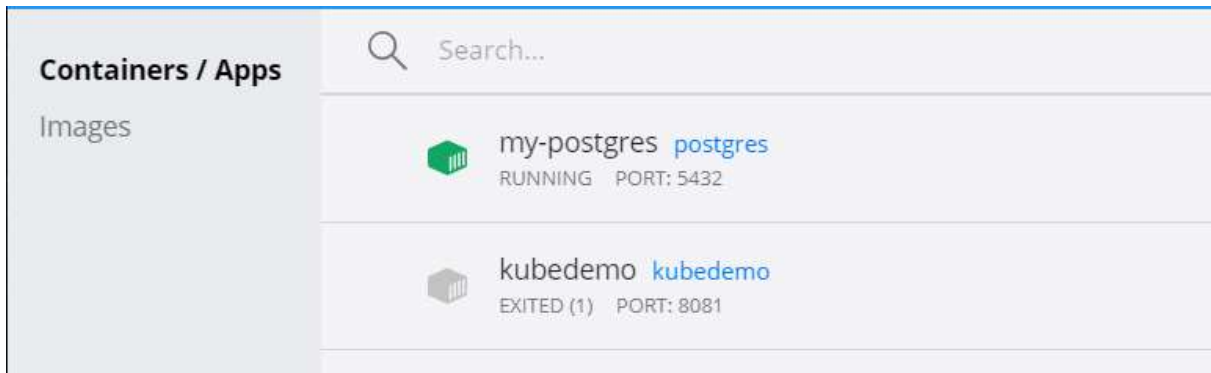
```
> docker build -t kubedemo .
```

Viene avviata la creazione dell'immagine e al termine questa viene aggiunta al registro.

Proviamo ora a lanciare un container della nuova immagine:

```
> docker run -p 8081:8081 --name kubedemo -d kubedemo
```

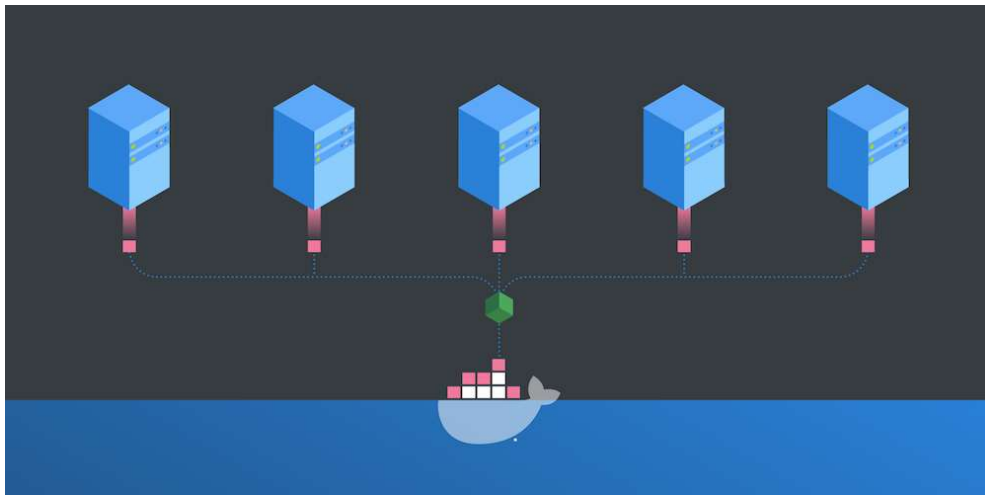
Il container è stato creato, tuttavia non viene attivato correttamente.



Questo perché l'applicazione non riesce a comunicare con il database. Venendo eseguita dall'interno di un container, questa non ha più il punto di vista dell'ambiente locale, per cui la connessione al db è presente su localhost:5432.

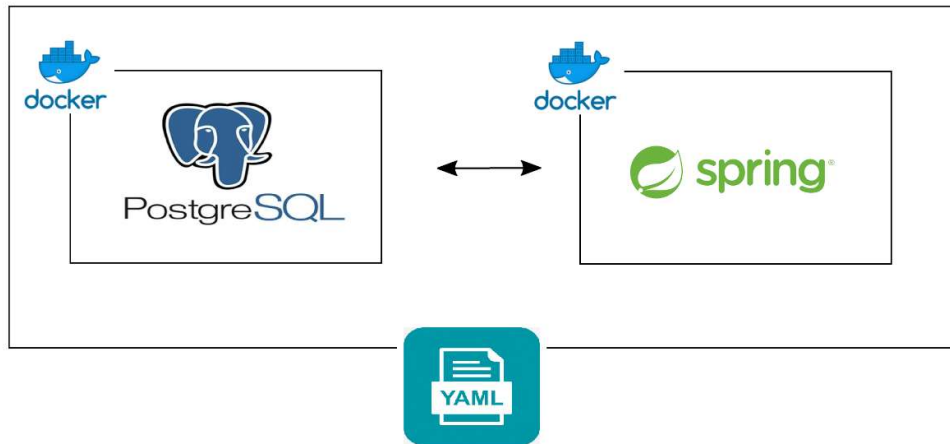
Comunicazione tra container - Docker Compose

Com'è dunque possibile far comunicare due container distinti? Entra in gioco un ulteriore strumento della suite Docker: Docker Compose.



Compose è uno strumento per la definizione ed esecuzione di applicazioni Docker multi-container. Permette la definizione delle varie componenti dell'applicazione mediante un file YAML, in modo da poter creare ed avviare tutte le componenti tramite un singolo comando.

Nel nostro caso i componenti in gioco sono due: il database e l'applicazione che legge i suoi dati. Creiamo quindi un file ***docker-compose.yml*** che li rappresenti.



Il file yml avrà il seguente contenuto:

```

1  version: "3"
2  services:
3    my-postgres:
4      image: postgres:latest
5      network_mode: bridge
6      container_name: postgres-s
7      volumes:
8        - kubedb:/var/lib/docker/volumes/kubedb/_data
9      expose:
10       - 5432
11      ports:
12       - 5432:5432
13      environment:
14       - POSTGRES_PASSWORD=postgres
15       - POSTGRES_USER=postgres
16       - POSTGRES_DB=kubedb
17      restart: unless-stopped
18
19    kubedemo:
20      image: kubedemo:latest
21      network_mode: bridge
22      container_name: kubedemo
23      expose:
24       - 8081
25      ports:
26       - 8081:8081
27      restart: unless-stopped
28      depends_on:
29       - my-postgres
30      links:
31       - my-postgres
32  volumes:
33    kubedb:

```

Per brevità ometteremo l'analisi di ogni singolo tag della specifica yaml per Docker Compose, tuttavia è opportuno esaminare la definizione **network_mode**: bridge.

Con Docker è possibile creare reti container-to-container e container-to-host, gestite da diverse tipologie di driver appositi. All'avvio di Docker, viene creata una rete bridge di default, alla quale, se non specificato diversamente, vengono collegati i container. Questa rete consente ai contenitori a essa collegati di comunicare tra loro, allo stesso tempo isolandoli dai componenti esterni alla rete. In Compose, in particolare, ogni servizio può essere localizzato in base al suo nome host, che è identico al nome del container.

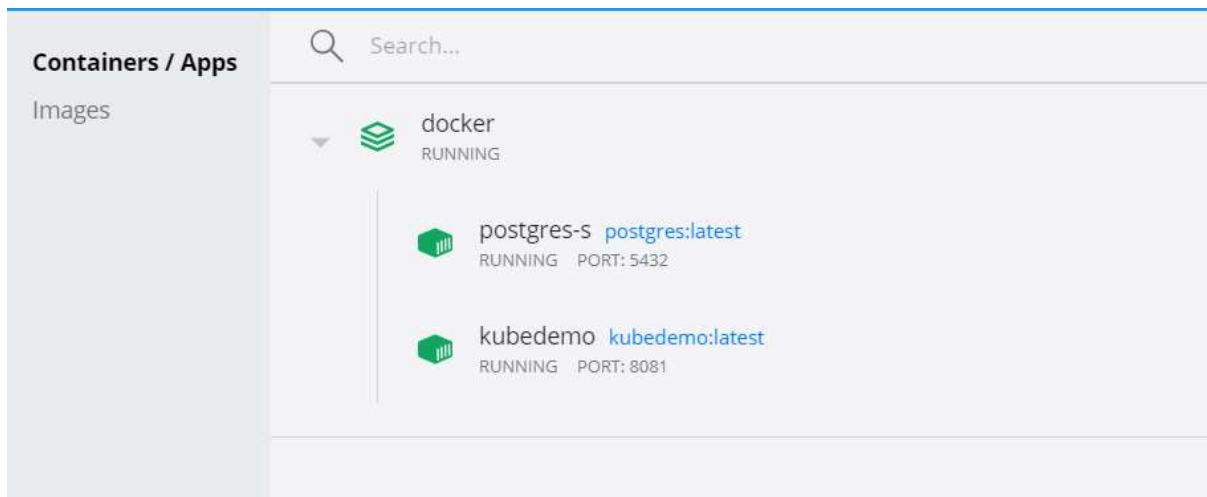
Pertanto, la stringa di connessione dell'applicazione Java sarà modificata come segue:

```
spring.datasource.url=jdbc:postgresql://postgres-s/postgres
```

Effettuata la modifica è necessario ricreare l'immagine come nel paragrafo precedente. Fatto questo, è possibile avviare lo script compose con il comando

> docker-compose up -d

Notiamo come nell'interfaccia di docker-desktop i nuovi container appaiano raggruppati.



Effettuando nuovamente la chiamata REST del paragrafo precedente, vengono nuovamente restituiti i dati presenti nel database.

Valutazione

Con questi semplici esempi di utilizzo pratico abbiamo potuto evidenziare le caratteristiche e i vantaggi della tecnologia Docker esposti nel capitolo precedente: rapidità, leggerezza, facilità di

utilizzo. Sebbene questo fosse un esempio minimale, è stato reso evidente come si presti alla separazione dei vari componenti del sistema informativo nelle sue componenti di dominio applicativo richiesta dallo sviluppo software in architettura a microservizi.

Vale però la pena di soffermarsi brevemente anche sugli svantaggi e gli aspetti più delicati che l'adozione di questa tecnologia porta con sé.

Già da questo piccolo esempio si intuisce quanto la questione della persistenza dei dati sia un fattore estremamente delicato. In un contesto di completa virtualizzazione e containerizzazione, l'utilizzatore deve porre estrema cura nell'assegnazione dei vari volumi a tutti i servizi che necessitino di persistenza dei dati e nel predisporre delle procedure di backup idonee.

È inoltre molto più facile, rispetto ad un contesto che non fa uso di contenitori, che dei dati vengano persi inavvertitamente, anche vista la semplicità con cui immagini e contenitori possono essere creati e cancellati, in particolare nel caso degli script Docker Compose, dove confondere il comando di arresto con quello di cancellazione avrebbe effetto immediato su tutti i componenti definiti nel file yml, volumi inclusi. La pluralità dei comandi di Docker, oltre a creare a volte confusione, non aiuta in questo ambito, essendo presenti vari modi di avviare, arrestare, cancellare container e immagini (ad esempio *docker run* crea un nuovo contenitore se non esistente e lo esegue, mentre *docker compose down* non solo arresta, ma rimuove i componenti di uno script compose, mentre *docker compose stop* si limiterebbe ad arrestarli).

Vengono inoltre mantenuti le classiche criticità dell'architettura a micro-servizi, come ad esempio la gestione dei log, e la comunicazione tra i vari servizi. Inoltre, proprio come il paradigma a micro-servizi, quando applicata a dei casi d'uso troppo semplici, come casi d'uso in cui il sistema informatico è soddisfatto da una singola applicazione (in sostanza, in cui i domini applicativi sono gli stessi dell'esempio pratico di questa trattazione: applicazione più database), non può esprimere nessuno dei suoi punti di forza ma anzi rischia di complicare inutilmente il processo di sviluppo.

Infine Docker non offre strumenti che forniscano funzionalità mirate per orchestrazione, scalabilità e gestione del carico, per le quali è necessaria l'adozione di ulteriori tecnologie specializzate. Inoltre, non permette una gestione distribuita su cluster, limitandosi a un singolo host. Al momento le soluzioni più affermate per colmare queste mancanze sono Docker Swarm e soprattutto Kubernetes. Nei prossimi capitoli esamineremo la seconda.

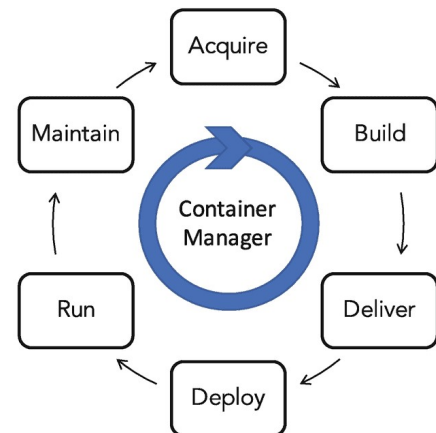
ORCHESTRAZIONE DI CONTAINER CON KUBERNETES

Concetto di orchestrazione

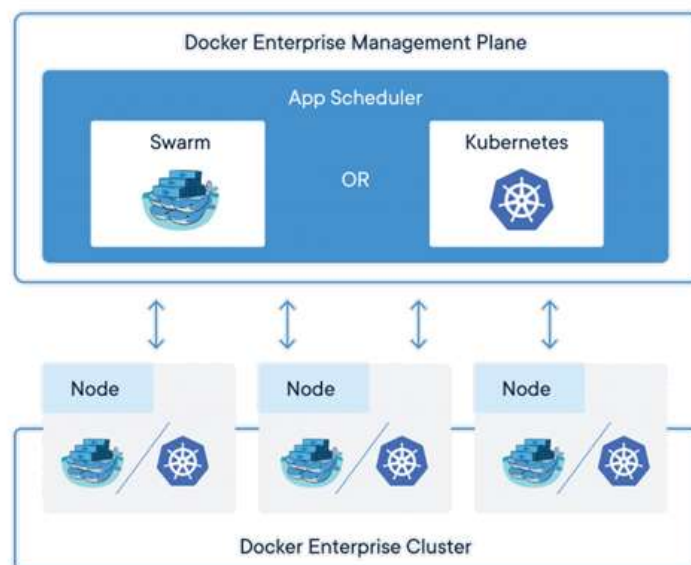
Quando i container vengono impiegati in sistemi di produzione complessi ed estesi, il loro numero può raggiungere cifre considerevoli da svariate dozzine fino anche, in casi estremi, migliaia. Le varie componenti potrebbero anche essere distribuite in un cluster comprendente vari host distinti. In questo contesto, i vari container vanno costantemente gestiti, distribuiti secondo richiesta, monitorati e connessi tra loro.

In particolare, possiamo enumerare come segue le principali necessità:

- Integrare e orchestrare i vari contenitori.
- Scalare in base alla richiesta.
- Rendere le varie componenti tolleranti ai malfunzionamenti.
- Permettere le comunicazioni nel cluster.
- Distribuire il carico.



Va da sé che una gestione manuale in uno scenario simile è impensabile. Eppure la capacità di ottemperare a queste richieste è alla base dei sistemi di cloud computing e dei loro relativi modelli di business, che tanta rilevanza hanno assunto negli anni recenti. È qui che entra in gioco il concetto di orchestrazione di container, con le tecnologie correlate.



Definiamo orchestrazione dei container l'automazione dei processi di distribuzione, gestione, scalabilità e connettività dei container.

L'orchestrazione dei container è applicabile a qualsiasi ambiente in cui si utilizzano i container, e risulta utile per la distribuzione della stessa applicazione su più ambienti, senza doverla riprogettare. Inserendo i microservizi nei container, è possibile gestire con semplicità i servizi, incluse l'archiviazione, le reti e la sicurezza.

I container offrono alle applicazioni basate su microservizi un'unità di distribuzione dell'applicazione ideale e un ambiente di esecuzione autosufficiente. Permettono di eseguire più parti di un'app in modo indipendente nei microservizi, sullo stesso hardware, con un controllo superiore sui singoli componenti e cicli di vita.

Scegliere l'orchestrazione per il ciclo di vita dei container aiuta inoltre i team DevOps a gestire più facilmente i flussi di integrazione e distribuzione continui (CI/CD). In concomitanza con l'utilizzo delle API e delle metodologie DevOps, i microservizi containerizzati rappresentano la base delle applicazioni cloud native.

Gli strumenti utilizzati per l'orchestrazione dei container offrono un framework per la gestione delle architetture basate su container e microservizi su larga scala. Numerosi sono gli strumenti disponibili per la gestione del ciclo di vita dei container. Esistono varie soluzioni, ma in questa trattazione prenderemo in esame Kubernetes, la più diffusa.

Kubernetes

Kubernetes è uno strumento open source per l'orchestrazione dei container ideato e sviluppato in origine dagli ingegneri di Google. Nel 2015, Google ha ceduto il controllo del progetto Kubernetes alla Cloud Native Computing Foundation, di recente formazione.



L'orchestrazione di Kubernetes consente di creare servizi applicativi che si estendono su più container, programmare tali container in un cluster, gestirne la scalabilità e l'integrità nel tempo.

kubernetes

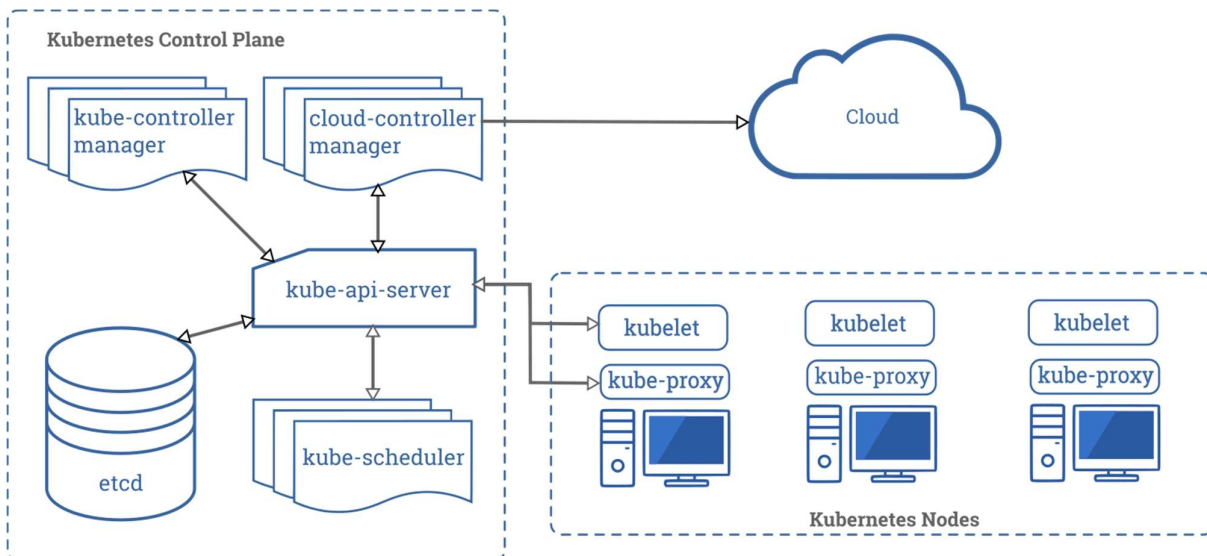
Consente di eliminare molti dei processi manuali previsti dal deployment e dalla scalabilità di applicazioni containerizzate, e di gestire con semplicità ed efficienza cluster di host, fisici e virtuali, su cui vengono eseguiti container Linux.

Più in generale, Kubernetes garantisce l'affidabilità necessaria ad adottare un'infrastruttura containerizzata negli ambienti di produzione.

I cluster possono espandersi su più host situati su cloud pubblici, privati o ibridi. Ecco perché Kubernetes è la piattaforma ideale per l'hosting di applicazioni cloud native caratterizzate da un'elevata scalabilità.

Infine, la piattaforma semplifica il bilanciamento e la portabilità dei carichi di lavoro, perché permette di spostare le applicazioni tra diversi ambienti senza doverle riprogettare.

Architetturalmente, da un punto di vista ad alto livello, un cluster Kubernetes implementa un paradigma master/slave, in cui un nodo master, detto pannello di controllo gestisce a libello globale i vari nodi cluster, detti Kubelets.



È proprio all'interno dei nodi Kubelet che vengono eseguiti i container.

Per molti versi, l'interazione con un sistema Kubernetes da parte dello sviluppatore software è simile a quella precedentemente vista con Docker. Anche Kubernetes mette a disposizione un'interfaccia a riga di comando e permette di definire la configurazione delle componenti del sistema tramite file yaml, sebbene con sintassi e specifiche differenti da quelle viste per Docker Compose.

Viene definito **carico** un'applicazione in esecuzione in Kubernetes. È fondamentale il concetto di **pod**, che comprende un insieme di uno o più container in esecuzione ed è la più piccola unità di esecuzione gestita. I container all'interno di un pod condividono le stesse risorse e rete locale.

Altri concetti sono quello di **volume**, analogo a quello visto in Docker per la gestione della persistenza, e di **service**, utilizzato per stabilire le modalità di gestione del traffico verso i pod. Infine anche il **networking** tra i vari componenti è gestito in modo simile a Docker, identificando le locazioni in base ai nomi dei service.

Vediamo più nel dettaglio le funzionalità principali offerte da Kubernetes:

- **Scoperta dei servizi e bilanciamento del carico:** Kubernetes può esporre i container mediante nomi DNS o indirizzi IP e, se necessario, mettere in atto logiche e servizi di distribuzione del carico, in modo che il servizio rimanga stabile.
- **Orchestrazione dello storage:** Kubernetes permette di montare automaticamente un sistema di archiviazione scelto, come per esempio storage locale, dischi forniti da cloud pubblici o altri ancora.
- **Rollout e rollback automatizzati:** definendo, per un container, lo stato desiderato, qualora il container dovesse trovarsi in uno stato diverso Kubernetes lo riporterà automaticamente alla configurazione indicata.
- **Ottimizzazione delle risorse:** per ogni container, è possibile definire delle indicazioni sulle quantità di CPU e memoria (RAM) da allocare ad esso. Kubernetes allocherà i container sui vari nodi per massimizzare l'uso delle risorse a disposizione.
- **Self-healing:** Kubernetes riavvia i container che bloccati, li sostituisce, li termina nel caso non rispondano agli health checks, e impedisce che giungano richieste ai container che non sono ancora pronti a riceverle.
- **Gestione di informazioni sensibili e della configurazione:** Kubernetes consente di memorizzare e gestire informazioni sensibili, come le password, i token OAuth e le chiavi SSH. È possibile distribuire e aggiornare le informazioni sensibili e la configurazione dell'applicazione senza dover ricostruire le immagini dei container e senza esporre le informazioni sensibili.

È evidente come Kubernetes sia una tecnologia estremamente complessa in grado di offrire un ampio spettro di funzionalità. Nel prossimo capitolo esporremo dei semplici esempi dell'utilizzo di Kubernetes. Ovviamente, su una singola macchina locale non è possibile dare una dimostrazione adeguata delle potenzialità dello strumento, ci limiteremo quindi a illustrare alcune semplici modalità di configurazione e l'interazione con i container Docker di cui si è discusso nei capitoli precedenti.

KUBERNETES IN AZIONE

Applicazione pratica

In questo capitolo illustreremo le funzionalità base di Kubernetes traslando il sistema d'esempio realizzato con Docker Compose su questa seconda piattaforma.

Non è necessario effettuare ulteriori installazioni, in quanto l'installazione di Docker desktop effettuata in precedenza include anche Kubernetes.

Riutilizzeremo le immagini create in precedenza per l'applicazione Java e il database Postgres, definendo per ognuno di essi un apposito pod tramite file di configurazioni yaml, rendendoli in grado di comunicare tra loro.

Cominciamo con la definizione del database, per cui prepariamo un file *postgres-pod.yaml*. Il file contiene la definizione di diversi elementi, che esamineremo uno a uno, commentandone i tratti salienti.

Il primo elemento che definiamo è il persistent-volume, che specifica le risorse fisiche da adibire alla persistenza dei dati.

```

1  apiVersion: v1
2  kind: PersistentVolume
3  metadata:
4    name: postgres-pv
5  spec:
6    capacity:
7      storage: 256Mi
8    volumeMode: Filesystem
9    accessModes:
10     - ReadWriteOnce
11    persistentVolumeReclaimPolicy: Delete
12    storageClassName: local-storage
13    local:
14      path: /run/desktop/mnt/host/c/kubernetes-volumes/postgres
15    nodeAffinity:
16      required:
17        nodeSelectorTerms:
18          - matchExpressions:
19            - key: kubernetes.io/hostname
20              operator: In
21              values:
22                - docker-desktop
23
```

Ad esso si combina un persistent-volume-claim, che rappresenta una richiesta di utilizzo del volume da parte di un altro elemento

```

26  apiVersion: v1
27  kind: PersistentVolumeClaim
28  metadata:
29    name: postgres-pvc
30  spec:
31    storageClassName: local-storage
32    accessModes:
33      - ReadWriteOnce
34    resources:
35      requests:
36        storage: 256Mi
37

```

I dati di connessione tramite un oggetto ConfigMap

```

55  apiVersion: v1
56  kind: ConfigMap
57  metadata:
58    name: postgres-config
59    labels:
60      app: postgres
61  data:
62    POSTGRES_DB: kubedb
63    POSTGRES_USER: postgres
64    POSTGRES_PASSWORD: postgres
65

```

Infine viene definito un oggetto deployment, che rappresenta l'applicazione in sé, da eseguire internamente a un container e che include gli elementi specificati precedentemente

```

67   apiVersion: apps/v1
68   kind: Deployment
69   metadata:
70     name: postgres
71   spec:
72     selector:
73       matchLabels:
74         app: postgres
75     template:
76       metadata:
77         labels:
78           app: postgres
79       spec:
80         containers:
81         - name: k-postgres
82           image: postgres
83           ports:
84             - containerPort: 5432
85           envFrom:
86             - configMapRef:
87               name: postgres-config
88           volumeMounts:
89             - name: storage
90               mountPath: /var/lib/postgresql/data
91         volumes:
92         - name: storage
93           persistentVolumeClaim:
94             claimName: postgres-pvc

```

Le specifiche per la connettività relativa al container che verrà creato sono definite in un service

```

39   apiVersion: v1
40   kind: Service
41   metadata:
42     name: postgres-s
43   spec:
44     type: LoadBalancer
45     selector:
46       app: postgres
47     ports:
48     - port: 5432
49       targetPort: 5432
50

```


Qui è necessaria una breve disquisizione sulle tipologie di service in Kubernetes. In questa configurazione è stato usato un tipo LoadBalancer, che come è facile intuire distribuisce il carico tra vari pod. In questo caso non fornirebbe benefici apprezzabili in quanto il pod relativo al container Postgres non è replicato. Tuttavia ha il vantaggio di effettuare in automatico le configurazioni necessarie per esporre il pod e renderlo raggiungibile dall'esterno del cluster, rendendo immediata la connessione dall'ambiente locale per la configurazione del database con i dati necessari al nostro esempio pratico.

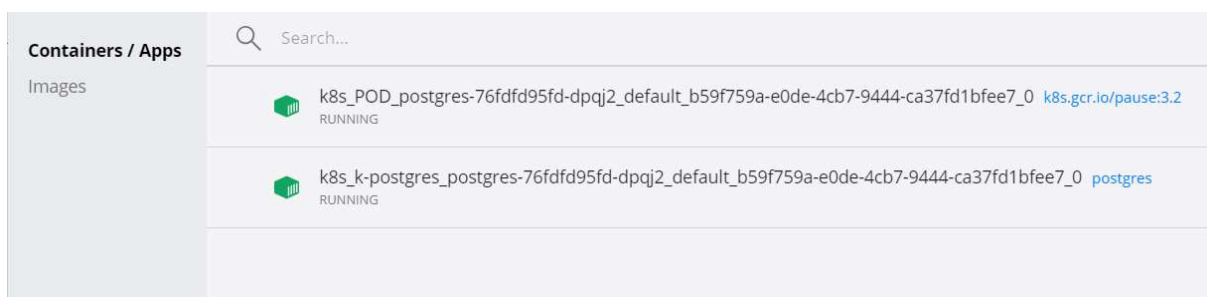
In generale Kubernetes gestisce tre tipi di service:

- **ClusterIP:** il tipo di default. Fornisce un punto di accesso per le altre applicazioni del cluster, ma non permette l'accesso dall'esterno.
- **NodePort:** permette un accesso esterno attraverso una porta specificata.
- **LoadBalancer:** principale metodo per esporre servizi all'esterno, crea automaticamente dei servizi NodePort e ClusterIp corrispondenti alle varie repliche del pod in modo da poter smistare il carico tra di esse.

Con questo abbiamo terminato la configurazione per eseguire il pod che conterrà il database. Possiamo richiedere a Kubernetes di elaborare il file tramite riga di comando, con il seguente comando:

> kubectl apply -f postgres-pod.yaml

Kubernetes creerà automaticamente tutti gli elementi specificati e eseguirà un container con Postgres. Controlliamo l'interfaccia grafica di Docker Desktop



Risultano in esecuzione due container: k8s.gcr.io/pause:3.2 è un container di sistema, che contiene il namespace relativo al pod appena creato, mentre postgres è il nostro container Postgres, che verrà nuovamente popolato con i dati di esempio.

Illustriamo ora la definizione dello yaml di configurazione dell'applicazione Java, kubedemo-pod.yaml.

Per prima cosa definiamo il Deployment.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    creationTimestamp: null
5    labels:
6      app: kubedemo
7    name: kubedemo
8  spec:
9    replicas: 2
10   selector:
11     matchLabels:
12       app: kubedemo
13   strategy: {}
14   template:
15     metadata:
16       creationTimestamp: null
17     labels:
18       app: kubedemo
19     spec:
20       containers:
21       - image: kubedemo
22         name: kubedemo
23         imagePullPolicy: Never
24         resources: {}
25         ports:
26         - containerPort: 8081
27   status: {}
```

Da notare la sezione spec con elemento replicas: 2. Ciò sta a indicare che verranno create due repliche di questo singolo pod. In questo modo il carico sarà bilanciato tra i due. Inoltre qualora uno dei pod dovesse subire dei malfunzionamenti, l'altro sarà in grado di soddisfare le richieste finché Kubernetes non lo avrà sostituito creando ed avviando automaticamente un container ex novo.

Specifichiamo la connettività con il service

```



32  apiVersion: v1
33  kind: Service
34  metadata:
35    name: kubedemo-lb
36  spec:
37    selector:
38      app: kubedemo
39    ports:
40      - port: 8081
41        targetPort: 8081
42    type: LoadBalancer

```

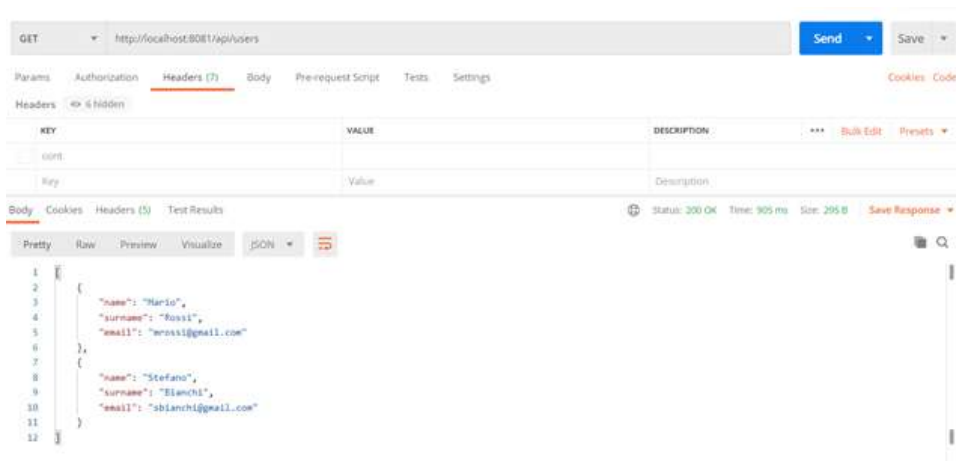
Richiediamo l'elaborazione del file yaml:

```
> kubectl apply -f kubedemo-pod.yaml
```

Dall'interfaccia grafica riscontriamo la presenza di due nuovi pod.

Containers / Apps	
Search...	
Images	 k8s_POD_postgres-76dfd95fd-dpqj2_default_b59f759a-e0de-4cb7-9444-ca37fd1bfee7_0 k8s.gcr.io/pause:3.2 <small>RUNNING</small>
	 k8s_k-postgres_postgres-76dfd95fd-dpqj2_default_b59f759a-e0de-4cb7-9444-ca37fd1bfee7_0 postgres <small>RUNNING</small>
	 k8s_POD_kubedemo-6bc6c58d5b-cdd99_default_9a70e453-ab08-4664-a878-b63ebc8b2fb7_0 k8s.gcr.io/pause:3.2 <small>RUNNING</small>
	 k8s_POD_kubedemo-6bc6c58d5b-q7fqw_default_793a954a-53b6-44a2-b31f-0ffc262fb050_0 k8s.gcr.io/pause:3.2 <small>RUNNING</small>
	 k8s_kubedemo_kubedemo-6bc6c58d5b-cdd99_default_9a70e453-ab08-4664-a878-b63ebc8b2fb7_0 sha256:cae8df5f9828cfd81c79785a10c1b3631ce74f0530ec123f97b00497c6d176f2 <small>RUNNING</small>
	 k8s_kubedemo_kubedemo-6bc6c58d5b-q7fqw_default_793a954a-53b6-44a2-b31f-0ffc262fb050_0 sha256:cae8df5f9828cfd81c79785a10c1b3631ce74f0530ec123f97b00497c6d176f2 <small>RUNNING</small>

Rieseguiamo la chiamata REST per verificare la correttezza del funzionamento.



Valutazione

In questo capitolo abbiamo avuto modo di dimostrare come sia possibile traslare un'applicazione Docker Compose su Kubernetes, in modo da poterne sfruttare le funzionalità. Particolarmente degne di nota sono le funzionalità di restart e load balancing automatiche, che ottemperano a due delle necessità più comuni e critiche.

Kubernetes porta però con sé una complessità da non sottovalutare, a cominciare dal richiedere un notevole impegno nella fase di apprendimento.

L'elevato numero di funzionalità offerte, pur essendo un punto di forza, si traduce in configurazioni complesse caratterizzate da una sintassi complessa e svariate tipologie di oggetti e funzionalità, il che può creare confusione.

L'automatizzazione di molti processi di gestione, se da un lato rappresenta un grande vantaggio, dall'altro li rende più o meno oscuri all'utilizzatore, diminuendone la comprensione del funzionamento del sistema.

In conclusione Kubernetes è uno strumento potente, particolarmente per la realizzazione di grandi applicazioni con necessità di gestire carichi molto elevati, a patto che se ne possieda una buona comprensione con relativa competenza tecnica.

CONCLUSIONI

Sono stati illustrati i concetti di container e orchestrazione e le necessità per far fronte alle quali sono stati ideati e adottati, relativamente allo sviluppo di software moderni, estesi e scalabili, con particolare riferimento alle architetture a micro-servizi.

Sono state inoltre presentate le principali tecnologie rilevanti per questo ambito e, con dei semplici esempi di applicazione pratica, ne abbiamo lasciato intuire le potenzialità sottolineando anche le criticità che portano con essi.

Tanto le metodologie presentate tanto le tecnologie con cui vengono messe in pratica rappresentano lo stato dell'arte nello sviluppo software moderno e con ogni probabilità rimarranno centrali per i prossimi anni. Inoltre, sebbene non sia argomento di questa trattazione, le innovazioni presentate rappresentano la spina dorsale dei servizi cloud, basati sulla virtualizzazione e l'erogazione dei servizi controllata secondo necessità, e dei nuovi business model ad esso collegati. Sono inoltre alla base delle sperimentazioni sulle architetture serverless, dove non si incentra la gestione su in singolo server ma su un contesto distribuito.

Analizzando il contesto che circonda queste tecnologie, si evince ancora una volta come l'evoluzione del mezzo tecnico sia guidata dalle necessità del contesto e del business, e come dalle nuove possibilità offerte dall'innovazione tecnologica sorgono nuovi paradigmi, modelli lavorativi ed economici.

BIBLIOGRAFIA

<https://docs.microsoft.com/it-it/virtualization/windowscontainers/about/containers-vs-vm>

https://en.wikipedia.org/wiki/OS-level_virtualization

https://www.partech.it/wp-content/uploads/2017/12/image_docker_page.png

<https://www.criticalcase.com/it/blog/container-vs-virtual-machines.html>

<https://bolcom.github.io/docker-for-testers/images/postgres-container.png>

https://miro.medium.com/max/720/1*C27c2zIHixp40boXTiEqTw.png

<https://start.spring.io/>

https://www.tosolini.info/wp-content/uploads/2020/09/feat_postgresql-1440x564_c.jpg

<https://pspdfkit.com/images/blog/2018/how-to-use-docker-compose-to-run-multiple-instances-of-a-service-in-development/article-header-d92d5dfa.png>

<https://www.guruadvisor.net/it/software-saas/939-introduzione-a-docker-pt-3-storage-e-connettivita>

<https://hackernoon.com/why-and-when-you-should-use-kubernetes-8b50915d97d8>

<https://www.redhat.com/it/topics/containers/what-is-container-orchestration>

https://media.springernature.com/original/springer-static/image/chp%3A10.1007%2F978-3-319-92378-9_14/MediaObjects/458769_1_En_14_Fig2_HTML.png

<https://medium.com/@kumargaurav1247/need-of-container-orchestration-a9f5dfbee0e3>

<https://kubernetes.io/it/docs/concepts/overview/what-is-kubernetes/>

<https://kubernetes.io/it/docs/tutorials/>

<https://severalnines.com/database-blog/using-kubernetes-deploy-postgresql>

<https://geekflare.com/wp-content/uploads/2019/12/kubernetes-architecture.png>