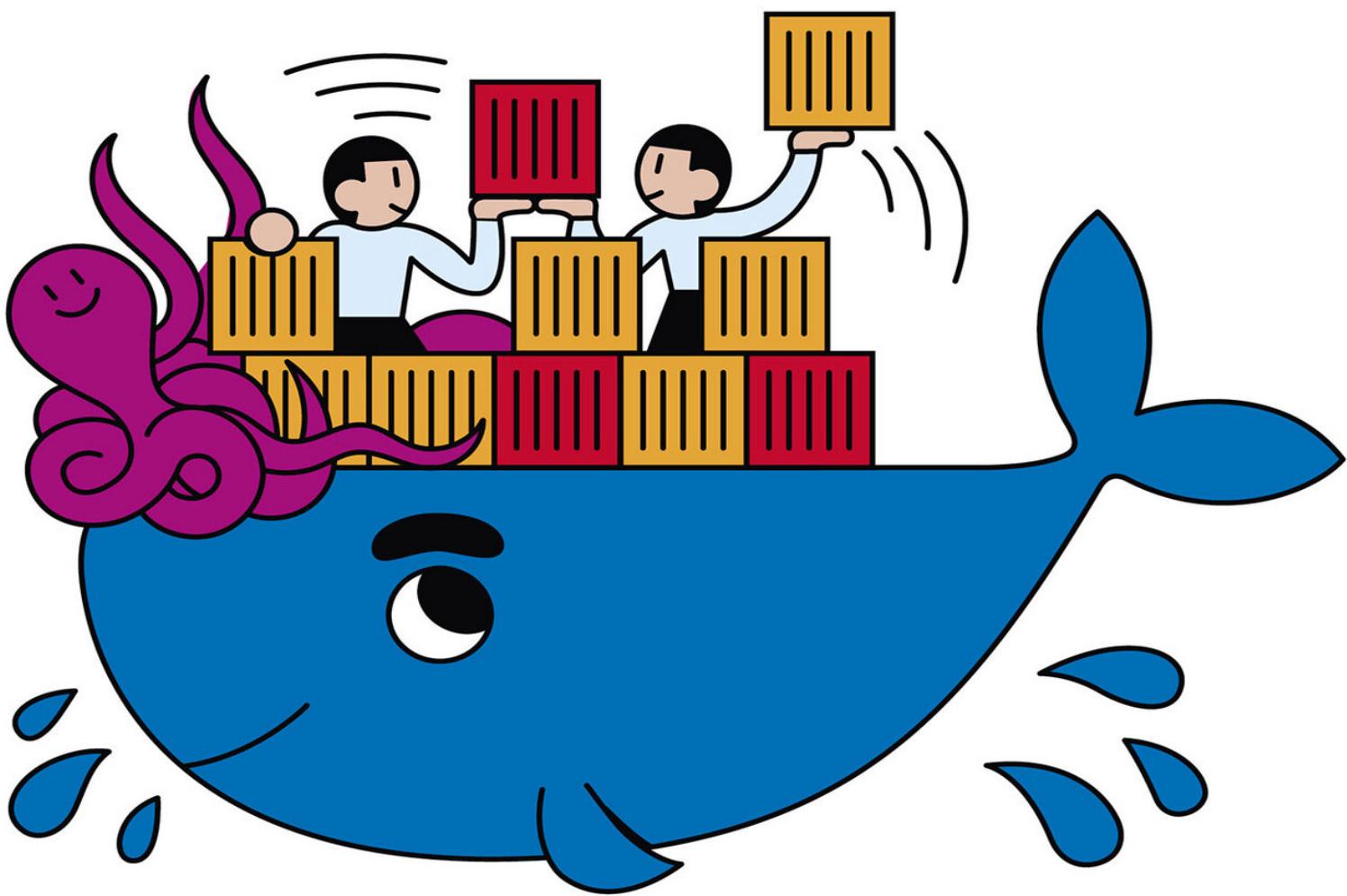


Docker



**Sviluppare e rilasciare
software tramite container**

DOCKER

Sviluppare e rilasciare software tramite container

Serena Sensini

APGEO

© Apogeo - IF - Idee editoriali Feltrinelli s.r.l.
Socio Unico Giangiacomo Feltrinelli Editore s.r.l.

ISBN ebook: 9788850319022

Il presente file può essere usato esclusivamente per finalità di carattere personale. Tutti i contenuti sono protetti dalla Legge sul diritto d'autore.

Nomi e marchi citati nel testo sono generalmente depositati o registrati dalle rispettive case produttrici.

L'edizione cartacea è in vendita nelle migliori librerie.

~

Sito web: www.apogeonline.com

Scopri le novità di Apogeo su [Facebook](#)

Seguici su [Twitter](#)

Collegati con noi su [LinkedIn](#)

Guarda cosa stiamo facendo su [Instagram](#)

Rimani aggiornato iscrivendoti alla nostra [newsletter](#)

~

Sai che ora facciamo anche CORSI? [Dai un'occhiata al calendario.](#)

*A mia sorella, senza la quale sarei persa.
A mia madre, che mi guarda con gli occhi pieni di orgoglio ogni giorno
di più.*

*A mio padre, al quale dedico ogni mia vittoria.
A te, che mi guardi nell'unico modo in cui voglio essere guardata.*

Introduzione

Quando Docker ha iniziato a farsi strada nel settore IT, in ambienti come conferenze o blog tecnologici, non era una di quelle novità ben viste, e spesso gli sviluppatori l'hanno liquidato come l'ennesimo progetto pieno di promesse, ma troppo complesso per essere adottato.

La realtà dei fatti è che i cambiamenti impauriscono: la maggior parte delle persone è restia a stravolgere le proprie abitudini, e nel settore IT c'è spesso l'usanza di "non toccare se funziona tutto bene"; forse per questo motivo, e sicuramente anche per altre ragioni, il progetto Docker e i vari strumenti a disposizione hanno avuto bisogno di qualche anno prima di entrare ufficialmente nel panorama delle soluzioni utilizzate dalle grandi aziende leader del settore.

Una frase che mi è sempre piaciuta di Andrea Camilleri è la seguente: "Arriva un momento nel quale t'adduni, t'accorgi che la tua vita è cambiata. Fatti impercettibili si sono accumulati fino a determinare la svolta. O macari fatti ben visibili, di cui però non hai calcolato la portata, le conseguenze".

Nel 2018 ho avuto l'opportunità di avvicinarmi a questa tecnologia e di apprezzarne le mille sfaccettature, soprattutto per il tipo di percorso professionale che ho intrapreso, e che continuo a portare avanti; avendo iniziato la mia esperienza lavorativa come sistemista, spesso ho avuto a che fare con architetture software complesse, mal progettate e di conseguenza difficili da implementare e manutenere. Ho rivisto in Docker tutto il buono della virtualizzazione delle soluzioni software, ma senza la difficoltà delle ore e ore di installazione e configurazione dei sistemi: con poche righe scritte in un file di configurazione, Docker permette invece di ottenere una gestione pulita e semplice anche di progetti molto complessi. Basti pensare alle aziende che sfruttano Docker (con l'ausilio di Kubernetes e di diversi cloud provider) per fornire servizi di prenotazione di vacanze, oppure per lo shopping online, o per fornire le notizie del giorno: se guardiamo la Figura I.1, troviamo i loghi di aziende leader nei diversi settori.

Spesso ho utilizzato questa tecnologia per favorire la distribuzione di nuove soluzioni o per la migrazione di soluzioni esistenti; anche questo tipo di attività diventa estremamente semplice, avendo a disposizione più di 15.000 immagini di base su cui fare affidamento per ricostruire e “ridare vita” a soluzioni monolitiche che magari sembravano giunte al termine dei loro giorni di vita.

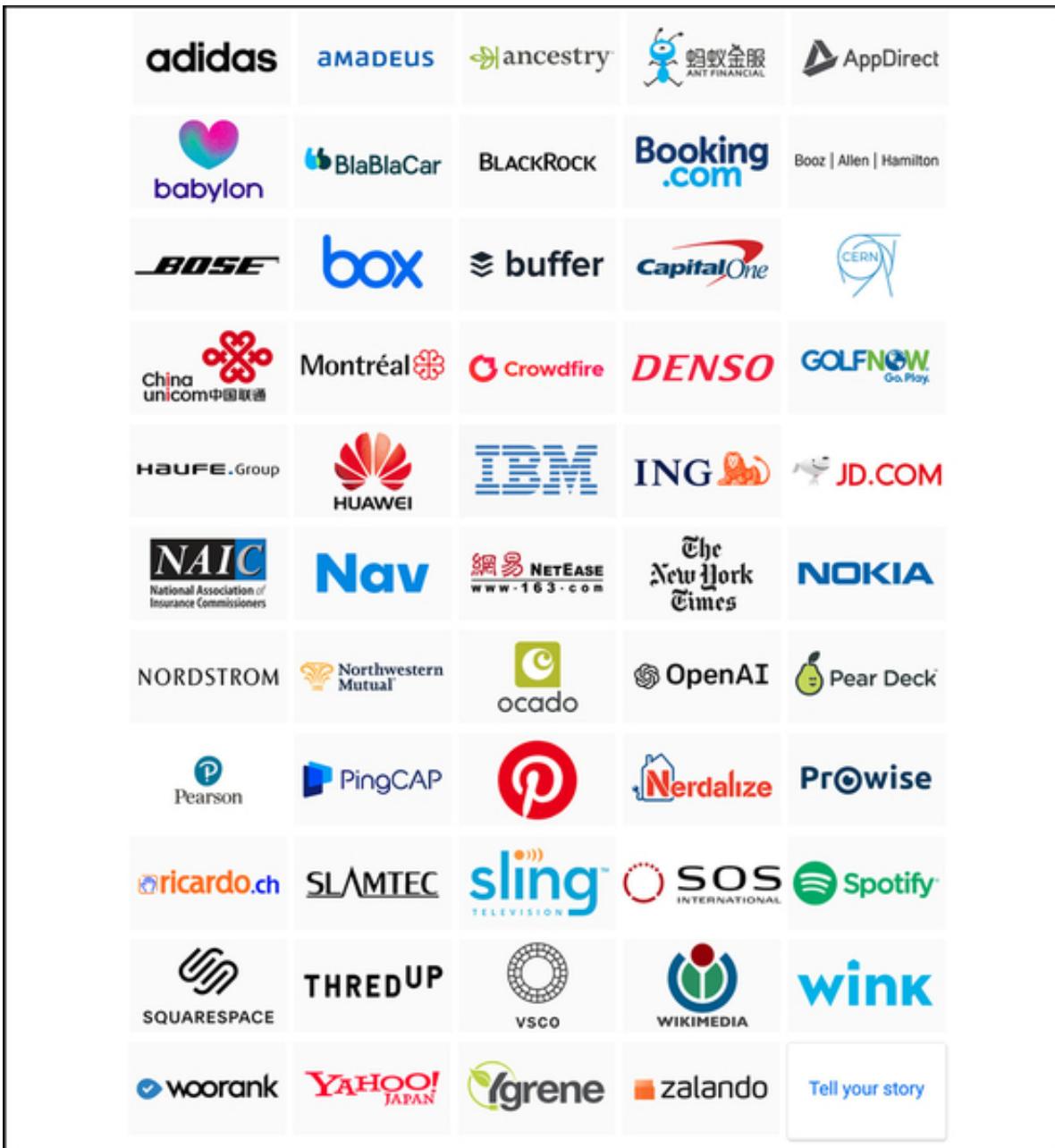


Figura I.1 Alcune aziende leader di settore che utilizzano Docker per fornire i propri servizi.

Come in tutti i lavori, ci sono stati momenti davvero difficili, in cui la documentazione fornita non sempre era di aiuto, e quasi mai in italiano.

Nel corso del tempo sono riuscita a raccogliere moltissimo materiale scritto, che mi ha aiutato a fissare procedimenti e concetti necessari alla messa in posa di diversi progetti i cui componenti di base erano simili tra loro; aggiungiamo anche che sono una di quelle persone alle quali piace fissare le idee per iscritto (sul repository GitHub ho diversi progetti che contengono guide sintetiche ma efficaci su molte tecnologie che ho utilizzato nel tempo).

Docker non è sicuramente una tecnologia di immediata comprensione, soprattutto per uno sviluppatore; la documentazione fornita è più adatta a un architetto software con esperienza pluriennale e con una mente molto razionale. In questo manuale, invece, si parte dalle basi, non dando per scontato nessun aspetto del settore in cui viene applicato, né dei casi d'uso che un qualsiasi informatico può trovarsi a dover affrontare. Infatti, dopo alcuni capitoli introduttivi che servono a porre le basi per incamerare i concetti fondamentali dei vari strumenti a disposizione, si parte subito per un viaggio molto pratico, che permette al lettore di “toccare con mano” questa tecnologia, senza inutili aforismi.

Sicuramente l’argomento centrale del manuale sono i container e la loro gestione; viene però dato ampio spazio anche ai principali “collaboratori” di Docker, che non possono e non devono mancare all’interno del bagaglio di un professionista che voglia rendere questa esperienza una vera e propria competenza: è presente infatti una sezione dedicata a Docker Swarm e a Docker Compose, strumenti per un’orchestrazione media, oltre a un capitolo introduttivo su Kubernetes, che certamente meriterebbe di essere trattato in un volume a parte.

Fondamentali sono le Appendici: contengono in breve tutti i comandi, che vengono analizzati nel dettaglio nelle diverse sezioni, complete di esempi e *cheatsheet* che possono essere utilizzati per avere un elenco rapido sempre a disposizione delle operazioni più comuni nelle diverse tecnologie analizzate.

A chi è rivolto questo manuale?

Questo manuale si rivolge a tutti coloro che hanno un minimo di esperienza nello sviluppo e in generale hanno competenze in ambito amministrativo, inteso come gestione di ambienti virtuali e non; se conosci Linux, oppure sei un sistemista con conoscenze da sviluppatore, o sei un architetto software, o se hai anche solo una certa esperienza con i concetti di base, ti troverai senz'altro a tuo agio. Se invece non rientri in queste categorie, non preoccuparti: nei primi capitoli avrai a disposizione i concetti fondamentali che ti permetteranno di comprendere al meglio quello che verrà; in ogni caso, ogni spiegazione che fa riferimento ai vari comandi o al codice utilizzato è sempre molto dettagliata e cerca di coprire tutti gli aspetti necessari alla piena comprensione di quanto sviluppato.

Il mio augurio va a te che hai deciso di intraprendere questa strada, con la speranza che il duro lavoro di mesi ti sia d'aiuto per crescere a livello professionale e personale.

Buona lettura, e buon lavoro!

Introduzione a Docker

Docker nasce in un'epoca in cui si aveva bisogno di passare da architetture monolitiche e con un forte accoppiamento tra i vari componenti di un applicativo a strutture più snelle e flessibili. Chi conosce Docker sa che la prima impressione è quella sbagliata: uno strumento che richieda capacità simili a quelle di almeno tre diversi ruoli del settore (amministratore di rete, architetto software e sistemista) non può che somigliare a una *diavoleria*. Come ho detto, l'impressione è sbagliata: questa tecnologia regala grandi libertà di movimento ai professionisti del mondo IT - nel senso più ampio del termine - che hanno buone conoscenze di base nei sopraccitati ambiti, e che però hanno bisogno di garanzie di scalabilità, di alta disponibilità di un prodotto o servizio, di servizi di logging, di monitoraggio delle risorse e via dicendo. Parliamo di tante attività che singolarmente richiedono la cooperazione di molti settori, e che non sempre collimano perfettamente. Docker rende questo tipo di operazioni a costo - quasi - zero tramite un'unica piattaforma, che può essere facilmente integrata con strumenti di orchestrazione, grazie ai quali società come Spotify, Expedia, e anche la divisione della BBC News hanno rivoluzionato il loro modo di *distribuire* le loro soluzioni.

Facciamo però un attimo un passo indietro: come nasce questo progetto, e dove siamo ora?

Le origini di Docker

Quando Docker è stato annunciato per la prima volta nel 2013 durante la Python Developers Conference a Santa Clara, in California, è stato relativamente facile per la maggior parte delle persone liquidarlo come

l'ennesimo progetto “tutto fumo, niente arrosto” prodotto nel settore tecnologico. Si sbagliavano: la tecnologia è decollata, e a fine 2014 erano state scaricate 100 milioni di immagini; all’inizio del 2017, sono arrivate a oltre otto miliardi. Il tasso di assorbimento è stato fenomenale.

Ma una cosa è applicare metodi completamente nuovi per lo sviluppo e la distribuzione di software a livello di startup o di piccole imprese, e un’altra è applicare gli stessi metodi a operazioni consolidate in ambito aziendale, quindi su scala molto più ampia.

I numeri parlano chiaro: sono passati quasi otto anni dall’annuncio iniziale di Docker, e ora abbiamo abbastanza sondaggi, statistiche ed esperienze maturate in diversi settori per fornire un quadro ragionevolmente completo della sua attuale posizione nel mondo IT: un ritratto di chi lo sta usando, chi è seriamente intenzionato ad adottarlo e a quali risultati è probabile che sia possibile giungere.

I *container* sono disponibili da molti anni nelle distribuzioni *Unix-based*, ma raramente sono stati utilizzati, a causa della complessità richiesta per l’implementazione di uno di questi ambienti. Pertanto, storicamente, le implementazioni di container sono state costruite appositamente con un unico obiettivo in mente, il che ha reso difficile, se non impossibile, fornire qualità aggiuntive, come ridimensionamento e portabilità.

“Docker”: da dove nasce il nome?

La società *dotCloud, Inc.* viene fondata nel 2010 come PaaS: l’obiettivo iniziale era quello di fornire strumenti di *deploy* di applicazioni tramite servizi *cloud*, senza che il cliente dovesse preoccuparsi di *build* o manutenzione dei prodotti. Questa piccola realtà nasce da una startup finanziata all’interno della Silicon Valley tramite il programma *Y Combinator*; lavorano al progetto, chiamato poi Docker, in open-source e sviluppano il concetto di container come uno strumento accessibile dalla riga di comando Python. Il nome iniziale era *dc*, abbreviazione di *dock*, e si basava su servizi chiamati *cloudlet*; nel 2013 Solomon Hykes, uno dei fondatori del gruppo, decide di presentare quello che finora era un progetto interno al Pycon; il progetto ha così tanto successo che dirottano tutti i loro sforzi nel portarlo a termine, compresa l’idea di aprire un distaccamento dell’azienda, che prende il nome proprio di *Docker, Inc.*, per via di *dock*, che viene fuso con la parola *worker* formando appunto il nome *Docker*.

Docker, in un certo senso, ha dato origine a una sorta di Rinascimento dei container Linux, guidando una crescente ondata di interesse e possibilità e portandolo a una rapida adozione nel settore IT. Tramite Docker, i container Linux si sono ritrovati catapultati in una élite di

tecnologie davvero rivoluzionarie, con il potere di trasformare il panorama informatico, gli ecosistemi correlati e i mercati.

Innovazione spesso non significa necessariamente introdurre una tecnologia completamente nuova; come molti suoi predecessori, il successo di Docker è alle spalle dei giganti. Si basa su anni di innovazione tecnologica ed evoluzione, che ora forniscono i mattoni di base che semplificano l'utilizzo di strumenti come Docker.

Chi utilizza Docker?

Alcuni nomi sono già stati fatti, ma non parliamo solo di grandi aziende: startup, aziende di medie dimensioni, organizzazioni di livello aziendale hanno adottato sempre più frequentemente Docker come realtà delle proprie aziende.

Ma se un numero crescente di aziende si impegna a utilizzare Docker, che cosa le ha portate a fare questa scelta e come gestiscono il passaggio dalla distribuzione monolitica a quella basata su container?

Il quadro generale che emerge tra i professionisti IT è sempre lo stesso: l'installazione e la manutenzione dell'infrastruttura di una soluzione software spesso sono stati compiti impegnativi, mentre l'implementazione di una soluzione basata su Docker in un ambiente di produzione è relativamente semplice e senza complicazioni.

I *container*, concetto cuore su cui Docker pone le sue fondamenta, non sono una tecnologia unica; sono una raccolta di tecnologie che sono state sviluppate in oltre dieci anni. Le funzionalità di Linux (come i *namespace* e i *cgroup*) sono disponibili già da diverso tempo - dal 2008 circa.

Perché, allora i *container* non sono stati usati per tutto quel tempo? La risposta è che pochissime persone hanno saputo trovare il modo di realizzarli.

Docker è molto popolare per le possibilità che offre per la distribuzione del software. Molti problemi come l'inefficienza o la flessibilità vengono facilmente risolti con l'uso di container. Tra i motivi principali che portano alla popolarità questo strumento ci sono i seguenti.

1. *Facilità d'uso.* Gran parte della popolarità di Docker è grazie alla semplicità su cui si basa l'infrastruttura. Docker può essere appreso rapidamente, anche grazie alle molte risorse disponibili online per imparare a creare e gestire i container. Docker è open-source, quindi tutto ciò che serve per iniziare è un computer con un sistema operativo che supporti Virtualbox, e il gioco è fatto.
2. *Ridimensionamento più rapido dei sistemi.* I container consentono di svolgere molto più lavoro rispetto alle capacità hardware realmente presenti, e magari di gran lunga inferiori alle aspettative. All'inizio di Internet, l'unico modo per ridimensionare un sito web era acquistare o affittare più server. Pensiamo quindi al costo del ridimensionamento: molti siti sono diventati vittime del loro stesso successo, sborsando decine di migliaia di dollari per acquistare nuove risorse hardware. I container consentono agli operatori di *data center* o ai fornitori di servizi *cloud* di garantire ai clienti grandi carichi di operazioni, utilizzando meno hardware.
3. *Migliore distribuzione del software.* La distribuzione del software tramite container può anche essere più efficiente. I container sono portatili, ossia sono completamente autonomi. I container includono un volume del disco isolato; tale volume viaggia accompagnando il container, proprio perché sviluppato e distribuito in vari ambienti. Le dipendenze del software (librerie, compilatori e così via) vengono *impacchettati* all'interno del container. Se un container funziona in locale, verrà eseguito allo stesso modo in un ambiente di sviluppo, collaudo e produzione. I container possono eliminare i problemi di variabilità della configurazione, comuni durante la distribuzione di binari o codice non elaborato.
4. *Flessibilità.* Il funzionamento delle applicazioni in container è più flessibile e resiliente di quello delle applicazioni che non li utilizzano. Gli orchestrator - che vedremo più avanti - si occupano dell'esecuzione e del monitoraggio di centinaia o migliaia di container, fornendo un unico strumento di gestione dell'ambiente. Gli *orchestrator* sono strumenti molto potenti per la gestione di grandi implementazioni e sistemi complessi. Forse l'unica cosa più popolare di Docker in questo momento è proprio Kubernetes, attualmente l'orchestrator più diffuso.

5. *Networking*. Docker supporta reti definite dal software. L'interfaccia a riga di comando e il Docker Engine consentono agli sviluppatori di definire reti isolate per i container, senza dover toccare un singolo *router*. Gli sviluppatori e gli architetti software, o coloro che lavorano con queste tecnologie, possono progettare sistemi con topologie di rete complesse e definire le reti nei file di configurazione. Questo rappresenta un enorme vantaggio per la sicurezza: i container di un'applicazione possono essere eseguiti in una rete virtuale isolata, con percorsi di ingresso e uscita strettamente controllati.
6. *Architetture a microservizi*. Anche l'ascesa dei microservizi ha contribuito alla popolarità di Docker. I *microservizi* sono funzioni semplici, di solito accessibili tramite HTTP/HTTPS, che fanno una sola cosa e la fanno bene. I sistemi software in genere iniziano come "monoliti", in cui un singolo binario supporta più funzioni di sistema, differenti. A mano a mano che crescono, i monoliti possono diventare difficili da manutenere e distribuire. I microservizi suddividono un sistema in funzioni più semplici, che possono essere implementate in modo indipendente. I *container* sono host eccezionali per i microservizi: autonomi, facilmente implementabili ed efficienti.

Dovrei iniziare a usare Docker? Una domanda come questa ha quasi sempre come migliore risposta la cautela e la circospezione. Nessuna tecnologia è una panacea. Ogni tecnologia presenta inconvenienti, compromessi e avvertenze.

Detto questo... *Sì, usa Docker.*

Alcuni presupposti a sostegno di questa risposta:

- devi sviluppare software distribuito con l'intento di spremere dalla tua infrastruttura ogni hertz e ogni byte della RAM a disposizione;
- stai progettando il software per carichi e prestazioni elevati, anche se non hai ancora carichi elevati o hai bisogno delle migliori prestazioni in ambienti diversi da quelli di sviluppo;
- vuoi garantire un'alta velocità di distribuzione e trarne i benefici. Se aspiri alle pratiche DevOps nella consegna del software, i container

sono uno strumento chiave in questo punto del lavoro.

Quando non dovresti usare Docker

Lo sviluppo, la distribuzione e il funzionamento del software nei container sono molto diversi dallo sviluppo e dalla messa in produzione tradizionali.

Ci sono alcune valutazioni da fare prima di usare Docker.

1. *Il team ha bisogno di training.* Il set di competenze del tuo team richiede una significativa considerazione. Se ti mancano il tempo o le risorse per gestire i container o per disporre di un partner di consulenza per farti lavorare, dovresti aspettare. Lo sviluppo e le attività con i container non sono qualcosa che si possa “capire mentre vai”, a meno che il tempo a disposizione non sia quantificabile in mesi. Nonostante tutta la sua popolarità, Docker è un nuovo modo di sviluppare e distribuire software. L’ecosistema è in continua evoluzione e la popolazione di ingegneri esperti in questo campo è ancora relativamente piccola. Durante questa fase iniziale, molte aziende scelgono di collaborare con i partner ISV Enterprise per iniziare con Docker e i relativi sistemi. Se questa non è un’opzione per te, ti consigliamo di bilanciare il costo dell’utilizzo di Docker da solo con i potenziali benefici.
2. *Profilo ad alto rischio.* Il tuo profilo di rischio è un’altra considerazione importante di cui tenere conto. Se ti trovi in un settore regolamentato o esegui carichi di lavoro che generano entrate, fai attenzione ai container. L’uso di container su larga scala con orchestratori è molto diverso rispetto ai sistemi non containerizzati. I vantaggi dei container derivano da un’ulteriore complessità nei sistemi che li consegnano, gestiscono e monitorano.
3. *Bassa complessità del sistema.* Infine, considera i tuoi requisiti generali. I tuoi sistemi sono sufficientemente complessi da giustificare l’onere aggiuntivo derivante dalla containerizzazione? Per esempio, se la tua azienda è incentrata sulla creazione di siti web statici, potresti non aver bisogno di container.

In conclusione, Docker è popolare perché ha rivoluzionato la modalità di gestione dello sviluppo: in cinque anni la sua popolarità come strumento e piattaforma è salita alle stelle. Il motivo principale è che i container creano vaste economie di scala. I sistemi che richiedevano costose risorse hardware dedicate possono ora condividere l'hardware con altri sistemi.

È importante ricordare, ancora una volta, che Docker non è una panacea (nessuna tecnologia lo è). Ci sono dei compromessi da considerare quando si pianifica una strategia tecnologica, e passare ai container non è un'impresa banale. Considera i compromessi prima di impegnarti in una strategia basata su Docker. Un'attenta contabilità dei benefici e dei costi della containerizzazione può portarti ad adottare Docker. Se i numeri si sommano, Docker e container hanno il potenziale per aprire nuove opportunità per un'azienda.

Architetture software

Quando le persone nell'industria del software parlano di "architettura", si riferiscono a una nozione definita pericolosamente come uno degli aspetti più importanti della progettazione di un sistema software. Una buona architettura è importante, altrimenti diventa più lento e costoso aggiungere nuove funzionalità, in futuro.

Il termine "architettura" spesso suggerisce una separazione dalla programmazione e dona una malsana dose di pomposità allo sviluppo di una soluzione software; in realtà, una buona architettura è qualcosa che supporta la sua stessa evoluzione ed è profondamente intrecciata con la programmazione. La maggior parte delle carriere degli sviluppatori - sempre nel senso più ampio possibile del termine - si è concentrata sulle domande su come sia una buona architettura, su come i team possano crearla e su come coltivare al meglio un pensiero architettonico nei team di sviluppo.

Cos'è l'architettura?

Le persone nel mondo IT hanno discusso a lungo su una definizione precisa di *architettura*. Per alcuni è qualcosa di simile all'organizzazione

e alla pianificazione di un sistema o al modo in cui si legano fra loro i componenti di livello più elevato. Probabilmente la definizione che si avvicina di più nel raccogliere il pieno significato del termine è la seguente:

Un'architettura è l'intesa condivisa che gli sviluppatori hanno della progettazione del sistema.

Un secondo stile comune di definizione per l'architettura è che sono “le decisioni di progettazione che devono essere prese all'inizio di un progetto”; in breve, “L'architettura riguarda le cose importanti. Qualsiasi esse siano”. A prima vista, sembra un'affermazione banale, ma in realtà porta molta ricchezza: il cuore di pensare dal punto di vista architettonico il software è decidere che cosa è importante, (ossia ciò che è architettonico), e spendere energia per mantenere quegli elementi in buone condizioni. Affinché uno sviluppatore diventi un architetto, deve essere in grado di riconoscere quali elementi sono importanti, riconoscendo quali possono causare seri problemi se non vengono opportunamente controllati.

Le decisioni importanti nello sviluppo del software variano a seconda della scala di contesto al quale stiamo pensando. Una scala comune è quella di un'applicazione, quindi “architettura dell'applicazione”.

Il primo problema con la definizione dell'architettura di un'applicazione è che non esiste una definizione chiara di cosa sia un'applicazione. La mia opinione è che le applicazioni siano una costruzione sociale:

- un blocco di codice che viene visto dagli sviluppatori come un singolo prodotto;
- un gruppo di funzionalità che viene messo a disposizione del cliente;
- un insieme di requisiti funzionali, con il relativo costo.

Tutti punti di vista molto differenti - ma tutti validi -, che rendono la definizione molto ampia; questo porta infatti a diverse dimensioni potenziali di un'applicazione, che variano da poche decine a diverse centinaia di persone nel team di sviluppo.

Viste le premesse fatte, è doveroso sottolineare che l'architettura software, nel senso più generico, riguarda una moltitudine di soluzioni

differenti, che portano apparentemente allo stesso risultato - che non corrisponde alla resa. Possiamo infatti parlare di architetture *serverless*, di sistemi distribuiti o di sistemi a servizi o microservizi; ognuna di queste merita di essere trattata, anche se rapidamente, in maniera separata. Questa è, infatti, una parte fondamentale per poter apprezzare fino in fondo gli strumenti che utilizzeremo nel prossimo capitolo.

Sistemi distribuiti

Un *sistema distribuito* è un tipo di architettura con più componenti situati su macchine distinte che comunicano e coordinano le proprie azioni per apparire all'utente finale come un unico sistema coerente. Vengono quindi utilizzati vari elementi hardware e software; un sistema distribuito rientra in genere in una delle seguenti architetture di base: client-server, tre livelli, n-livello o peer-to-peer.

Le macchine che fanno parte di un sistema distribuito possono essere computer, server fisici, macchine virtuali, container o qualsiasi altro nodo che può connettersi alla rete, avere memoria locale e comunicare passando messaggi.

Esistono due modi generali di funzionamento dei sistemi distribuiti.

- Ogni macchina lavora per un obiettivo comune, e l'utente finale visualizza i risultati come un'unica unità coesa.
- Ogni macchina ha il proprio utente finale e il sistema distribuito facilita la condivisione delle risorse o dei servizi di comunicazione.

Sebbene i sistemi distribuiti possano talvolta essere oscuri, di solito hanno tre caratteristiche principali: tutti i componenti funzionano contemporaneamente, non esiste un clock globale e tutti i componenti si guastano indipendentemente l'uno dall'altro.

Ci sono tre ragioni per cui i team, generalmente, decidono di implementare dei sistemi distribuiti.

- *Scalabilità orizzontale*: poiché l'elaborazione avviene in modo indipendente su ciascun nodo, è facile, e generalmente economico, aggiungere ulteriori nodi e funzionalità, se necessario.

- *Affidabilità*: la maggior parte dei sistemi distribuiti è tollerante ai guasti, in quanto essi possono essere costituiti da centinaia di nodi che lavorano insieme. In genere il sistema non subisce interruzioni in caso del guasto di una singola macchina.
- *Prestazioni*: i sistemi distribuiti sono estremamente efficienti, perché i carichi di lavoro possono essere suddivisi e inviati a più macchine.

Tuttavia, i sistemi distribuiti non sono privi di sfide: i complessi processi di progettazione architettonica, costruzione e debug necessari per creare un sistema distribuito efficace possono essere davvero faticosi.

Tre sfide che è possibile dover affrontare nella progettazione di un sistema distribuito sono le seguenti.

- *Pianificazione*: un sistema distribuito deve decidere quali lavori devono essere eseguiti, quando devono essere eseguiti e dove devono essere eseguiti. Gli *scheduler*, alla fine, hanno dei limiti, che portano ad avere hardware sottoutilizzati e tempi di esecuzione imprevedibili.
- *Latenza*: maggiore è la distribuzione del sistema, maggiore è la latenza che si può verificare con le comunicazioni. Ciò porta spesso i team a fare compromessi tra disponibilità, coerenza e latenza.
- *Osservabilità*: raccogliere, elaborare, presentare e monitorare le metriche di utilizzo dell'hardware per i cluster di grandi dimensioni è una sfida significativa.

Architettura orientata ai servizi

Questo tipo di architettura rispecchia uno stile di progettazione software in cui i servizi vengono forniti agli altri componenti dai componenti dell'applicazione, attraverso un protocollo di comunicazione su una rete. I suoi principi sono indipendenti dai fornitori e da altre tecnologie. Nell'architettura orientata ai servizi, numerosi servizi comunicano tra loro, in due modi: attraverso il passaggio di dati o attraverso due o più servizi che coordinano un'attività.

Un servizio ha l'obiettivo di encapsulare una ben precisa funzionalità di logica applicativa per renderla accessibile come servizio da parte di client

sul Web. La cosiddetta SOA (*Service-Oriented Architecture*, ovvero architettura orientata ai servizi) è uno stile di architettura per la costruzione di diversi sistemi o applicazioni sulla base della composizione di un insieme di servizi; questo tipo di approccio fa largo uso dei *web service*, ossia dei moduli che sono accessibili tramite Internet e che rappresentano veri e propri servizi che forniscono dati in risposta a una richiesta dell’utente. Esistono diversi tipi di web service, a seconda del tipo di capacità fornita dal servizio.

L’architettura orientata ai servizi si occupa meno di come modularizzare un’applicazione e più di come comporre un’applicazione mediante l’integrazione di componenti software distribuiti e gestiti separatamente. Esistono due ruoli principali in una SOA: un fornitore di servizi e un consumatore di servizi, anche se un agente software può svolgere entrambi i ruoli.

Quando si tratta di implementare l’architettura orientata ai servizi (SOA), è possibile utilizzare una vasta gamma di tecnologie, a seconda dell’obiettivo finale e di ciò che si sta tentando di realizzare. Un esempio di standard del servizio web è SOAP, che sta per *Simple Object Access Protocol*; in breve, SOAP è una specifica del protocollo di messaggistica per lo scambio di informazioni strutturate nell’implementazione di servizi web nelle reti di computer. Sebbene all’inizio SOAP non sia stato ben accolto, dal 2003 ha guadagnato popolarità ed è sempre più utilizzato e accettato. Un’altra tipologia di web service sono quelli basati su REST, in cui i dati e le funzionalità sono considerati risorse e sono accessibili tramite URI. In questo caso, l’architettura deve essere di tipo client-server e viene progettata per essere utilizzata tramite un protocollo di comunicazione senza stato, in genere HTTP.

Microservizi

Ogni architettura deve arrivare a compromessi: punti di forza e punti deboli che dobbiamo valutare in base al contesto in cui viene utilizzato. Questo è certamente il caso dei *microservizi*; sebbene sia un’architettura utile, la maggior parte delle situazioni richiede un’architettura a monolite. I microservizi offrono vantaggi quali una struttura *modulare*, che rende la soluzione complessiva flessibile e facilmente scalabile, *indipendente*, motivo per cui la gestione delle prestazioni e della resilienza è più

semplice da gestire, nonché l'indipendenza della piattaforma dalla tecnologia utilizzata: con un'architettura basata su microservizi è possibile combinare più linguaggi, framework di sviluppo e tecnologie di gestione dei dati.

Il termine *architettura a microservizi* è sorto negli ultimi anni per descrivere un modo particolare di concepire la progettazione di applicazioni software come suite di servizi indipendenti. Sebbene non esista una definizione precisa di questo stile architettonico, esistono alcune caratteristiche comuni circa l'organizzazione in termini di capacità aziendale, distribuzione automatizzata, efficienza nella gestione degli *endpoint* e nel controllo decentralizzato dei dati.

In altre parole, questa modalità di lavoro è un approccio allo sviluppo di una singola applicazione come una suite di piccoli servizi, ciascuno dei quali è in esecuzione nel proprio processo e in comunicazione con meccanismi leggeri, spesso tramite API disponibili via HTTP. Questi servizi si basano su macro-funzionalità e possono essere distribuiti in modo indipendente da risorse di distribuzione completamente automatizzate.

Per iniziare a spiegare il design architettonico dei microservizi, è utile confrontarlo con lo stile monolitico: un'applicazione costruita come una singola unità. Spesso le applicazioni aziendali sono costruite in tre parti principali: un'interfaccia utente lato client (composta da pagine HTML e Javascript in esecuzione in un browser sul computer dell'utente), un database (costituito da più tabelle inserite in una gestione di database comune e solitamente relazionale) e un'applicazione lato server.

L'applicazione lato server gestirà le richieste, eseguirà la logica del servizio, recupererà e aggiornerà i dati dal database e selezionerà e popolerà le viste da inviare al browser. Un'applicazione di questo tipo è un *monolite*, ossia un singolo eseguibile logico. Qualsiasi modifica al sistema implica la creazione e la distribuzione di una nuova versione dell'applicazione lato server.

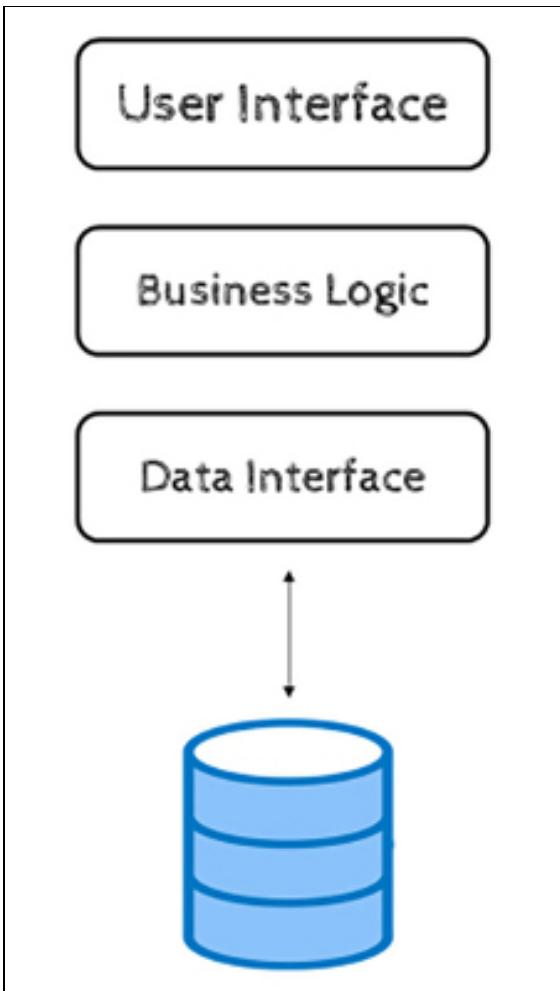


Figura 1.1 Esempio di architettura monolitica.

Tutta la logica per la gestione di una richiesta viene eseguita in un unico processo, consentendo di utilizzare le funzionalità di base del proprio linguaggio per suddividere l'applicazione in classi, funzioni e package. Con un po' di attenzione, possiamo eseguire e sottoporre a test l'applicazione sul sistema di uno sviluppatore e utilizzare una pipeline di distribuzione per garantire che le modifiche vengano correttamente sottoposte a test e distribuite in produzione.

Le applicazioni monolitiche possono avere successo, ma sempre più persone scelgono di non adottare questa via, soprattutto quando vengono distribuite più applicazioni in un sistema basato su una distribuzione cloud. I cicli di modifica sono collegati tra loro: una modifica apportata a una piccola parte dell'applicazione richiede la ricostruzione e la distribuzione dell'intero monolite. Al contempo, spesso è difficile

mantenere una buona struttura modulare; ciò rende più difficile mantenere all'interno di un modulo i cambiamenti che dovrebbero interessare solo quel modulo. Un ridimensionamento dovrà agire sull'intera applicazione piuttosto che su quelle parti di essa che richiedono maggiori risorse.

Quando parliamo di componenti ci imbattiamo nella difficile definizione di ciò che realmente significa parlare di un componente. La definizione che più ci si avvicina è che un *componente* rappresenta un'unità di software che è sostituibile e aggiornabile in modo indipendente.

Le architetture di microservizi possono utilizzare diverse librerie, ma il modo principale di strutturare il software è quello di suddividere i servizi. Definiamo le *librerie* come componenti collegati in un programma, cui si fa accesso mediante chiamate di funzione in memoria, mentre i *servizi* sono componenti fuori processo che comunicano tramite servizi web o chiamate a procedura. Uno dei motivi principali per l'utilizzo dei servizi (anziché delle librerie) come componenti è che sono distribuibili indipendentemente. Se si dispone di un'applicazione che consiste di più librerie in un singolo processo, una modifica a qualsiasi singolo componente comporta la ridistribuzione dell'intera applicazione. Ma se tale applicazione viene scomposta in più servizi, è possibile aspettarsi che anche molte modifiche a un singolo servizio richiedano solo la ridistribuzione di quel servizio.

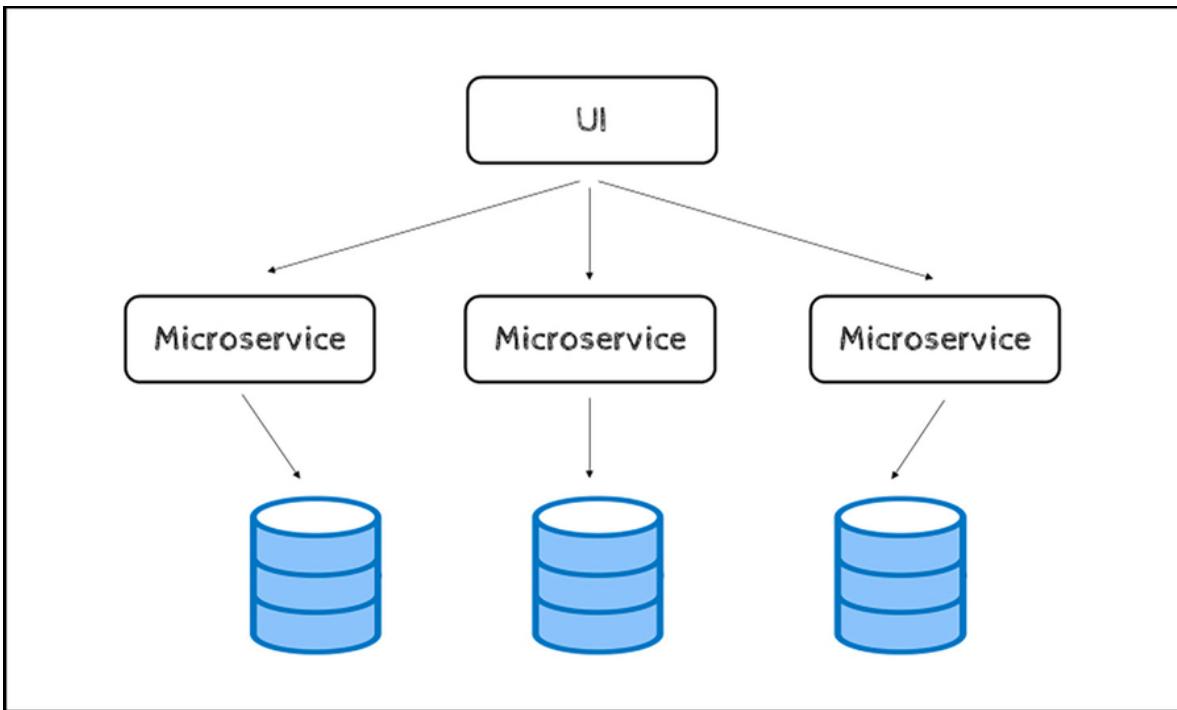


Figura 1.2 Esempio di architettura a microservizi.

A questo punto, è normale porsi la fatidica domanda: un’architettura a microservizi rappresenta una buona scelta per la soluzione per il progetto al quale lavoro?

“Dipende” è sicuramente la risposta più giusta, anche se è bene specificare da quali fattori dipende questa scelta. Il fulcro dell’uso o meno dei microservizi è la complessità del sistema da realizzare. L’approccio dei microservizi riguarda la gestione di un sistema complesso, ma per fare ciò si introduce comunque una serie di complessità: quando si utilizzano microservizi, è necessario lavorare su distribuzione automatizzata, monitoraggio, gestione dei guasti, eventuale coerenza e altri fattori introdotti da un sistema distribuito. Ci sono diversi modi per far fronte a tutto questo, ma è uno sforzo aggiuntivo e nessuno dei programmatore che lavorano nel settore IT da più di un decennio sembra avere tempo da impiegare a determinate attività, soprattutto se non rientrano tra le mansioni “quotidiane”.

Quindi la linea guida principale sarebbe quella di non considerare i microservizi, a meno che non si abbia un sistema troppo complesso da gestire come un monolite. È necessario prestare attenzione a garantire comunque una buona modularità dei componenti del monolite.

La complessità che ci spinge ai microservizi può nascere per diverse ragioni, tra cui la gestione di team di grandi dimensioni, la *multi-tenancy* (ossia quando un’istanza di un sistema è eseguita su un server, ma fruibile da diversi client), il supporto di più modelli di interazione con l’utente, consentendo alle diverse funzioni di evolversi in modo indipendente e facilitando il ridimensionamento; il fattore più importante è quello delle dimensioni: spesso si scopre solo alla fine dello sviluppo di un progetto di avere un monolite troppo grande per essere modificato e distribuito.

Architetture serverless

Come molte tendenze del software, non esiste una visione chiara di cosa sia un sistema *serverless*. Per cominciare, comprende due aree diverse ma sovrapposte.

- Serverless è un termine che è stato utilizzato per la prima volta per descrivere applicazioni che incorporano in modo significativo o completo applicazioni e servizi di terze parti, ospitate su cloud, per gestire lo stato e la logica lato server. Si tratta in genere di applicazioni *rich client* (per esempio app web a pagina singola o app mobili) che utilizzano database accessibili da risorse cloud (per esempio Firebase), servizi di autenticazione (come OAuth) e altro. Questi tipi di servizi sono stati precedentemente descritti come *(Mobile) Backend as a Service*, abbreviato in BaaS.
- Serverless può anche significare applicazioni in cui la logica lato server è scritta dallo sviluppatore dell’applicazione, ma, a differenza delle architetture tradizionali, viene eseguita in istanze senza stato attivate da eventi (come una chiamata di un servizio) e completamente gestite da una terza parte. Un modo di chiamarle è “Funzioni come servizio” o FaaS. AWS Lambda è attualmente una delle implementazioni più popolari di questo tipo di architettura, ma ce ne sono molte altre.

Pensiamo a un tradizionale sistema *client-oriented* a tre livelli con logica lato server. Un buon esempio è una tipica app di e-commerce: per

esempio un negozio di libri online.

Tradizionalmente, l'architettura sarà simile al diagramma rappresentato nella Figura 1.3. Supponiamo che sia implementato in Java o Javascript lato server, con la parte di UI in HTML e Javascript come client.

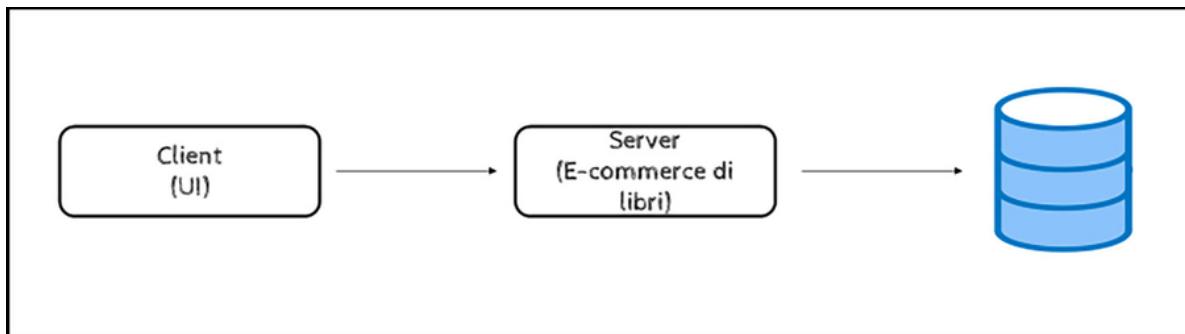


Figura 1.3 Esempio di sistema per un e-commerce per la compravendita di libri.

Con un'architettura serverless, il sistema di e-commerce sarebbe più simile a quello rappresentato nella Figura 1.4.

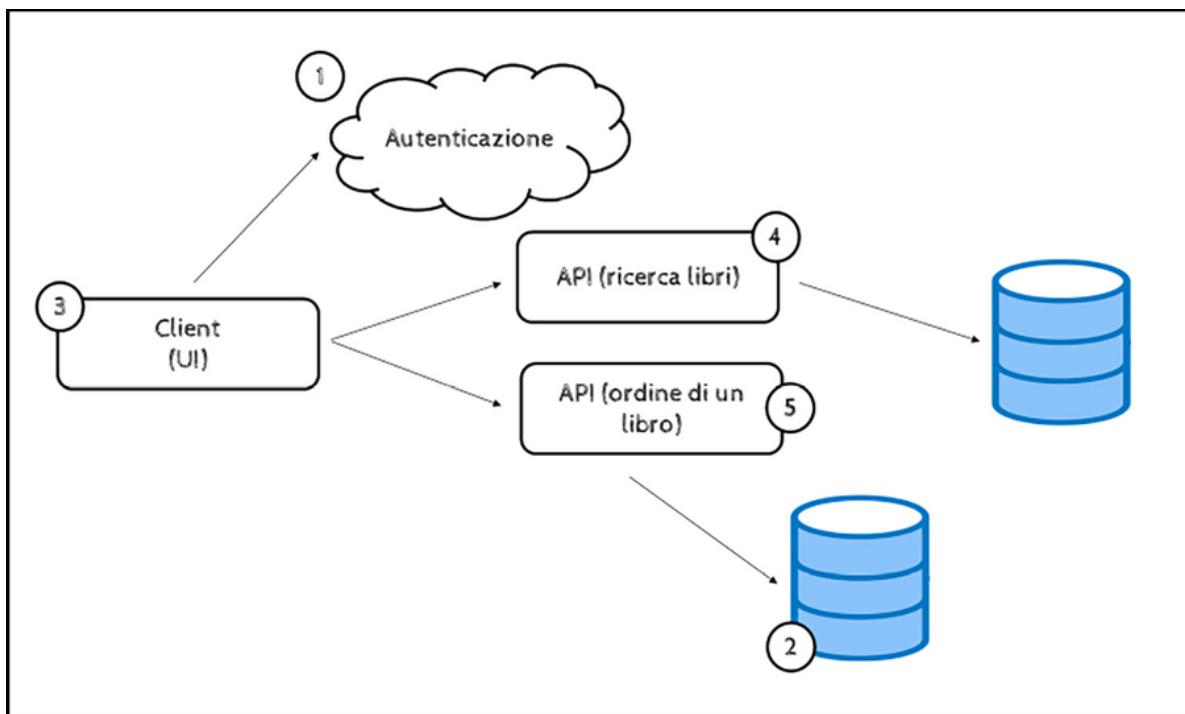


Figura 1.4 Architettura serverless per il sistema di e-commerce.

Questo esempio mostra un punto molto importante: nella versione originale, tutto il flusso, il controllo e la sicurezza erano gestiti

dall'applicazione sul server centrale. Nella versione serverless non esiste una gestione “centrale” per tutte queste funzioni; esiste una preferenza per una coreografia asincrona rispetto all'orchestrazione, dove ogni componente gioca un ruolo più consapevole dal punto di vista architettonico. L'idea è comune anche nell'approccio ai microservizi, come visto in precedenza.

Pattern

Un *pattern* di architettura è il modello di una soluzione riutilizzabile a un problema comune nell'architettura software in un determinato contesto. In altre parole, si tratta di soluzioni generalizzate di un dato problema, le quali guidano l'organizzazione dello sviluppo di una soluzione. Ci tornerà utile elencare qui alcuni dei più comuni pattern di architetture per la costruzione del software, così da mantenere elevati i livelli di qualità della progettazione e del design, nonché di resa delle funzionalità da implementare.

Layered pattern

Questo modello può essere utilizzato per strutturare programmi che possono essere scomposti in gruppi di attività autonome, ognuna delle quali si trova a un determinato livello di astrazione. Ogni layer fornisce servizi al layer direttamente superiore.

I quattro layer (o livelli o strati) più comuni di un sistema di informazione generale sono i seguenti.

- Layer di presentazione (noto anche come UI, *interfaccia utente*).
- Layer dell'applicazione (noto anche come *layer di servizio*).
- Layer della logica di business (noto anche come *layer di dominio*).
- Layer di accesso ai dati (noto anche come *layer di persistenza*).

Lo scenario rappresentato nella Figura 1.5 è quello più comune e rappresenta una *comunicazione top-down*: un client effettua una richiesta al *Layer n*, la richiesta viene via via decomposta in sotto-richieste per i

Layer n-1, n-2 e così via, e ciascuna richiesta può dar luogo a una risposta; le risposte alle diverse richieste vengono via via elaborate e combinate - e risalgono tra i layer. Un esempio classico è quello di un'applicazione web: un utente si collega a una pagina web ed effettua una ricerca tramite parole chiave. Il layer con cui l'utente si interfaccia è quello di presentazione, ossia la pagina web; questa, a sua volta, propone i risultati della ricerca dell'utente andando a interrogare il layer sottostante, ossia il layer della logica di business, che recupera i dati dal layer di persistenza.

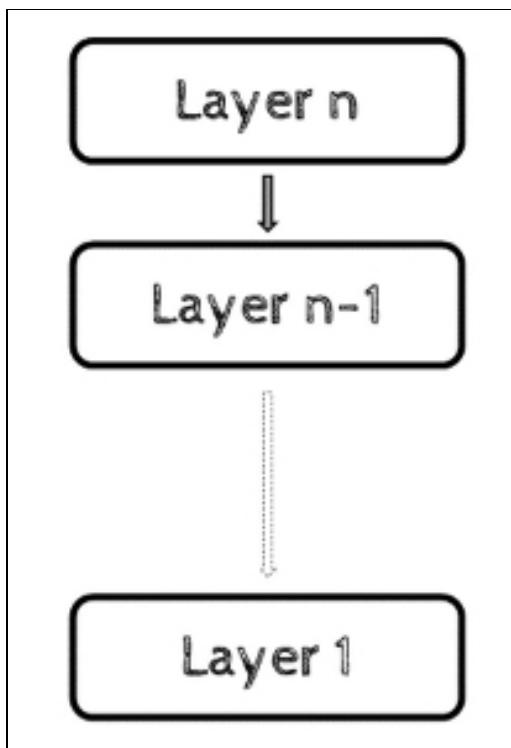


Figura 1.5 Rappresentazione del pattern.

Client-server pattern

Questo modello è composto da due parti: un server e uno o più client. I client possono invocare i servizi dei server: un client può iniziare l'interazione con un server, invocando da un server un servizio di cui ha bisogno e aspettando il risultato di tale richiesta. I server hanno invece la

responsabilità di erogare i servizi, specifici, richiesti da ciascun client e sono sempre a disposizione delle richieste dei client (Figura 1.6).

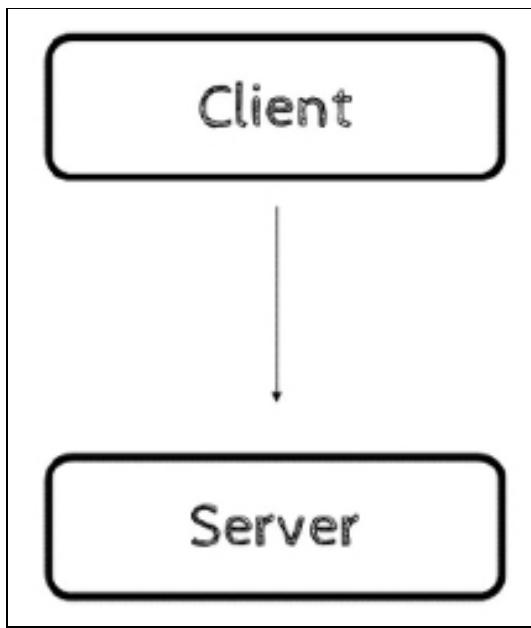


Figura 1.6 Rappresentazione client-server.

Nell'esempio rappresentato nella Figura 1.6 è riportata l'interazione di un singolo client con un server: un'*architettura two-tier*. In questo caso, il client comunica direttamente con il server, senza bisogno di intermediari. Il modello può essere esteso utilizzando un'interfaccia che funge da ponte nella comunicazione tra le parti; questo tipo di architettura si definisce *three-tier* e viene supportata da uno strato di *middleware* che normalmente permette la comunicazione tra l'interfaccia utente e il sistema di persistenza, come nel caso sopracitato dell'applicazione web.

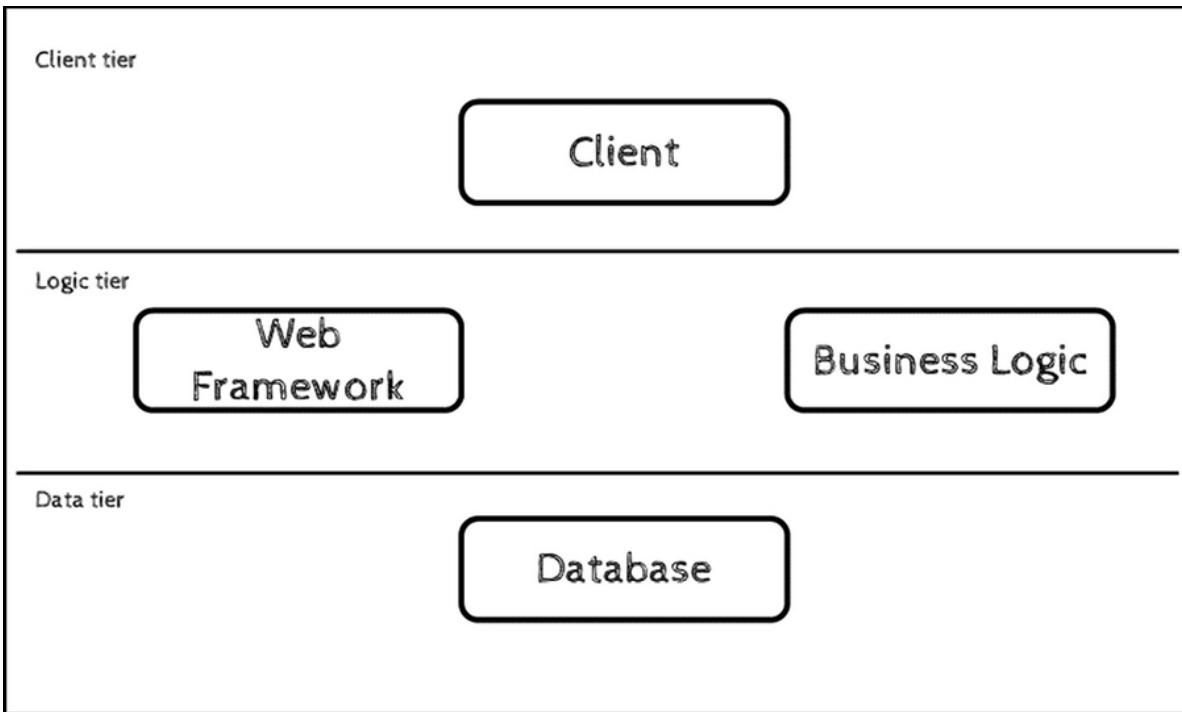


Figura 1.7 Rappresentazione architettura web 3-tier.

Master-slave pattern

Questo modello, come si intuisce dal nome, è composto da componenti di tipo *master* e di tipo *slave*. Il componente master distribuisce il lavoro tra più componenti slave identici e dai risultati restituiti dagli slave calcola un risultato finale.

Il modello master-slave viene spesso utilizzato per *applicazioni multi-thread*, in cui è necessario risolvere più istanze dello stesso problema. Il master crea e lancia gli slave per risolvere queste istanze “in parallelo”. Quando tutti gli slave hanno terminato il proprio lavoro, il master raccoglie i risultati e li elabora. Viene anche utilizzato nella replica del database, dove il database master è considerato la fonte autorevole e i database slave sono sincronizzati con esso.

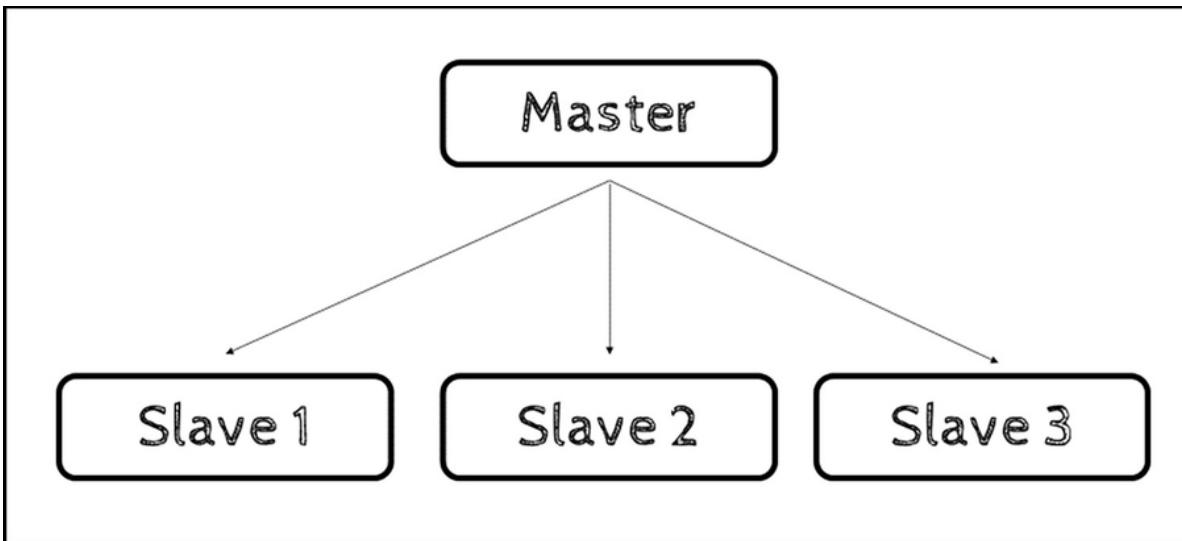


Figura 1.8 Rappresentazione del pattern master-slave.

Broker pattern

Questo modello viene utilizzato per strutturare sistemi distribuiti con componenti disaccoppiati; questi componenti possono infatti interagire tra loro tramite invocazioni a servizi remoti. Un *componente broker* è responsabile del coordinamento della comunicazione tra i componenti. Per esempio, dei server pubblicano i propri servizi su un broker; i client richiedono un servizio al broker; il broker reindirizza il client al servizio adeguato, secondo quanto presente nel proprio registro.

Tre componenti principali sono coinvolti in un sistema broker: il *broker*, il *server* e il *client*, come nella Figura 1.9.

Il broker è un componente di routing dei messaggi, all'interno di un dato sistema: passa i messaggi dal client al server e dal server al client, gestendo in prima battuta la giusta direzione del messaggio. Questi messaggi sono richieste di servizi e risposte a tali richieste, nonché messaggi relativi alle eccezioni che si sono verificate. Spesso le richieste sono codificate come chiamate all'API del broker, motivo per cui il broker è anche responsabile della gestione degli errori in risposta a queste chiamate.

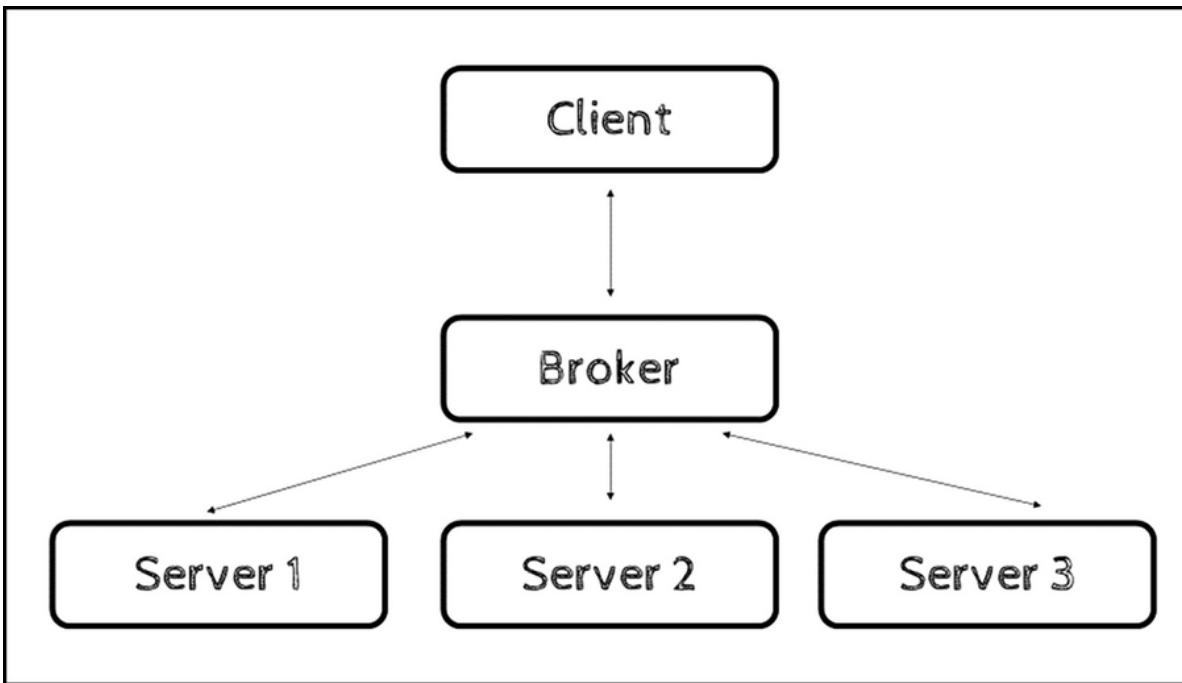


Figura 1.9 Rappresentazione del pattern broker.

Model-view-controller pattern

Questo modello, spesso abbreviato in MVC, divide un'applicazione interattiva in tre parti ben distinte.

- *Model*: contiene le funzionalità e i dati principali.
- *View*: mostra le informazioni all'utente (è possibile definire più view).
- *Controller*: gestisce l'input dell'utente.

In altre parole:

- il *model* contiene solo i dati dell'applicazione, non contiene alcuna logica che descriva come presentare i dati a un utente;
- la *view* presenta i dati del modello all'utente; è una vista e sa come accedere ai dati del modello, ma non conosce il significato di questi dati o che cosa l'utente possa fare per manipolarli;

- il *controller* si frappone tra la view e il model; attende gli eventi attivati dalla view ed esegue l'azione appropriata a questi eventi; nella maggior parte dei casi, si tratta di richiamare sul model un metodo che restituisca informazioni, e quindi è possibile affermare che il controller gestisca la *business logic* del sistema.

Questo consente di separare le rappresentazioni interne delle informazioni dal modo in cui le informazioni vengono presentate e accettate dall'utente. Disaccoppia i componenti e consente un riutilizzo efficiente del codice.

Il *modello MVC* viene comunemente utilizzato per lo sviluppo di interfacce utente moderne. Fornisce i pezzi fondamentali per la progettazione di un programma desktop o mobile, nonché per le applicazioni web. Funziona bene con la programmazione a oggetti, poiché i diversi model, view e controller possono essere trattati come oggetti e riutilizzati all'interno di un'applicazione.

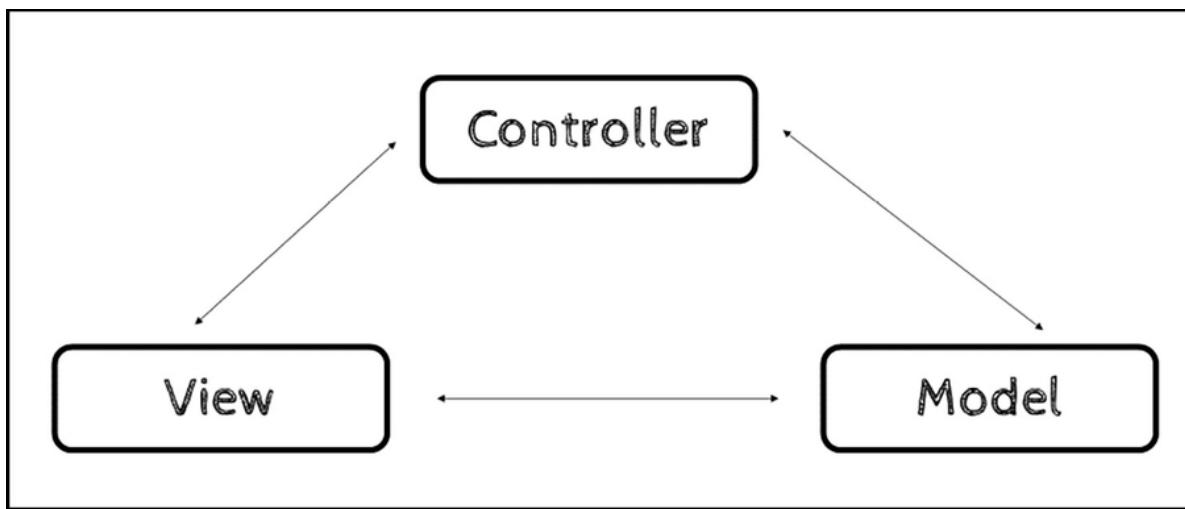


Figura 1.10 Rappresentazione del modello MVC.

Pipe and Filter

Questo tipo di pattern lavora con i flussi di dati e aiuta a strutturare sistemi (o componenti) che devono effettuare l'elaborazione di flussi di dati, ossia che devono trasformare un flusso di dati in ingresso in un

flusso di dati in uscita; per esempio, trasformare una lista di ordini in una lista di spedizioni, gestire collezioni di polizze assicurative e relativi rimborsi.

Un filtro, in generale, effettua un'elaborazione o una trasformazione - e non necessariamente una selezione - dei dati che vi transitano e, per questo, è probabilmente meglio pensare a un filtro come a un componente che effettua un'elaborazione.

Ogni filtro espone un'interfaccia molto semplice: riceve un messaggio sulla *pipe* in entrata, elabora il messaggio e pubblica i risultati sulla *pipe* in uscita. La pipe collega un filtro al successivo, inviando messaggi di output da un filtro all'altro. Poiché tutti i componenti utilizzano la stessa interfaccia esterna, possono essere realizzati con soluzioni differenti, collegando i componenti a diverse pipe. Possiamo aggiungere nuovi filtri, omettere quelli esistenti o riordinarli in una nuova sequenza, il tutto senza dover cambiare i filtri stessi. La connessione tra filtro e pipe viene talvolta chiamata *porta*. Nella forma di base, ciascun componente del filtro ha una porta di input e una porta di output. La *pipeline* definisce il compito complessivo dell'elaborazione, modellando il flusso di dati e la relativa trasformazione.

Per esempio, supponiamo che arrivi un nuovo ordine sotto forma di un messaggio. Un requisito di sicurezza potrebbe essere che il messaggio venga crittografato, per impedire a possibili hacker di spiare gli ordini di un cliente durante la comunicazione con il sistema. Inoltre, è possibile che vengano inviati messaggi duplicati dai vari utenti, che sbagliando fanno clic più volte su un pulsante. Per evitare spedizioni duplicate e clienti insoddisfatti, è necessario eliminare i messaggi duplicati prima di iniziare le successive fasi di elaborazione degli ordini. Per soddisfare questi requisiti, occorre trasformare i messaggi eventualmente duplicati e crittografati contenenti dati relativi l'ordine in un flusso di messaggi.

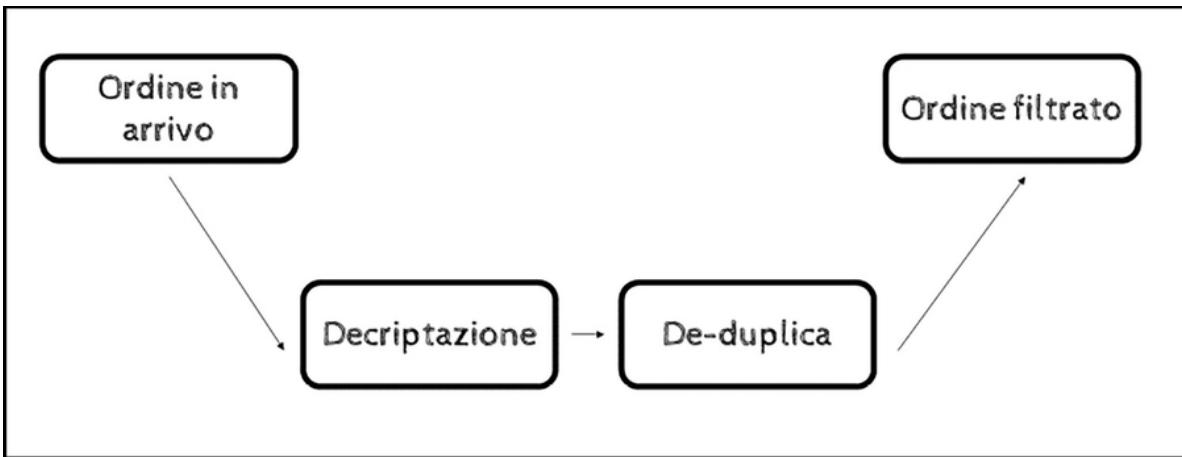


Figura 1.11 Rappresentazione del flusso di elaborazione con pipe e filtri.

Questo pattern fornisce un metodo efficace per una decomposizione dell'elaborazione di un dato compito in passi distinti, implementati tramite componenti a filtro collegati da una pipeline. Sostiene un approccio incrementale, dove ogni filtro esegue il proprio compito, collaborando all'elaborazione complessiva.

Conclusioni

In sintesi, l'architettura, in generale, riguarda le qualità e le strutture di grandi sistemi software; contribuisce al raggiungimento delle qualità desiderate dalle parti interessate al sistema. Si tratta quindi di un'astrazione del sistema, che ne enfatizza gli elementi e le loro interazioni: un elemento dell'architettura è un componente fondamentale del sistema. Gli elementi di interesse per un'architettura possono essere di varia natura: software - per esempio un modulo, un processo, un componente EJB o .NET, un web service o una base di dati - oppure elementi non-software - per esempio un'unità di deployment oppure un team di sviluppo.

Dal momento che in questo libro andremo a trattare soprattutto software, è comune distinguere tra due tipi principali di questo tipo di elementi, parzialmente già definiti nei paragrafi precedenti: i *componenti*, ossia gli elementi responsabili dell'implementazione di funzionalità e della gestione di dati, e i *connettori*, ovvero gli elementi atti alle interazioni tra componenti, che ne caratterizzano l'integrazione. Le

responsabilità per alcune qualità del sistema possono essere attribuite ai connettori; inoltre, questi sono riutilizzabili in più contesti, sotto forma di middleware.

L'architettura di un sistema software è l'insieme delle strutture del sistema che comprendono gli elementi software, le relazioni tra di essi e le loro proprietà necessarie per ragionare sul sistema. In generale, la misura della qualità di un'architettura indica la definizione delle caratteristiche o delle proprietà di un processo o di un progetto rispetto a quanto ci si attende in termini di resa o risultati di una determinata attività.

Alcuni attributi in termini di qualità sono rappresentabili nei seguenti termini.

- *Prestazioni* (ossia, *performance*): la capacità del sistema di operare in modo prevedibile entro il profilo di prestazioni richiesto.
- *Scalabilità*: la capacità del sistema di rispondere a diverse variazioni del carico di lavoro.
- *Robustezza*: la capacità del sistema di essere flessibile a fronte di cambiamenti anche dopo il rilascio della soluzione, mantenendo costi costanti.
- *Sicurezza*: la capacità del sistema di garantire utilizzi propri del sistema.
- *Affidabilità*: in generale, il livello con cui l'utente può fare affidamento sulle funzionalità del sistema per fruire dei servizi forniti.
- *Alta disponibilità (high availability, HA)*: la capacità di un sistema di essere completamente o parzialmente funzionante come e quando richiesto, anche a fronte di guasti di componenti del sistema.
- *Resilienza* (definita anche *fault tolerance*): la capacità del sistema di operare come richiesto, anche a fronte di guasti di componenti hardware o software del sistema.
 - Con conseguente *capacità di recupero*, che riguarda il ripristino del sistema dai fallimenti, ovvero, la capacità di recuperare i dati e di rendere il sistema nuovamente operativo dopo un suo fallimento, entro tempi predefiniti e accettabili.
- *Time to market*: il tempo di realizzazione di un sistema o di un nuovo servizio; una qualità importante soprattutto in presenza di

pressioni competitive nel mercato o di finestre temporali stringenti circa l'opportunità di un sistema o servizio.

- Ultima, ma non meno importante: la *semplicità*. La definizione dell'architettura software è la prima fase nella creazione di un sistema in cui vengono presi in considerazione diversi fattori, tra cui la semplicità garantisce quasi sempre una soluzione a prova di risultato.

In genere, questi attributi non possono essere raggiunti in totale isolamento: ottenere una qualità può avere un effetto negativo (o positivo) su un'altra, pertanto, la progettazione dell'architettura deve considerare con attenzione tutti i possibili compromessi, e le conseguenze di ciascuna possibile decisione di progetto.

Alcuni di questi aspetti potrebbero risultare familiari, per esperienza personale o perché citati nella prima parte di questo capitolo; sono tutti concetti che fanno da fondamenta nello sviluppo che ha portato alla nascita di Docker e che oggi lo rendono uno strumento così valido.

In conclusione, qualunque sia il tipo di attività che andremo a svolgere, le considerazioni fatte in termini decisionali di architettura sono fondamentali prima di mettere in pratica qualsiasi tipo di progetto.

Teniamolo bene a mente!

Installazione di Docker

L'installazione di Docker è semplice e veloce. Docker è attualmente supportato su diverse piattaforme Linux, incluse Ubuntu e Red Hat Enterprise Linux (RHEL). Sono supportate anche varie derivate e distribuzioni correlate, come Debian, CentOS, Fedora, Oracle Linux e molte altre.

Attualmente, il team Docker consiglia di distribuire Docker su Ubuntu, Debian o la famiglia RHEL (CentOS, Fedora e così via), ma in questo capitolo vedremo come installare Docker su diverse piattaforme, non solo Unix-based. Tutte le istruzioni riportate di seguito valgono anche per l'installazione di Docker su un ambiente virtuale; a quel punto, un OS basato su Linux di quelli sopracitati sarebbe la scelta più adatta.

Nel caso dei sistemi operativi Windows e macOS, utilizzeremo dei wrapper sviluppati direttamente dalla community (rispettivamente, *Docker for Windows* e *Docker for Mac*) che permettono di installare Docker come un normale programma, andando a creare una sorta di container che lo esegue.

Per tutti questi tipi di installazione, Docker presenta alcuni prerequisiti di base. Per usare Docker è necessario:

- un sistema basato su un'architettura a 64 bit (attualmente solo `x86_64` e `amd64`); le architetture a 32-bit *non* sono attualmente supportate;
- eseguire un kernel Linux 3.10 o successivo; alcuni kernel precedenti eseguiranno comunque correttamente Docker; i risultati possono variare notevolmente; tuttavia, in caso di problemi, verrà spesso fatto riferimento a un aggiornamento del kernel.

Installare Docker su Ubuntu/Debian

Innanzitutto, controlliamo di avere un kernel Linux sufficientemente recente. Possiamo farlo usando il comando `uname`.

Listato 2.1 Controllo della versione di kernel installato.

```
$ uname -a  
Linux server 4.15.0-74-generic #84 Ubuntu SMP Thu Dec 19 08:05:28 UTC 2019 x86_64  
x86_64 GNU/Linux
```

La versione del kernel è la 4.15.0-74, quindi sicuramente successiva alla 3.10. Verifichiamo che la macchina sia aggiornata e poi procediamo al passo successivo.

Listato 2.2 Verifica e installazione degli aggiornamenti di sistema.

```
$ sudo apt-get update && sudo apt-get upgrade
```

Ora abbiamo tutto il necessario per aggiungere Docker al nostro sistema. Per installare Docker, utilizzeremo i pacchetti DEB del team Docker. Innanzitutto, dobbiamo installare alcuni pacchetti prerequisiti e aggiungere la chiave ufficiale del prodotto.

Listato 2.3 Installazione dei requisiti di Docker.

```
$ sudo apt-get install \  
apt-transport-https \  
ca-certificates \  
curl \  
gnupg-agent \  
software-properties-common
```

Listato 2.4 Aggiunta della chiave GPG.

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

NOTA

Il comando `apt-key` fa parte di SecureApt, uno strumento di crittografia avanzata per convalidare i pacchetti scaricati: `apt-key` è infatti un programma utilizzato per gestire un keyring di chiavi gpg per apt sicuro, mentre GPG è lo strumento usato in apt sicuro per firmare i file e verificarne le firme. Questo ci assicura che il software Docker sia firmato e certificato come fonte sicura.

Utilizzate il comando seguente per configurare il repository Docker e controllare la presenza di aggiornamenti.

*****ebook converter DEMO

Watermarks*****

Listato 2.5 Aggiunta del repository.

```
$ sudo add-apt-repository \
  "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) \
  stable"
```

Listato 2.6 Installazione degli aggiornamenti.

```
$ sudo apt-get update && sudo apt-get upgrade
```

Ora è finalmente possibile procedere con l'installazione di Docker nella versione *Community*. Lanciamo il comando seguente e poi verifichiamo che l'installazione sia andata a buon fine, utilizzando il comando `docker info`.

Listato 2.7 Installazione di Docker.

```
$ sudo apt-get install docker-ce
```

Listato 2.8 Verifica della corretta installazione di Docker.

```
$ sudo docker info
```

Se l'installazione è andata a buon fine, dovremmo ottenere come output del secondo comando le indicazioni relative al numero di container e immagini presenti sul sistema, per il momento zero.

Installare Docker su Windows

Se stai utilizzando Windows, puoi iniziare rapidamente a utilizzare Docker installando Docker per Windows: si tratta di una raccolta di componenti Docker tra cui una piccola macchina virtuale Hyper-V con uno strumento a riga di comando di supporto che è installato su un sistema Windows e che fornisce un ambiente Docker.

Docker per Windows viene infatti fornito in diversi componenti:

- il client e il server Docker;
- Docker Compose (ne ripareremo nel Capitolo 8);
- Docker Machine, che consente di creare host Docker;

- Kitematic, una GUI che consente di eseguire Docker localmente e interagire con Docker Hub.

Docker per Windows richiede Windows 10 Pro, Enterprise o Education a 64 bit e Microsoft Hyper-V. Se il sistema non soddisfa questi requisiti, è possibile installare Docker Toolbox, che utilizza Oracle Virtual Box anziché Hyper-V.

Per installare Docker per Windows è necessario scaricare il programma di installazione da <https://download.docker.com/win/stable/InstallDocker.msi>, avviare il programma di installazione scaricato e seguire le istruzioni per installare Docker. Poiché l'installazione andrà a modificare le variabili d'ambiente, probabilmente sarà necessario riavviare Windows una volta terminata l'esecuzione.

Al termine dell'installazione, dobbiamo verificare che l'installazione di Docker funzioni correttamente, provando a connettere il nostro client locale al daemon Docker in esecuzione sulla macchina virtuale. Per assicurarci che l'applicazione Docker sia in esecuzione, apriamo una finestra del terminale ed eseguiamo il codice seguente.

Listato 2.9 Verifica della corretta installazione di Docker.

```
$ docker info
```

Se l'installazione è andata a buon fine, dovremmo vedere come risultato le indicazioni relative al numero di container e immagini presenti sul sistema, al momento, zero. Sarà possibile aprire il programma Docker anche tramite l'elenco di applicazioni di sistema.

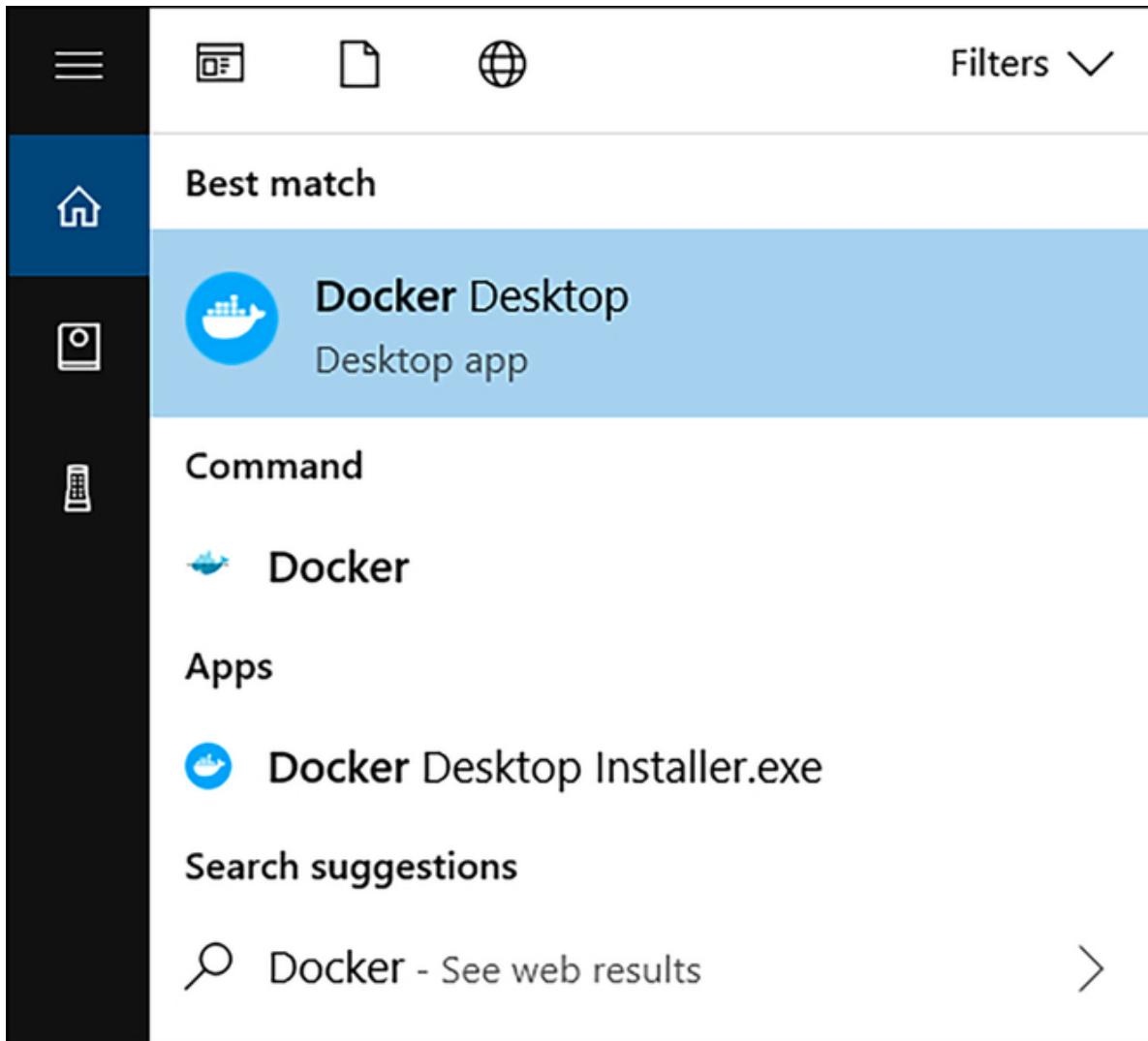


Figura 2.1 Applicazione Docker su Windows.

D'ora in poi, seguendo gli esempi in questo libro, verrà talvolta chiesto di usare i volumi o di eseguire il comando `docker` con una serie di parametri; potrebbe non funzionare correttamente su Windows, dal momento che la gestione del file system tra un sistema Unix-based e Windows è ben diversa. Si consiglia infatti di lavorare con gli esempi che vedremo su un sistema basato su Linux per prendervi maggiore familiarità, e poi passare, se necessario, ad altri sistemi operativi.

Installare Docker su macOS

Così come per Windows, per i sistemi macOS dobbiamo scaricare il suo programma di installazione. Verifichiamo però prima di avere i seguenti requisiti.

- Il Mac deve essere un modello 2010 o successivo, con il supporto hardware Intel per la virtualizzazione dell’unità di gestione della memoria (MMU), comprese le tavole di pagine estese (EPT) e la modalità senza restrizioni. Puoi verificare se la tua macchina ha questo supporto lanciando il seguente comando nel terminale: `sysctl kern.hv_support`.
- Se il tuo Mac supporta il framework Hypervisor, il comando precedente restituirà alla riga dove si trova `kern.hv_support` il valore `1`.
- macOS deve essere versione 10.13 o successiva, ossia Catalina, Mojave o High Sierra. Ovviamente si raccomanda l’aggiornamento all’ultima versione di macOS.
- Almeno 4 GB di RAM.
- VirtualBox precedente alla versione 4.3.30 *non* deve essere installato, in quanto non compatibile con Docker Desktop.

Poiché le nuove versioni principali di macOS sono rese generalmente disponibili a tutti, Docker smette di supportare la versione più vecchia e supporta la versione più recente di macOS (più le due versioni precedenti).

Se questi requisiti sono soddisfatti, è possibile scaricare la versione più recente di Docker for Desktop dall’URL

<https://hub.docker.com/editions/community/docker-ce-desktop-mac> e lanciare l’installazione seguendo le istruzioni.

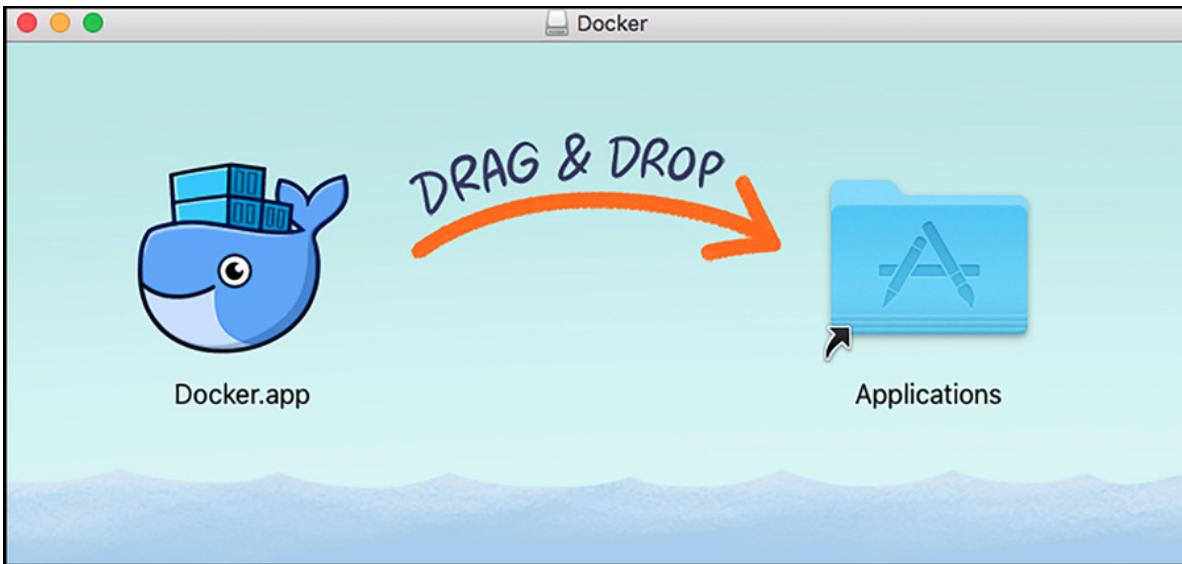


Figura 2.2 Installazione di Docker su macOS.

Al termine dell’installazione, dobbiamo verificare che l’installazione Docker per macOS funzioni correttamente, provando a connettere il nostro client locale al daemon Docker in esecuzione sulla macchina virtuale. Per assicurarci che l’applicazione Docker sia in esecuzione, basta aprire una finestra del terminale e lanciare il comando seguente.

Listato 2.10 Verifica della corretta installazione di Docker.

```
$ docker info
```

Se l’installazione è andata a buon fine, dovremmo vedere come risultato le indicazioni relative al numero di container e immagini presenti sul sistema, al momento, zero.

Installare Docker su Raspberry Pi

L’hardware di cui ha bisogno Docker è un Raspberry Pi 1 o 2 o successivi e una scheda SD. Avendo a disposizione un Raspberry 2, sarà necessario utilizzare una scheda microSD, altrimenti è sufficiente una normale scheda SD. Si consiglia in entrambi i casi di impiegarne una da almeno 4 GB.

Dal momento che il sistema operativo su cui si basa Raspian è Unix-based, sarà sufficiente lanciare il seguente comando per installare il client

Docker su Raspberry Pi con un solo comando da terminale. *Et voilà! Les jeux sont faits!*

Listato 2.11 Installazione di Docker.

```
$ curl -sSL https://get.docker.com | sh
```

Installare Docker su RHEL/CentOS/Fedora

Il procedimento di installazione su un sistema di questo tipo è molto simile all'installazione su Ubuntu. Cambia però il gestore dei pacchetti.

Per iniziare a utilizzare Docker su un sistema di questo tipo, apriamo il terminale del sistema che intendiamo utilizzare, se in locale, oppure colleghiamoci tramite una sessione SSH; a questo punto, digitiamo il seguente comando, per installare Docker.

Listato 2.12 Installazione di Docker.

```
$ sudo yum install docker
```

Per installare l'ultima versione disponibile di Docker nella versione Community, lanciamo i seguenti comandi.

Listato 2.13 Installazione di Docker CE.

```
$ sudo yum remove docker docker-common docker-selinux docker-engine  
$ sudo yum-config-manager --add-repo  
https://download.docker.com/linux/centos/docker-ce.repo  
$ sudo yum install docker-ce
```

In questo caso, aggiungiamo e abilitiamo il repository Docker per poter scaricare questa distribuzione, dal momento che i file non sono direttamente disponibili nei repository ufficiali.

Una volta installato, abilitiamo il servizio Docker tramite il seguente comando.

Listato 2.14 Abilitare il servizio Docker.

```
$ sudo systemctl enable docker.service
```

Per gestire il servizio, possiamo utilizzare l'utility `systemctl` e i relativi sottocomandi `start`, `stop`, `restart` o `status`, che ci permettono di avviare, arrestare, riavviare o vedere lo stato del servizio.

Listato 2.15 Gestire il servizio Docker.

```
$ sudo systemctl start docker.service ## <-- Avvio di docker ##  
$ sudo systemctl stop docker.service ## <-- Arresto di docker ##  
$ sudo systemctl restart docker.service ## <-- Riavvio di docker ##  
$ sudo systemctl status docker.service ## <-- Stato del servizio docker ##
```

Lanciamo il comando seguente e verifichiamo che l'installazione sia andata a buon fine, utilizzando il comando `docker info`.

Listato 2.16 Verifica della corretta installazione di Docker.

```
$ sudo docker info
```

Se l'installazione è andata a buon fine, dovremmo vedere come output del secondo comando le indicazioni relative al numero di container e immagini presenti sul sistema, al momento, zero.

Che cosa abbiamo imparato

- I requisiti minimi per installare Docker sul sistema.
- Come installare Docker Desktop su Windows e macOS.
- Come installare Docker su Ubuntu, e su sistemi basati su Red Hat, CentOS e Raspberry Pi.

Concetti preliminari

Prima di cominciare questo viaggio all'interno del mondo Docker, è necessario porre alcuni mattoni essenziali per proseguire nella costruzione delle nostre competenze; se vogliamo davvero apprezzare quanto promesso da un sistema come Docker, allora dobbiamo partire dalle origini e cercare di vedere perché, rispetto ai tradizionali sistemi di virtualizzazione, questa tecnologia ha impiegato così poco tempo ad affermarsi e in che modo possiamo sfruttarla al meglio.

Docker vs VM

Sia le VM sia i container possono aiutare a ottenere il massimo dalle risorse hardware e software del computer. I container sono i nuovi arrivati, ma le macchine virtuali sono state, e continuano a essere, estremamente popolari nei *data center* di tutte le dimensioni.

Se stiamo cercando la soluzione migliore per eseguire dei servizi, dobbiamo capire queste tecnologie di virtualizzazione, come si confrontano tra loro e quali sono i loro usi migliori.

Una *macchina virtuale* (VM) è l'emulazione di un sistema informatico; in altre parole, consente di eseguire sullo stesso hardware quelli che sembrano essere molti sistemi separati, anche se in realtà si tratta di un unico sistema.

I sistemi operativi e le loro applicazioni condividono risorse hardware da un singolo server host o da un pool di server host. Ogni VM richiede il proprio sistema operativo sottostante e l'hardware è virtualizzato. Un *hypervisor*, o monitor, di una macchina virtuale è un software, un firmware o un hardware che crea ed esegue le macchine virtuali; si trova

tra l'hardware e la macchina virtuale, come indicato nella Figura 3.1, ed è necessario per virtualizzare il server.

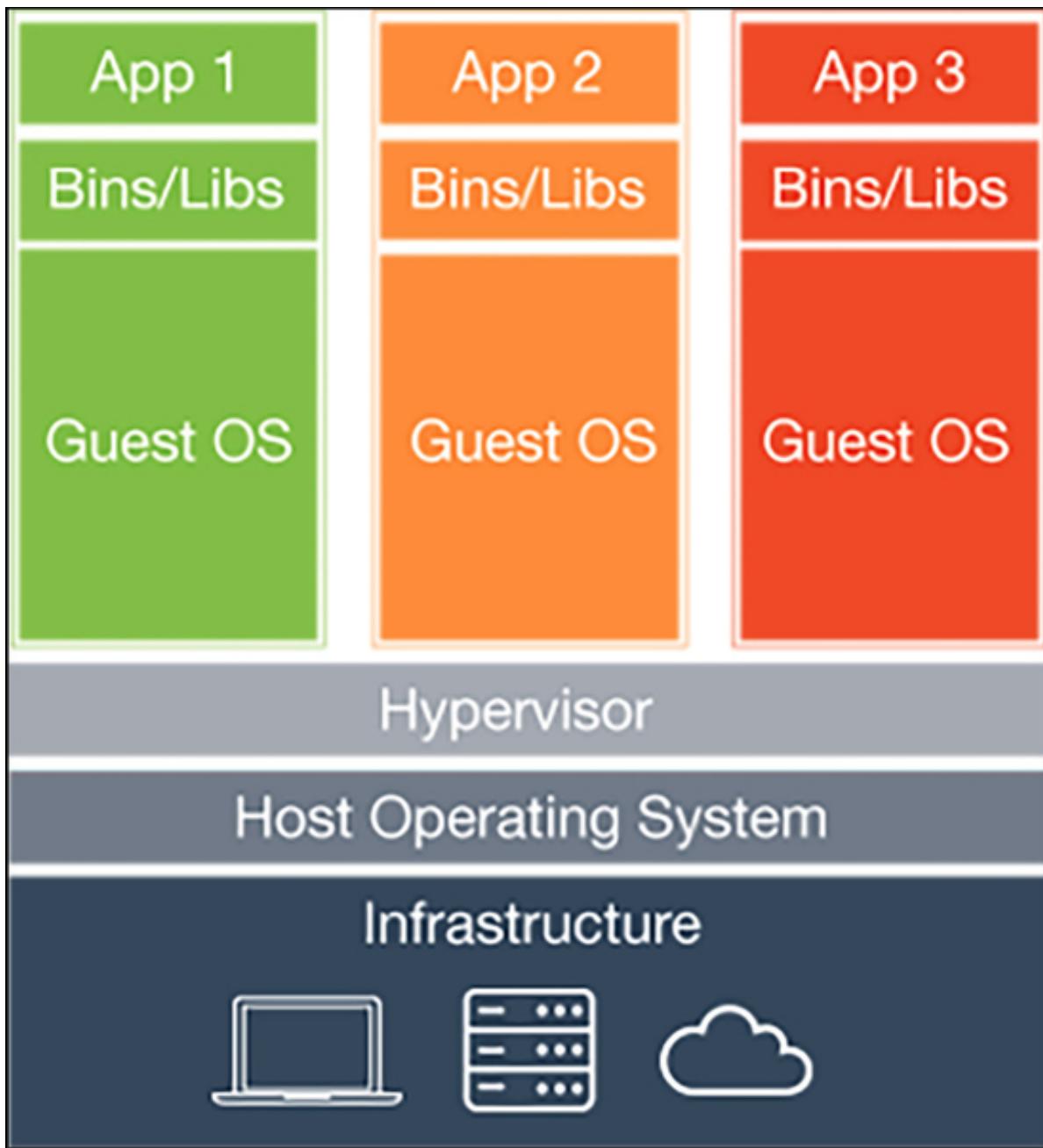


Figura 3.1 La struttura delle VM.

Le macchine virtuali, tuttavia, possono richiedere molte risorse di sistema. Ogni VM esegue non solo una copia completa di un sistema operativo, ma una copia virtuale di tutto l'hardware che il sistema operativo deve impiegare. Ciò aggiunge rapidamente molti cicli di RAM

e CPU; è comunque più economico rispetto all'esecuzione di computer realmente separati, ma per alcune applicazioni può essere eccessivo, il che ha portato allo sviluppo del concetto di container.

Con i *container*, al contrario, invece di virtualizzare il computer sottostante come una macchina virtuale, viene virtualizzato solo il sistema operativo.

I container si collocano sopra un server fisico e il suo sistema operativo host, in genere Linux o Windows. Ogni container condivide il kernel del sistema operativo host e, di solito, anche gli eseguibili e le librerie. I componenti condivisi sono di sola lettura; la condivisione delle risorse del sistema operativo, come le librerie, riduce significativamente la necessità di riprodurre il codice del sistema operativo e fa sì che un server possa sopportare più carichi di lavoro con un'unica installazione. I container sono quindi eccezionalmente leggeri: hanno dimensioni di pochi megabyte e impiegano solo alcuni secondi per avviarsi. Rispetto ai container, le macchine virtuali impiegano diversi minuti per essere operative e hanno un ingombro molto maggiore rispetto a un container equivalente, anche di diversi gigabyte.

I container sono un'astrazione a livello di app che raggruppa codice e dipendenze insieme. Più container possono essere eseguiti sullo stesso computer e possono condividere il proprio kernel del sistema operativo con altri container, tutti in esecuzione come processi isolati nello spazio utente.

Docker non ha niente di magico: ha bisogno di hardware di base su cui funzionare, quindi il primo livello è comunque comune all'infrastruttura prevista anche nelle macchine virtuali.

Oltre a ciò, c'è un sistema operativo host: può essere qualsiasi sistema operativo in grado di eseguire Docker e, come abbiamo visto nel capitolo precedente, Docker supporta (quasi) ogni tipo di sistema operativo.

Seguendo la Figura 3.2, questi primi due livelli sono le basi della nostra architettura; il terzo livello è costituito dal *Docker Engine*, che occupa il posto del livello Hypervisor nell'architettura a macchine virtuali. Contiene, tra i suoi componenti, il *docker daemon*, un servizio che viene eseguito in background ed è responsabile dell'esecuzione del container Docker.

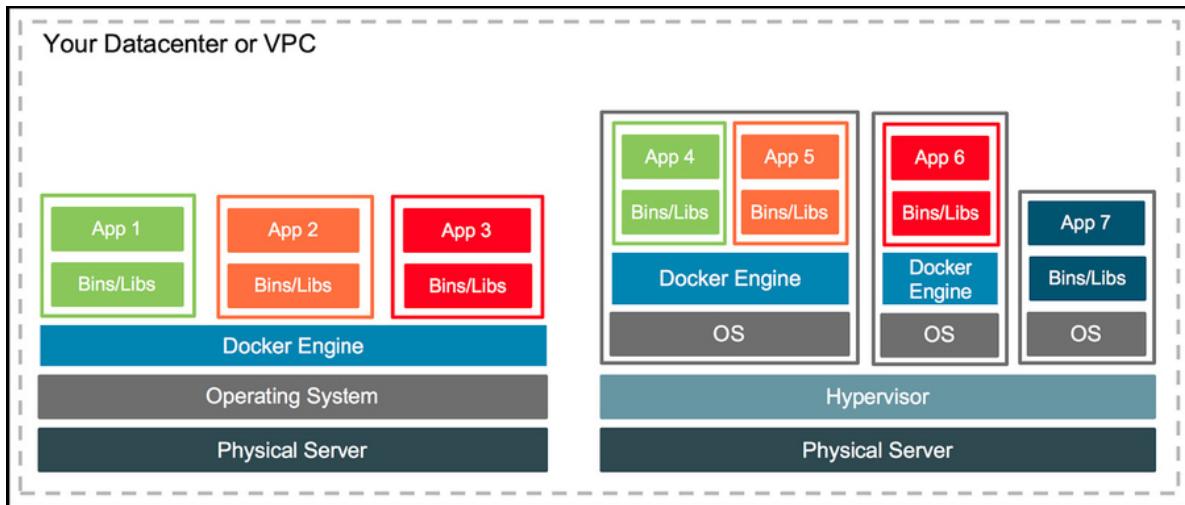


Figura 3.2 VM e Docker a confronto.

Al livello immediatamente superiore troviamo le librerie e gli eseguibili, replicati per ogni container; ancora sopra troviamo il codice sorgente del container, dove risiede quindi la nostra applicazione; il daemon Docker gestisce tutti questi livelli di container.

In questo modo eliminiamo la necessità di un SO guest e così risparmiamo grandi quantità di memoria. Pertanto, i container funzionano un po' come macchine virtuali, ma in un modo molto più specifico e granulare: isolano una singola applicazione e le sue dipendenze - tutte le librerie software esterne che l'app deve eseguire - sia dal sistema operativo sottostante sia dagli altri container. Tutte le app eseguite in un container condividono un unico sistema operativo, ma sono compartmentate tra loro e dal sistema in generale.

Le istanze di app in container utilizzano molta meno memoria rispetto alle macchine virtuali, si avviano e si arrestano più rapidamente e possono essere impacchettate molto più densamente sull'hardware host. Tutto ciò equivale a una minore spesa per l'IT; è chiaro che i risparmi sui costi varieranno a seconda delle applicazioni in gioco e della loro intensità di risorse, ma i container risultano immancabilmente più efficienti delle VM. È anche possibile risparmiare sui costi delle licenze software, poiché per sopportare gli stessi carichi di lavoro sono necessarie molte meno istanze del sistema operativo.

Il software prodotto deve rispondere rapidamente alle mutevoli condizioni imposte dal mercato o dai clienti; ciò implica sia un facile

ridimensionamento sia un facile aggiornamento degli applicativi, per aggiungere nuove funzionalità. I container semplificano la rapida messa in produzione di nuove versioni di software e, se necessario, il *rollback* rapido a una versione precedente.

Poiché i container incapsulano tutto (ma anche solo) ciò che un'applicazione deve eseguire, consentono alle applicazioni di essere spostate facilmente tra gli ambienti. Qualsiasi host con Docker, che si tratti del laptop di uno sviluppatore o di un'istanza di cloud pubblico, può eseguire un container Docker.

Leggeri, portatili e autonomi, i container Docker semplificano la creazione di software lungo linee lungimiranti, in modo da evitare di risolvere i problemi di domani con i metodi di sviluppo di ieri.

Uno dei modelli software che si adatta perfettamente all'uso di Docker è il pattern a microservizi, in cui le applicazioni sono costituite da più componenti disaccoppiati, come abbiamo visto nel precedente capitolo. Infatti, decomponendo le applicazioni tradizionali monolitiche in servizi distinti, i microservizi consentono alle diverse parti di un'app di essere ridimensionate, modificate e revisionate separatamente - da team separati e con tempistiche separate, adatte alle esigenze dell'attività di progetto. I container non sono necessari per implementare i microservizi, ma sono perfettamente adatti all'approccio a microservizi e ai processi di sviluppo agili in generale.

Quando si tratta di confrontare i due approcci, si potrebbe dire che i container Docker hanno molto più potenziale delle macchine virtuali. È evidente come i container siano in grado di condividere il singolo kernel e le librerie delle applicazioni; i container vantano un sovraccarico per il sistema inferiore rispetto alle macchine virtuali, e le prestazioni dell'applicazione all'interno di un container sono generalmente uguali o migliori rispetto a quelle della stessa applicazione in esecuzione in una macchina virtuale.

Non sarebbe corretto confrontare Docker e macchine virtuali e nominare un vincitore assoluto, poiché essi sono destinati a un uso diverso. Docker, senza dubbio, sta guadagnando slancio in questo periodo storico, ma non si può dire che sostituisca le macchine virtuali. Nonostante la crescente popolarità di Docker, in alcuni casi una macchina virtuale è una scelta migliore: sono considerate una scelta adatta in un ambiente di produzione, piuttosto che container Docker, poiché vengono

eseguite sul sistema operativo senza costituire una minaccia per il computer host. Se le applicazioni devono essere sottoposte a test, Docker è la scelta giusta, in quanto Docker offre più piattaforme OS per svolgere test approfonditi del software o dell'applicazione.

Inoltre, non molte aziende digitali fanno affidamento sulle macchine virtuali come scelta principale e preferiscono migrare verso l'utilizzo di container perché la distribuzione è relativamente lunga e l'esecuzione di microservizi è anche una delle maggiori sfide. Tuttavia, vi sono ancora alcune aziende che preferiscono le macchine virtuali rispetto a Docker, mentre le aziende interessate alla sicurezza di livello aziendale per la propria infrastruttura preferiscono utilizzare Docker.

Tabella 3.1 Proprietà di VM e Docker.

VM	Docker
Isolamento a livello hardware	Isolamento a livello software
Ogni macchina virtuale ha il proprio OS	Ogni container può condividere il proprio OS
Il boot avviene in qualche minuto	Il boot avviene in pochi secondi
La portabilità non è semplice, anche per via delle dimensioni	Portabilità semplice
Dimensioni in termini di GB	Dimensioni di qualche MB
Se l'uso delle risorse hardware non è opportunamente gestito, è difficile avere buone performance con più macchine virtuali su uno stesso host	Uso delle risorse minimale

Mentre Docker presenta vantaggi in termini di efficienza e prestazioni, non si può commettere l'errore di pensare che i container siano intrinsecamente migliori delle macchine virtuali. Docker e macchine virtuali hanno specifici vantaggi in determinate situazioni. Determinare se un'applicazione viene inserita in un container o in una macchina virtuale dipende interamente dai requisiti di utilizzo. Nei prossimi paragrafi vedremo come scegliere tra i due.

Tipo di applicazione

Il primo fattore da considerare nella scelta è il tipo di applicazione da distribuire. I container sono progettati per supportare applicazioni che non sono correlate al sistema operativo ospitante. I container Docker sono adatti a quelle applicazioni che si desidera eseguire indipendentemente dall'infrastruttura del sistema host; ciò significa che se si desidera sostenere sistemi integrati che comunicano più strettamente con il dispositivo host, è meglio optare per una macchina virtuale.

Dimensione

Il fattore successivo da prendere in considerazione nella scelta tra i due approcci è la dimensione dell'applicazione. La quantità di risorse di elaborazione necessarie per l'esecuzione efficace dell'applicazione determinerà quale è meglio scegliere. Se stiamo cercando di supportare applicazioni che richiedono solo una macchina per funzionare su un sistema basato a microservizi, un container Docker è la scelta più naturale; al contrario, una macchina virtuale è la scelta migliore per servizi ad alta priorità, come i database.

Tabella 3.2 Scegliere tra VM e Docker.

Caso d'uso	VM	Docker
Host per container	Si	No
Data center	Si	No
Microservizi	No	Si
Applicazioni web	No	Si
App legacy	No	Si

Come possiamo vedere nella Tabella 3.2, le macchine virtuali hanno il vantaggio in due casi d'uso principali: host di ambiente container e database. Le macchine virtuali hanno la capacità di interagire a livello di hardware virtuale e consentono all'utente di apportare modifiche all'infrastruttura. Le macchine virtuali vengono spesso utilizzate come host container a causa della loro capacità di interagire con l'hardware.

D'altra parte, Docker offre un modo leggero per distribuire *app legacy* in quasi tutti gli ambienti. Avere le librerie contenute a fianco

dell'applicazione le rende facili da distribuire; anche i microservizi sono ideali per Docker, perché la semplice struttura dei container si presta bene alle applicazioni che hanno un compito principale (al contrario di un'applicazione con più responsabilità differenti).

Architettura di Docker

L'architettura di Docker segue il modello client-server. Il client Docker comunica con il daemon, che svolge la maggior parte delle attività, come quella di costruzione, esecuzione e distribuzione dei container Docker. Nel complesso, Docker è costituito dai seguenti componenti.

- *Docker Client (CLI)*: una funzione utilizzata per attivare i comandi docker tramite riga di comando, come vedremo nel prossimo capitolo.
- *Docker Host*: comando per eseguire il daemon.
- *Registro*: per l'archiviazione delle immagini docker, come il Docker Hub al quale accenneremo a breve.

Nella Figura 3.3 vediamo come questi elementi interagiscono tra loro.

Gli utenti possono interagire con Docker tramite il client; quando vengono eseguiti dei comandi, il client li invia al daemon `dockerd`, che li esegue. Il client Docker può risiedere sullo stesso host del daemon o connettersi a un daemon su un host remoto; un client docker può comunicare con più daemon. Lo scopo principale del client Docker è quello di fornire un mezzo per gestire l'estrazione di immagini da un registro e per eseguirle su un host Docker. I comandi più comuni che vengono utilizzati tramite client sono per esempio `docker pull`, `docker build` o `docker run`, istruzioni che vedremo in dettaglio nel prossimo capitolo.

L'host Docker fornisce un ambiente completo per costruire ed eseguire applicazioni; comprende il daemon Docker, le immagini, i container, le reti e lo spazio di archiviazione. Il daemon è responsabile di tutte le azioni relative ai container e riceve comandi tramite l'interfaccia a riga di comando o l'API REST. Può anche comunicare con altri daemon per gestire i suoi servizi.

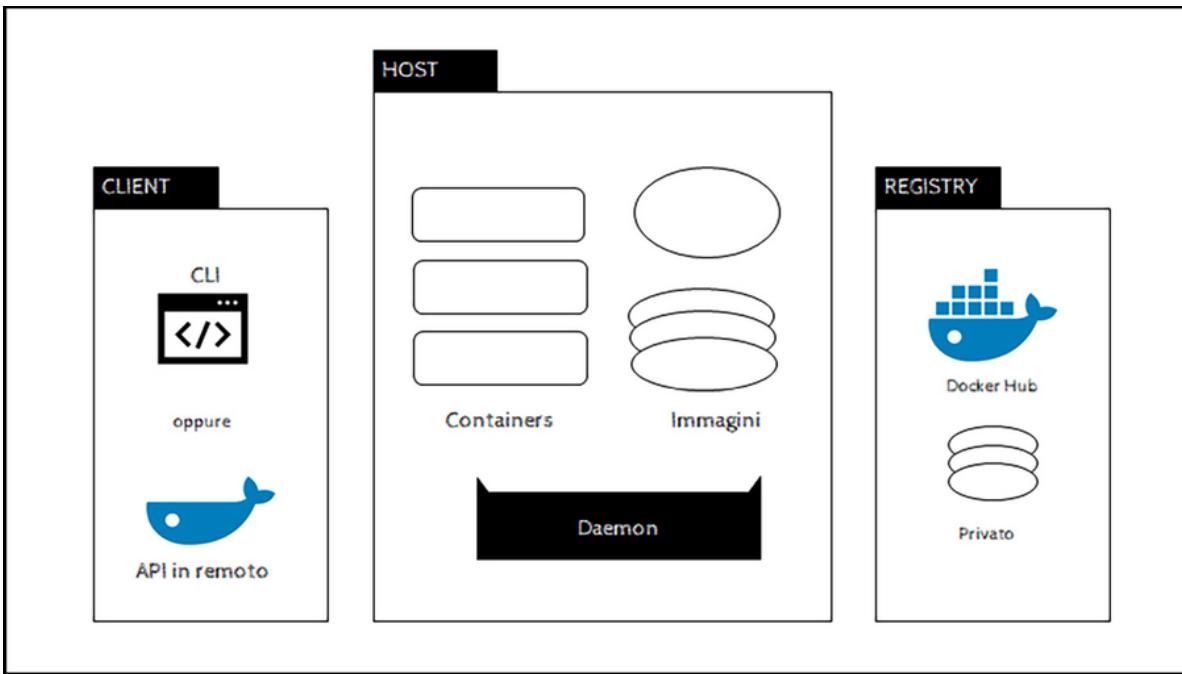


Figura 3.3 Rappresentazione dell'architettura di Docker.

Il daemon Docker estrae e costruisce le immagini del container come richiesto dal client; una volta estratta un'immagine, crea un oggetto funzionante a partire dall'immagine stessa, noto come *container*, utilizzando un set di istruzioni. Possiamo infatti pensare all'immagine come a uno schema di come il container dovrà essere, una volta eseguito; facendo un paragone nell'ambito della programmazione a oggetti, l'immagine è come una classe, mentre il container ne è la sua istanza.

Questi due componenti costituiscono quello che si chiama *Docker Engine*: è il cuore dell'architettura Docker e può essere disaccoppiato o installato su una stessa macchina host.

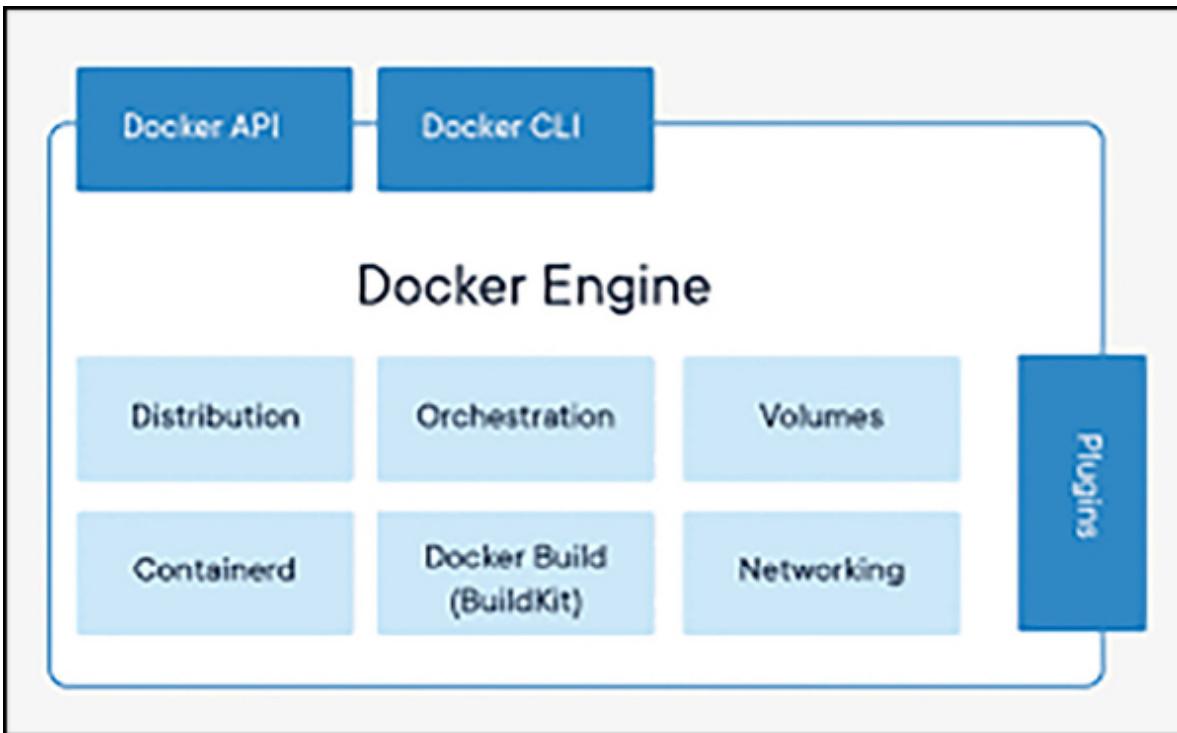


Figura 3.4 Docker Engine.

Infine, il registro è la posizione in cui sono memorizzate le immagini Docker e può essere pubblico o privato; Docker Hub è il luogo predefinito delle immagini Docker ed è un registro pubblico.

Chiaramente, è possibile avere un registro privato in cui mantenere le proprie immagini in locale, senza il bisogno di pubblicarle sul Docker Hub.

Passiamo a questo punto agli oggetti principali che usa Docker: immagini, Dockerfile e container; ognuno di questi ha una funzione ben specifica, che esamineremo in dettaglio con vari esempi pratici.

Immagini

Le immagini sono i mattoni del mondo Docker, che permette di creare e lanciare dei container a partire dalle immagini. Sono basate su un tipo particolare di file che, all'esecuzione, utilizza in modo ben stratificato i file system, che vengono creati passo dopo passo utilizzando una serie di istruzioni, come per esempio:

- aggiungi un file;
- esegui un comando;
- apri una porta.

Possiamo considerare le immagini come il “codice sorgente” dei container. Sono altamente portatili e possono essere condivise, archiviate e aggiornate. In questo manuale, impareremo a utilizzare le immagini esistenti e a costruire nuove immagini.

Abbiamo detto che un’immagine Docker è composta da file system sovrapposti: alla base c’è un file system di tipo UnionFS, che consente di prendere diversi file system e creare un’unione dei loro contenuti con il livello più alto, che sostituisce qualsiasi file simile trovato nei file system.

In Unix tutto è un file; per esempio, a parte i normali file di dati, anche i dispositivi di sistema sono esposti attraverso il *namespace* del file system. Per esempio, se collegassimo un hard disk al nostro computer con un sistema Unix, questo potrebbe essere visto come un file chiamato `sda` nella directory `/dev`, e sarà possibile accedervi tramite il suo percorso assoluto, che è `/dev/sda`.

Anche le partizioni del disco sono viste come file di dispositivo, all’interno di `rootfs`: per esempio `/dev/sda1` è la prima partizione primaria del disco ed è ulteriormente formattata con un driver di file system (etx3, ext4 e così via), in modo che possa archiviare file e directory al suo interno.

Quando un container è stato avviato, viene spostato in memoria e il file system di avvio viene smontato per liberare la RAM utilizzata dall’immagine del disco `initrd`.

Finora sembra un tipico stack di virtualizzazione Linux. I container Docker offrono l’esecuzione di diversi tipi di software sotto forma di livelli. Durante la compilazione, vengono “impilati” più strati di software immutabili, per ottenere le applicazioni desiderate con le loro dipendenze.

In un avvio Linux più tradizionale, il file system di root viene montato in sola lettura e quindi passa alla modalità di lettura o scrittura dopo l’avvio e viene sempre eseguito un controllo di integrità. Nel mondo Docker, al contrario, il file system di root rimane in modalità di sola lettura e Docker si avvale di un `mount` per aggiungere più file system di

sola lettura al file system di root. Una `mount union` è un `mount` che consente di avere più file system montati contemporaneamente, anche se ha l'aspetto di un unico file system.

Le immagini possono essere sovrapposte l'una all'altra: come si può vedere nell'esempio della Figura 3.5: si ha un'immagine principale e una serie di livelli, fino a raggiungere la parte inferiore della pila di immagini, dove l'immagine finale è l'immagine di base.



Figura 3.5 Astrazione interna di un container.

Quando un container viene lanciato da un'immagine, Docker monta un file system di lettura-scrittura sopra tutti i livelli sottostanti. È qui che verranno eseguiti tutti i processi che vogliamo far eseguire dal nostro container Docker.

Immaginando, per esempio, di voler montare un server nginx su una distribuzione Alpine, un'immagine Docker potrebbe essere rappresentata come nella Figura 3.6.

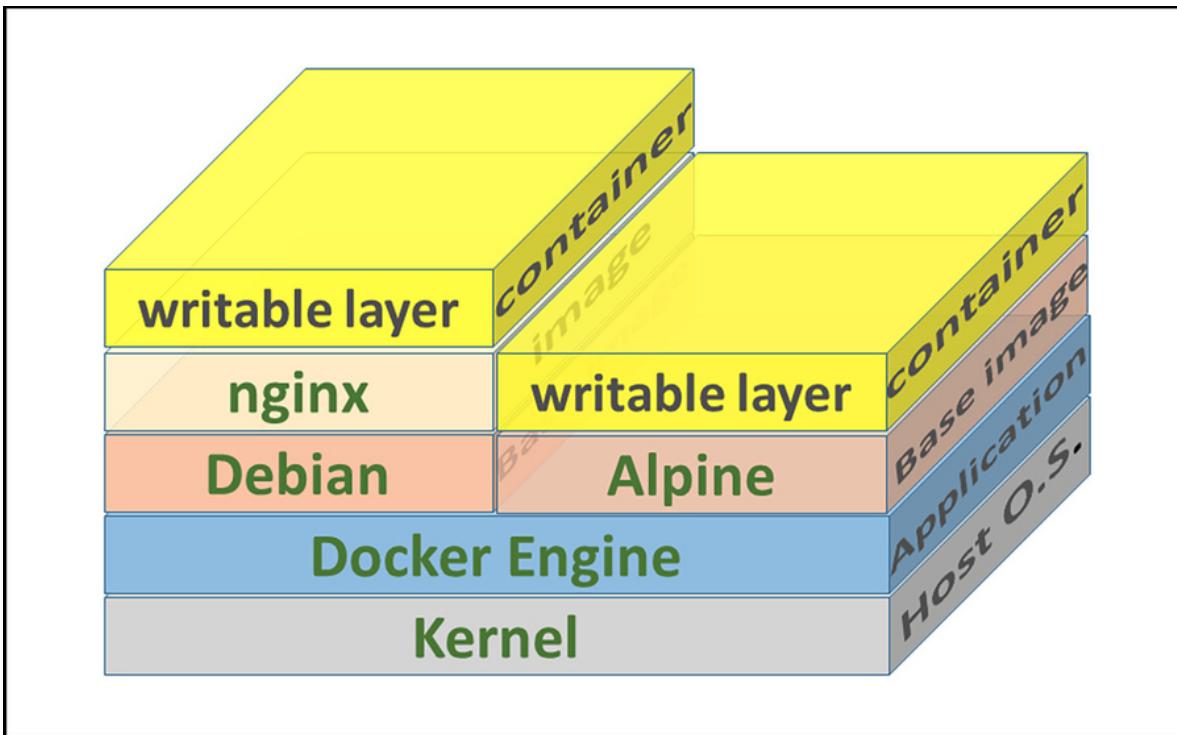


Figura 3.6 Struttura di un’immagine Docker.

Dopo aver aggiunto un livello per il sistema operativo Alpine, viene aggiunto un livello sovrastante per il server nginx, e sopra entrambi i livelli viene posto un livello “scrivibile”, che è il livello con cui possiamo lavorare, copiando file o modificandone le configurazioni.

Abbiamo detto che ogni container Docker si basa su un’immagine e che questa fornisce la base per tutto ciò che verrà distribuito ed eseguito. Per avviare un container, è necessario scaricare un’immagine pubblica o crearne una. Sul Docker Hub (di cui vedremo il funzionamento dettagliato fra qualche capitolo) è possibile trovarne di ogni tipo, a seconda delle esigenze: è importante, qualora decidessimo di scaricarne una, prestare attenzione a quelle ufficiali e a quelle prodotte dalla community.

Le immagini ufficiali Docker sono ospitate su Docker Hub e sono progettate per:

- fornire repository di sistemi operativi di base essenziali (per esempio Ubuntu, oppure CentOS) che fungono da punto di partenza per la maggior parte degli utenti;

- fornire soluzioni per l'esecuzione di più linguaggi di programmazione, database e altri servizi di base, simili a quelli offerti da una “piattaforma come servizio” (PaaS);
- fornire *best practice* su come un Dockerfile dovrebbe essere utilizzato, nonché fornire una documentazione chiara che funga da riferimento per altri autori;
- in ultimo, ma non meno importante, assicurarsi che gli aggiornamenti di sicurezza vengano applicati in modo tempestivo.

Docker, come società, possiede un team dedicato che è direttamente responsabile della revisione e pubblicazione di tutti i contenuti nelle immagini ufficiali.

La creazione e gestione di immagini a partire da quelle ufficiali è ovviamente consentita. I nuovi utenti Docker sono spesso incoraggiati a utilizzare le immagini ufficiali nei loro progetti, perché queste immagini sono progettate per i casi d'uso più comuni. Gli utenti esperti si trovano spesso a rivedere le immagini ufficiali come parte del loro processo di apprendimento nella costruzione di un Dockerfile e a modificarle secondo le loro esigenze.

Dockerfile

Abbiamo nominato più volte il termine Dockerfile: vediamo quindi che cos'è e perché è strettamente correlato al concetto di immagine.

Diamone una definizione precisa: *un Dockerfile è un file di testo che definisce un'immagine Docker*. Utilizzeremo dunque un Dockerfile per creare un'immagine Docker personalizzata oppure ne andremo a scaricare una dal Docker Hub, allo scopo di definire un ambiente personalizzato da utilizzare come container.

Per creare un'immagine Docker personalizzata con gli strumenti predefiniti, è necessario acquisire familiarità con i Dockerfile; abbiamo detto che questo file descrive tutti i passaggi necessari per creare un'immagine e normalmente è contenuto nella directory principale del repository del codice sorgente per l'applicazione.

Un tipico Dockerfile potrebbe essere simile a quello mostrato nel Listato 3.1, che crea un container per un'applicazione REST basata su

Node.js.

Listato 3.1 Esempio di Dockerfile per un'applicazione Node.js.

```
FROM node:13
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 8080
ENTRYPOINT echo "Il container si sta avviando..."

CMD [ "node", "server.js" ]
```

Non è necessario avere familiarità con Node.js per poter comprendere l'esempio soprastante; è utile soffermarsi invece su determinati elementi presenti nel Dockerfile che rappresentano le istruzioni che verranno eseguite alla costruzione del container. Queste istruzioni sono sempre in maiuscolo, all'inizio di ogni riga; ognuna di esse ha un preciso scopo e accetta diversi parametri, come vedremo a breve.

Istruzione FROM

Partiamo dall'istruzione `FROM`: la prima cosa che dobbiamo fare è definire a partire da quale immagine vogliamo costruire il container. Qui utilizzeremo l'ultima versione disponibile dal Docker Hub e la andremo a scaricare tramite l'istruzione `FROM`; questa deve essere sempre la prima istruzione del Dockerfile. Come parametro, richiede un nome e un tag (opzionale) per un'immagine che dovrebbe essere utilizzata come base per il container.

Chi ha già lavorato con Docker, sicuramente si sarà imbattuto nel concetto dei *tag*. Li troveremo spesso nella forma "`nomeimmagine:1`", in cui la parte dopo i due punti è proprio il nostro tag. Non viene sempre specificato quando si taggano le immagini, né è obbligatorio farlo, ma ci arriveremo.

Quando si inizia a utilizzare Docker, i tag sono sicuramente uno degli argomenti che portano più confusione. In poche parole, i tag Docker trasmettono informazioni utili su una specifica versione/variante di una data immagine. Sono alias dell'ID dell'immagine e sono rappresentabili anche tramite una stringa esadecimale come questa: `a5551bf98e78`; in realtà, è solo un modo di fare riferimento all'immagine e per identificarla. Una

buona analogia è il modo in cui i tag tramite Git si riferiscono a un determinato commit nella cronologia.

Ora, è chiaro che specificando una versione, andremo a scaricare l'immagine che è stata rilasciata in quella versione, con le modifiche specificate; nel caso dell'esempio riportato sopra, andremo a scaricare l'immagine Node.js con tag ¹³.

Scopriamo che cosa succede se non specifichiamo un tag quando scarichiamo un'immagine; entra in gioco un altro tag, ossia *latest*: ogni volta che un'immagine viene taggata senza un tag esplicito, per impostazione predefinita viene assegnato questo tag. È una scelta di denominazione sfortunata, che crea molta confusione, anche se possiamo considerarlo come il tag di default che viene dato alle immagini quando non ne viene specificato uno.

Molta confusione è causata dall'aspettativa che si tratti dell'ultima versione dell'immagine, specialmente nei Dockerfile. Consideriamo però i vari scenari con un esempio; supponiamo che nel Dockerfile sia presente la seguente dichiarazione (Listato 3.2).

Listato 3.2 Istruzione FROM: primo scenario.

```
FROM node
```

Poiché non abbiamo specificato alcun tag, Docker aggiungerà il tag *latest* e proverà a estrarre l'immagine `node:latest`.

Invece, nello scenario dell'esempio precedente, andremo a scaricare esattamente l'immagine taggata con ¹³ di Node.js; il principale risultato da ciò che abbiamo trattato finora è che *latest* è proprio come qualsiasi altro tag. È compito dello sviluppatore taggare correttamente le immagini, in modo tale che l'ultima punti sempre all'ultima versione stabile dell'immagine.

Pertanto, non specifichiamo esplicitamente un tag nei nostri Dockerfile durante il pull delle immagini, poiché potremmo finire con una versione completamente diversa dell'immagine base rispetto a quella che avevamo usato prima; usando, per esempio, un'immagine custom, non è detto che il numero del tag corrisponda, per esempio, alla versione del linguaggio di programmazione che intendiamo utilizzare, proprio perché come ultima può essere taggata anche una vecchia versione.

Se creassimo un Dockerfile come il seguente e caricassimo quest'immagine sul DockerHub senza specificare alcun tag e dandogli il

nome `ubuntu` (la cosa non è possibile, ma l'esempio è a puro scopo didattico), qualunque sviluppatore che la andasse a scaricare, a meno che non verifichi il contenuto del Dockerfile, si troverebbe con una versione di Ubuntu vecchia di diversi anni.

Listato 3.3 Istruzione FROM: secondo scenario.

```
FROM ubuntu:14.04
ENV NXLOG_VERSION=2.9.1716
RUN apt-get update && \
    apt-get install -y libapr1 libdbi1 libperl5.18 openssl
...
```

Istruzione WORKDIR

Fatta questa premessa, proseguiamo con la spiegazione dell'esempio riportato all'inizio del paragrafo; come seconda istruzione abbiamo trovato la seguente.

```
WORKDIR /app
```

L'istruzione `WORKDIR` imposta la directory di lavoro per tutte le istruzioni `RUN`, `CMD`, `ENTRYPOINT`, `COPY` e `ADD` che seguono nel Dockerfile. Se il comando `WORKDIR` non è riportato, verrà utilizzata la cartella principale del file system. Nel caso di esempio, la directory di lavoro è impostata su `/app`. Se la directory del progetto non esiste, verrà creata automaticamente e il path di lavoro diventerà quella stessa cartella; se infatti potessimo lanciare il comando `pwd` all'interno della macchina, questo ci restituirebbe proprio `/app`.

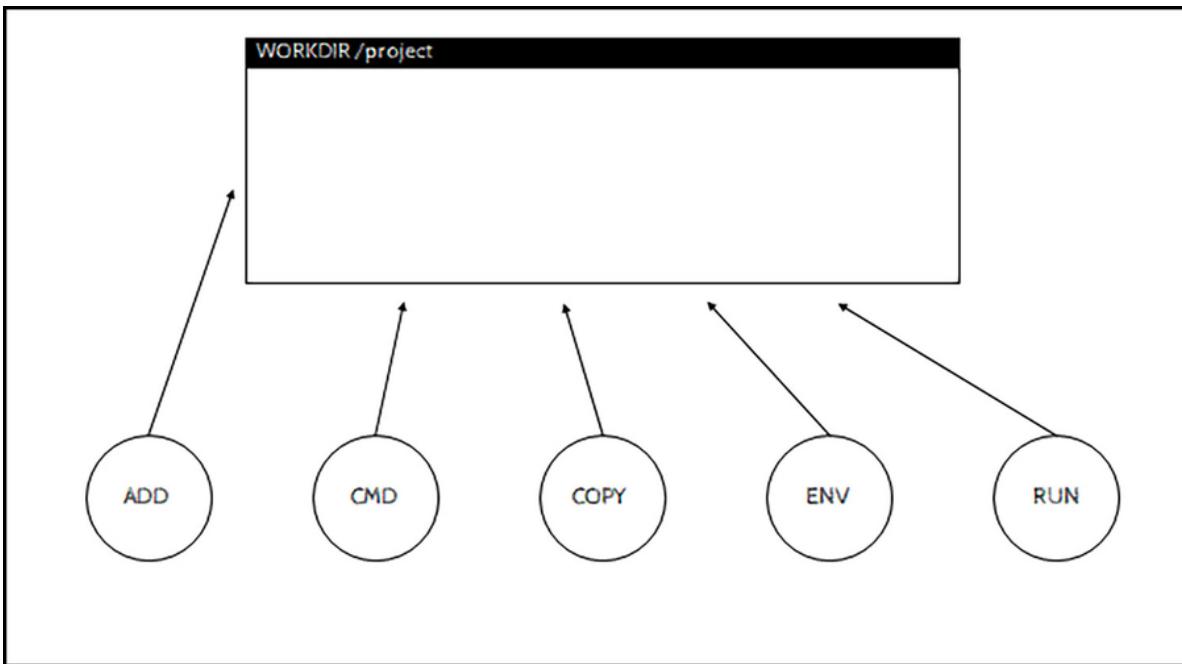


Figura 3.7 Esempio di rappresentazione della cartella di lavoro dopo l'esecuzione di WORKDIR.

È possibile utilizzare più volte il comando `WORKDIR` per impostare una nuova directory di lavoro in qualsiasi fase del Dockerfile. Il percorso della nuova directory di lavoro deve essere specificato in relazione alla directory di lavoro corrente; questo significa che a seconda della cartella da utilizzare, potremmo aver bisogno di tornare a una cartella che si trova allo stesso livello o anche a un livello superiore del file system.

Se viene fornito un percorso relativo, la seconda istruzione `WORKDIR` avrà come percorso relativo quello dell'istruzione `WORKDIR` precedente. Per esempio, nel caso seguente, l'output del comando `pwd` in questo Dockerfile sarebbe `/app/assets/images`.

Listato 3.4 Istruzione WORKDIR utilizzata più volte.

```
WORKDIR /app
WORKDIR assets
WORKDIR images
```

```
RUN pwd
```

Sebbene le directory possano essere create e modificate manualmente, si consiglia vivamente di utilizzare `WORKDIR` per specificare la directory corrente in cui si desidera lavorare, poiché semplifica la risoluzione dei problemi.

Dopo aver creato una directory chiamata `app` per contenere il codice dell'applicazione all'interno dell'immagine e aver definito che questa sarà la directory di lavoro della nostra applicazione, troviamo il comando `COPY`.

Listato 3.5 Esempio di Dockerfile per un'applicazione Node.js.

```
COPY package*.json ./
```

Questa immagine viene fornita con Node.js e NPM già installati, quindi la prossima cosa che dobbiamo fare è installare le dipendenze della nostra, usando il binario `npm`; piuttosto che copiare l'intera directory di lavoro, stiamo solo copiando il file `package.json`, che contiene informazioni sulle librerie da installare.

Il comando `COPY` copia nuovi file o directory da una cartella o un file sorgente e li aggiunge al file system del container nel percorso specificato come destinazione. È possibile specificare più risorse sorgenti, ma devono essere relative alla directory di origine in fase di creazione (il contesto della build). Ogni sorgente può contenere caratteri jolly, come nell'esempio sopra: non verrà copiato solo il file `package.json`, ma anche il file `package-lock.json`, che viene utilizzato esclusivamente per bloccare le dipendenze su un determinato numero di versione.

La destinazione, invece, è un percorso assoluto oppure un percorso relativo a `WORKDIR`, in cui i file specificati verranno copiati all'interno del container di destinazione; se la destinazione non termina con una barra finale, verrà considerato come un comune file e il contenuto specificato come sorgente verrà copiato nel file di destinazione. Se la destinazione non esiste, verrà creata insieme a tutte le directory mancanti nel suo percorso.

Istruzione RUN

L'istruzione successiva è `RUN`.

```
RUN npm install
```

`RUN` esegue i comandi in un nuovo livello e crea una nuova immagine. Per esempio, spesso questo comando è usato per installare pacchetti software; viene creata un'immagine intermedia, che poi viene scartata una volta terminata l'esecuzione dell'attività richiesta.

Per comprendere al meglio le potenzialità di questa istruzione, è doveroso fare un po' di chiarezza sul suo funzionamento e sui modi in cui può essere utilizzata. Quando Docker esegue un container, esegue al suo interno un'immagine. Questa immagine, in genere viene creata eseguendo le istruzioni Docker, che aggiungono livelli sopra l'immagine esistente o la distribuzione del sistema operativo. La distribuzione del sistema operativo è l'immagine iniziale e ogni livello aggiunto crea una nuova immagine. L'immagine Docker finale ricorda una “cipolla”, con la distribuzione del sistema operativo all'interno e sopra di essa un certo numero di livelli.

Modalità di esecuzione: diretta e shell

Questa istruzione, così come le prossime due che andremo a vedere, hanno due modalità di esecuzione: la modalità shell e la modalità di esecuzione diretta. Nel primo caso, l'istruzione ha la forma mostrata nel Listato 3.6.

Listato 3.6 Sintassi dell'istruzione RUN in modalità shell.

```
<istruzione> <comando> <parametro>
```

Nell'esempio che segue, `RUN` è l'istruzione, `npm` il comando e `install` il parametro.

```
RUN npm install
```

Quando l'istruzione viene eseguita in modalità shell, richiama in background `/bin/sh -c <comando>` e si verifica la normale elaborazione della shell, come se l'istruzione fosse lanciata direttamente dalla riga di comando.

Invece, la forma di esecuzione diretta è simile alla seguente.

Listato 3.7 Sintassi dell'istruzione RUN in modalità di esecuzione diretta.

```
<istruzione> ["eseguibile", "parametro1", "parametro2", ...]
```

Ecco alcuni esempi.

```
RUN ["npm", "install"]
CMD [/bin/echo, "Hello world"]

ENTRYPOINT [/bin/echo, "Hello world"]
```

Quando l'istruzione viene eseguita in questa modalità, richiama direttamente l'eseguibile, e l'elaborazione della shell non avviene. Questa

forma ci permette anche di utilizzare *l'interprete bash* al posto della shell, in questo modo.

```
CMD ["/bin/bash", "Hello world"]
```

Tornando all'istruzione RUN, questa ti consente di installare un'applicazione e i relativi pacchetti richiesti per il corretto funzionamento. Esegue qualsiasi comando sull'immagine corrente e crea un nuovo livello eseguendo il commit dei risultati. Spesso troveremo questo tipo di istruzione utilizzato più volte in un Dockerfile, per la gestione di aggiornamenti e dipendenze.

RUN viene utilizzata nelle due forme descritte in precedenza:

- RUN *comando* (modalità shell);
- RUN ["*eseguibile*", "*parametro1*", "*parametro2*"] (modalità di esecuzione diretta).

Per esempio, in un'immagine con sistema operativo Unix, una pratica comune è quella di verificare se ci sono aggiornamenti delle librerie e installarli; è possibile farlo utilizzando la seguente istruzione.

Listato 3.8 Esempio di aggiornamento dei pacchetti.

```
RUN apt-get update && apt-get install -y
```

Si noti che `apt-get update` e `apt-get install` sono eseguiti in una singola istruzione `RUN`; questo per assicurarsi che verranno installati gli ultimi pacchetti. Se `apt-get install` fosse in un'istruzione `RUN` separata, andrebbe a riutilizzare un livello aggiunto dall'aggiornamento `apt-get`, che potrebbe essere stato creato diverso tempo fa.

Istruzioni CMD e ENTRYPOINT

Riprendendo l'esempio da cui eravamo partiti, abbiamo le ultime tre istruzioni.

```
EXPOSE 8080
ENTRYPOINT echo "Il container si sta avviando..."
CMD [ "node", "server.js" ]
```

Partiamo dalle ultime due: `CMD` ed `ENTRYPOINT` sono due esempi di istruzioni che, così come `RUN`, eseguono dei comandi; in questo caso, `ENTRYPOINT` esegue il comando `echo`, che stamperà sulla console la stringa

specificata come parametro. `CMD` andrà invece a invocare l'eseguibile `node` passando come parametro il file `server.js`. Dov'è la differenza allora, tra questi comandi?

L'istruzione `CMD` consente di impostare un comando predefinito, che verrà eseguito solo quando si esegue il container senza specificare un comando. Se il container Docker viene eseguito con un comando, il comando predefinito verrà ignorato. Se Dockerfile ha più di un'istruzione `CMD`, tutte le istruzioni `CMD` tranne l'ultima vengono ignorate.

`CMD` può avere tre forme:

- `CMD ["eseguibile", "parametro1", "parametro2"]` (modalità di esecuzione diretta, preferito);
- `CMD comando parametro1 parametro2` (modalità shell);
- `CMD ["parametro1", "parametro2"]` (imposta ulteriori parametri predefiniti per `ENTRYPOINT` in formato exec).

Le prime due forme sono state già trattate in precedenza; la terza viene normalmente utilizzata insieme a un'istruzione `ENTRYPOINT` nella forma di esecuzione diretta; imposta i parametri predefiniti che verranno aggiunti dopo i parametri `ENTRYPOINT` se il container viene eseguito senza argomenti della riga di comando.

L'istruzione `ENTRYPOINT` consente di configurare un container che verrà lanciato come eseguibile. Sembra simile a `CMD`, perché consente anche di specificare un comando con parametri; la differenza però è che il comando `ENTRYPOINT` e i suoi parametri non vengono ignorati quando il container Docker viene eseguito con i parametri della riga di comando.

`ENTRYPOINT` viene eseguito nelle due forme seguenti:

- `ENTRYPOINT ["eseguibile", "parametro1>", "parametro2"]` (modalità di esecuzione diretta, preferito);
- `ENTRYPOINT comando parametro1 parametro2` (modulo shell).

È necessario prestare molta attenzione nella scelta della modalità `ENTRYPOINT`, poiché il comportamento tra le due forme differisce in modo significativo: nella prima forma si consente di impostare comandi e parametri e quindi di utilizzare entrambe le forme di `CMD` per impostare parametri aggiuntivi che hanno maggiori probabilità di essere modificati.

Gli argomenti `ENTRYPOINT` vengono sempre utilizzati, mentre quelli `CMD` possono essere sovrascritti dagli argomenti della riga di comando, forniti durante l'esecuzione del container Docker.

Per esempio, se avessimo le seguenti istruzioni, il risultato prodotto sarà "Hello world!". Se invece, durante l'esecuzione dell'immagine, andassimo a utilizzare dei parametri aggiuntivi, la stringa "world!" verrebbe ignorata.

Listato 3.9 Esempio di istruzioni `ENTRYPOINT` e `CMD`.

```
ENTRYPOINT ["/bin/echo", "Hello,"]
CMD ["world!"]
```

In sintesi, utilizzeremo l'istruzione `RUN` per creare l'immagine aggiungendo dei livelli sopra l'immagine iniziale; andremo a utilizzare `ENTRYPOINT` e `CMD` quando creeremo un'immagine Docker eseguibile in cui queste istruzioni debbano sempre essere eseguite. Inoltre, si utilizza `CMD` se è necessario fornire argomenti predefiniti aggiuntivi che potrebbero essere sovrascritti dalla riga di comando durante l'esecuzione del container.

Esempi di Dockerfile

Vediamo ora qualche altro esempio di Dockerfile, per dare un'occhiata ad altre istruzioni che potremmo voler utilizzare: in questo esempio, predisporremo un Dockerfile per l'esecuzione di un server Nginx.

Listato 3.10 Dockerfile per un container con nginx.

```
FROM nginx
COPY content /usr/share/nginx/html
COPY conf /etc/nginx
VOLUME /usr/share/nginx/html

VOLUME /etc/nginx
```

In questo caso, scarichiamo l'ultima immagine disponibile, chiamata `nginx`, che è basata su Ubuntu (vedremo più avanti come verificare questa e altre informazioni).

Se vogliamo modificare direttamente il contenuto o i file di configurazione, possiamo usare un container che abbia accesso ai file; è necessario creare una nuova immagine con i volumi appropriati specificati per l'immagine. Supponendo di voler copiare i file della

cartella `content`, come nell'esempio sopra riportato, specificando anche i volumi per il contenuto e i file di configurazione, utilizziamo il precedente Dockerfile. In questo caso, sarà possibile - come vedremo più avanti - accedere ai file presenti nei volumi messi a disposizione, modificarli e visualizzare in tempo reale il risultato prodotto dalle modifiche apportate.

Nel seguente esempio, andiamo invece a eseguire un'applicazione Python in un'immagine basata su Ubuntu, tramite l'istruzione `FROM`; in questo caso, andremo a impostare come directory principale di lavoro la cartella `/app`, poi a copiare il contenuto della cartella corrente del sistema host nella cartella `/app` del container tramite l'istruzione `ADD` e infine andremo a eseguire lo script `app.py` tramite l'istruzione `CMD`.

Listato 3.11 Dockerfile per un'applicazione Python.

```
FROM python:3
WORKDIR /app
ADD .
CMD [ "python", "./app.py" ]
```

Istruzione ARG

Vediamo un ultimo esempio: creiamo un Dockerfile che esegua un'applicazione Spring Boot per Java.

Listato 3.12 Dockerfile per un'applicazione Spring Boot.

```
FROM openjdk:8-jdk-alpine
ARG JAR_FILE=target//*.jar
COPY ${JAR_FILE} app.jar

ENTRYPOINT ["java","-jar","/app.jar"]
```

Dando per scontato che l'applicazione Java sia pronta e sia stata anche effettuata la build tramite uno dei tanti strumenti messi a disposizione dai vari framework, il risultato sarà un file con estensione `.jar` che permetterà all'applicazione di essere eseguita. Nella prima parte del Dockerfile, scarichiamo l'immagine di OpenJDK, necessaria per l'esecuzione del file Java. Specifichiamo tramite l'istruzione `ARG` la variabile `JAR_FILE`, che viene posta uguale al contenuto di `target//*.jar` e, dopo aver copiato il contenuto del file `*.jar` nel file `app.jar`, eseguiamo l'applicazione tramite l'istruzione `ENTRYPOINT`, che prende l'eseguibile Java e, tramite il parametro `-jar`

(necessario per eseguire codice in questo formato) e il parametro `app.jar`, esegue l'applicazione.

`ARG`, al contrario di `ENV`, è disponibile solo durante la creazione di un'immagine Docker, non dopo che l'immagine è stata creata e che sono stati avviati i container. `ENV` è per quei container che verranno eseguiti in futuro e che avranno bisogno della definizione o modifica di alcune variabili d'ambiente; `ARG` serve invece per creare la tua immagine Docker. `ENV`, come abbiamo visto in qualche esempio in precedenza, ha principalmente lo scopo di fornire valori predefiniti per le future variabili d'ambiente; l'esecuzione di applicazioni a container può infatti accedere alle variabili d'ambiente. I valori `ARG` non sono disponibili dopo la creazione dell'immagine; un container in esecuzione non avrà accesso a un valore variabile `ARG`.

Container

Docker ci aiuta a costruire e distribuire container, all'interno dei quali è possibile “impacchettare” più applicazioni e servizi, come abbiamo visto negli esempi precedenti. I container vengono eseguiti a partire dalle immagini e possono contenere uno o più processi in esecuzione; non dimentichiamo che l'immagine rappresenta uno schema del container e di ciò che deve essere eseguito!

Un container Docker è rappresentato da una serie di file e operazioni che vanno combinate per poter creare un oggetto con il quale dialogare; Docker prende in prestito il concetto del container di trasporto merci: invece di spedire merci, i container Docker spediscono software.

Ogni container contiene un'immagine del software (il suo “carico”) e, come nella metafora, consente di eseguire una serie di operazioni, come la creazione, l'avvio, l'arresto, il riavvio o la distruzione. Come un container da trasporto, Docker non si preoccupa del contenuto quando esegue queste azioni; per esempio, se un container è costituito da un server web, un database o un server di applicazioni, viene eseguito sempre allo stesso modo.

Inoltre, Docker non si preoccupa di dove viene eseguito il nostro container: come un container da trasporto, è intercambiabile e altamente

portatile; per questa ragione (e per molte altre), le modalità d'uso sono infinite.

A questo punto, per poter lavorare con le immagini viste negli esempi precedenti, abbiamo bisogno di vedere il funzionamento di alcuni comandi che è possibile eseguire tramite riga di comando per la costruzione e la gestione dei container.

Build di un'immagine

Il primo comando che andremo a vedere è `build`: viene utilizzato per creare un'immagine da un file Docker, ma il comando deve essere eseguito nella stessa directory del file Docker. Quando viene creata un'immagine, vengono eseguiti i comandi specificati nel Dockerfile; il sistema operativo è installato insieme a tutti i pacchetti richiesti nel container Docker.

Il comando `build` viene eseguito dalla riga di comando utilizzando la sintassi seguente, dove `percorso directory` sarà l'indirizzo della directory in cui vogliamo far lavorare il nostro container; possiamo anche fornire l'indirizzo di un repository Git.

Listato 3.13 Sintassi del comando build.

```
docker build <opzioni> <percorso directory o URL>
```

Se vogliamo includere tutti i file oppure le cartelle che si trovano nella stessa directory del Dockerfile, possiamo costruire l'immagine come nel Listato 3.14, dove il simbolo “.” specifica la cartella corrente.

Listato 3.14 Build della cartella corrente.

```
docker build .
```

Questo comando può essere lanciato solo se ci si trova all'interno della stessa cartella contenente il Dockerfile; è infatti grazie a questo file che Docker può “costruire” il container, a partire dall'immagine.

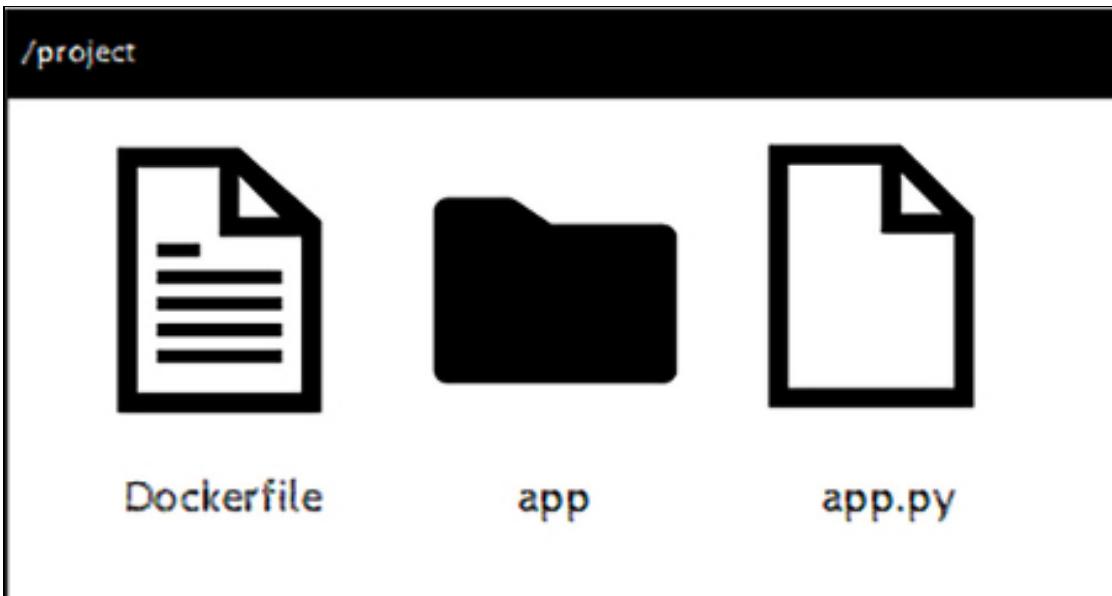


Figura 3.8 Struttura di un progetto Docker.

Un’opzione che ci tornerà molto utile da utilizzare con il comando `build` è `-t`: questo parametro ci permette di specificare un nome da associare alla nuova immagine. Eseguendo il comando seguente, andremo a denominare l’immagine con il nome `my_image`, così da poterla facilmente riconoscere nel caso avessimo più di un’immagine su cui lavorare.

Listato 3.15 Build con tag.

```
docker build -t my_image .
```

È davvero consigliabile associare un nome alle immagini, soprattutto se create a partire da un Dockerfile customizzato; inoltre, utilizzando la notazione con i due punti dopo il nome dell’immagine, potremo anche assegnargli un tag, che altrimenti verrà impostato a `latest` di default.

Riprendendo l’esempio del Dockerfile per costruire un’applicazione in Python, se eseguissimo la build con il seguente comando, l’output sarebbe simile a quello qui riportato.

Listato 3.16 Build immagine Python.

```
root@vbox:/home/sere/Documenti/python# docker build -t python-app .
Sending build context to Docker daemon 3.072kB
Step 1/4 : FROM python:3
3: Pulling from library/python
f15005b0235f: Pull complete
41ebfd3d2fd0: Pull complete
b998346ba308: Pull complete
f01ec562c947: Pull complete
```

```

2447a2c11907: Pull complete
fdd2d569da3e: Pull complete
ac3886b74a9f: Pull complete
3c783a9b35dd: Pull complete
ce16dda809f6: Pull complete
Digest: sha256:e02bd1a92a0dd360a976ec3ce6ebd76f6de18b57b885c0556d5af4035e1767d
Status: Downloaded newer image for python:3
    ---> d47898c6f4b0
Step 2/4 : WORKDIR /app
    ---> Running in fd6e96d79b02
Removing intermediate container fd6e96d79b02
    ---> c94e77fb11d8
Step 3/4 : ADD .
    ---> 68a4dfec68d5
Step 4/4 : CMD ["python", "./app.py"]
    ---> Running in ddd76f14b766
Removing intermediate container ddd76f14b766
    ---> 7b9ea1b57ec5
Successfully built 7b9ea1b57ec5

Successfully tagged python-app:latest

```

Le righe successive al comando che abbiamo lanciato rappresentano i vari passaggi che Docker effettua seguendo le istruzioni presenti nel Dockerfile: nel primo step, scarica l'immagine `python:3`, per poi impostare al secondo step la cartella di lavoro, e così via.

Ora, se volessimo vedere le immagini presenti nel sistema host, dovremo utilizzare il comando `images`: l'output presenta le diverse immagini.

Listato 3.17 Elenco delle immagini di Docker.

```

root@vbox:/home/utente/Documenti/python# docker images
REPOSITORY      TAG      IMAGE ID      CREATED          SIZE
python-app      latest   7b9ea1b57ec5   25 seconds ago  933MB
python          3        d47898c6f4b0     4 days ago       933MB

```

Sotto la colonna `REPOSITORY` troveremo i nomi dei tag assegnati alle diverse immagini; in questo caso vediamo due immagini, una denominata `python-app` e l'altra `python`: per scoprirne la ragione, dobbiamo capire come funziona il file system delle immagini Docker e come sono organizzati i livelli dell'immagine. Come puoi vedere nell'output precedente, ogni immagine docker è composta da diversi passi, dove si crea una relazione gerarchica padre-figlio tra i vari livelli creati. Confrontando il risultato del comando precedente con quello in cui abbiamo eseguito la build, notiamo che l'ID dell'immagine `python-app` corrisponde a quello presente nella penultima riga dell'output della build, e si tratta infatti dello strato più esterno del nostro futuro container; l'ID dell'immagine `python` con tag `3` lo troviamo subito prima dell'inizio del secondo step.

Alla velocità della cache!

Per migliorare la velocità delle build, Docker utilizzerà una cache locale quando pensa di procedere in maniera sicura, perché questo a volte può portare a problemi imprevisti. Nell'output della build di alcuni Dockerfile potremmo notare righe come questa:

```
--> Running in 12345g1f12c8
```

Se invece vedessimo

```
--> Using cache
```

Sappiamo che Docker ha deciso di utilizzare la cache; è possibile disabilitare la cache per una build utilizzando l'argomento `--no-cache` come parametro per il comando `build`.

La cosa importante da tenere a mente è che non stiamo eseguendo un sistema operativo all'interno del container; questo container utilizza un'immagine dei file del sistema operativo *Unix-based* per eseguire i comandi. Quando Unix esegue un comando, inizia a cercare altri file eseguibili o librerie che rispondano al comando richiesto; quando eseguiamo queste cose normalmente, su una macchina, il sistema operativo dice alla macchina di cercare quei file in cartelle specifiche, come per esempio `/usr/lib`; in questo caso, prima vengono caricate le librerie e poi viene eseguito il programma.

Quello che fa un container è dire a un processo di cercare altrove quei file. Negli esempi che abbiamo visto, eseguiamo i diversi comandi nel contesto di una macchina con distribuzione Alpine e alimentiamo l'eseguibile con le librerie Alpine. L'eseguibile richiesto, come per esempio il comando `ls`, viene quindi eseguito come se fosse su Alpine, anche se il sistema operativo host è Ubuntu.

Esecuzione di un container

A questo punto, possiamo eseguire l'immagine come container, tramite il comando `run` e il nome associato all'immagine durante la build.

Listato 3.18 Esecuzione dell'immagine come container.

```
root@vbox:/home/utente/Documenti/python# docker run python-app
```

In questo caso, l'output sarà "Hello world", dal momento che il file Python eseguiva un semplice output della stringa; effettuando però il `run` di altri tipi di immagini, come quella per il server `nginx`, questo comando

potrebbe non restituire apparentemente nulla; per verificare che il container sia in esecuzione, possiamo utilizzare il comando `ps`.

Listato 3.19 Scoprire i container attivi.

```
root@vbox:/home/utente/Documenti/python# docker ps
```

Nel caso del container Python, il risultato sarebbe il seguente.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

Nessuna informazione utile, perché il container utilizzato nell'esempio esegue la sola operazione di output della stringa, e poi termina. Se infatti vedessimo l'elenco completo dei container presenti, l'output sarebbe differente (Figura 3.9).

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
f8a9785fe500	python-app	"python ./app.py"	About a minute ago	Exited (0) About a minute ago		priceless_hellman
b5692149b4d4	python	"python3"	2 minutes ago	Exited (0) 2 minutes ago		festive_mcclintock

Figura 3.9 Output del comando `ps -a`.

Qui possiamo confermare il fatto che il container non è più attivo. Eseguendo lo stesso procedimento di `build` e `run` per l'esempio del paragrafo precedente per la costruzione di un server Nginx, eseguendo il comando `run`, noteremo che non ci viene restituito il comando del terminale, ma rimane come "appeso". Provando ad aprire un'altra scheda del terminale e lanciando il comando `ps`, vedremo (Figura 3.10) che il container è attivo.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
43d26e76e327	nginx	"nginx -g 'daemon off;'"	7 minutes ago	Up 7 minutes	80/tcp	strange_nightingale

Figura 3.10 Output del comando `ps`.

Essendo un application server, potremmo provare ad aprire il browser su `http://localhost:80` e vedere se visualizza la pagina principale; in questo caso, però, ci viene restituito un errore. Questo perché la porta esposta dal server è interna alla rete Docker: questo argomento lo approfondiremo nel prossimo capitolo, ma ci basti sapere che il server, attualmente, è raggiungibile solo all'interno del container, e che per far sì che sia disponibile anche dal sistema host, dobbiamo fornire un parametro supplementare al comando `run`, ossia `-p`, dove possiamo specificare come mappare la porta in uscita dal container Docker verso una porta del sistema host.

Listato 3.20 Run con specifica della porta.

```
root@vbox:/home/utente # docker run -d -p 8080:80 nginx
```

In questo caso, specifichiamo che la porta `80` del container deve essere collegata alla porta `8080` del sistema ospite; il risultato è che tramite browser l'indirizzo `http://localhost:8080` è ora raggiungibile e mostra la pagina d'esempio del server Nginx.

E se la build non andasse a buon fine? Immaginiamo, per esempio, di avere il seguente Dockerfile, dove il nome del pacchetto da installare è scritto male (`redi-server` invece di `redis-server`).

Listato 3.21 Dockerfile errato.

```
FROM ubuntu:18.04
RUN apt-get update && \
    apt-get install -y redi-server && \
    apt-get clean
EXPOSE 6379

CMD ["redis-server", "--protected-mode no"]
```

Eseguendo il comando `build`, verrà visualizzato un errore:

```
root@vbox:/home/sere/Documenti/nginx# docker build -t nginx .
Sending build context to Docker daemon 2.048kB
Step 1/4 : FROM ubuntu:18.04
18.04: Pulling from library/ubuntu
5bed26d33875: Pull complete
f11b29a9c730: Pull complete
930bda195c84: Pull complete
78bf9a5ad49e: Pull complete
Digest: sha256:bec5a2727be7fff3d308193cfde3491f8fba1a2ba392b7546b43a051853a341d
Status: Downloaded newer image for ubuntu:18.04
---> 4e5021d210f6
Step 2/4 : RUN apt-get update && apt-get install -y redi-server && apt-get clean
---> Running in 7e6fba099bae
Get:1 http://archive.ubuntu.com/ubuntu bionic InRelease [242 kB]
Get:2 http://security.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]
...
Get:18 http://archive.ubuntu.com/ubuntu bionic-backports/main amd64 Packages [2496 B]
Fetched 17.7 MB in 11s (1554 kB/s)
Reading package lists...
Reading package lists...
Building dependency tree...
Reading state information...
E: Unable to locate package redi-server

The command '/bin/sh -c apt-get update && apt-get install -y redi-server && apt-get clean' returned a non-zero code: 100
```

Diciamo che a questo punto vogliamo eseguire il debug di questo errore, non essendo chiara la causa; possiamo usare il comando `run` per

creare un container a partire dall'ultimo passaggio riuscito nella build; in questo esempio, dobbiamo far riferimento all'ID immagine `4e5021d210f6`, ossia l'ultima immagine intermedia il cui passaggio è terminato senza errori. Andiamo quindi a eseguire il comando `run` con le opzioni `-t` e `-i`, per poter accedere al terminale del container e verificare se il comando inserito nel Dockerfile è corretto, oppure se ci sono suggerimenti forniti da apt (come avviene spesso nei casi di errata digitazione dei package). Questo ci permette di identificare il problema, uscire dal container e correggere opportunamente il Dockerfile prima di effettuare nuovamente la build.

Convenzioni di denominazione

Avrete forse fatto caso ai nomi che vengono assegnati automaticamente ai container; li possiamo vedere quando eseguiamo il comando `ps`, nell'ultima colonna: `furious_heisenberg, agitated_darwin, romantic_curi...` Da dove nascono i nomi dei container predefiniti di Docker? Sono abbastanza "particolari" da attirare l'attenzione e mettere curiosità. Quando creiamo un nuovo container e non gli assegnamo un nome personalizzato (passando `--name <nome>` tramite la riga di comando), Docker genera un nome automatico. Per tutti i comandi eseguiti in seguito che interagiscono con il container, è possibile utilizzare solo l'ID hash o il suo nome, quindi sì, il nome è abbastanza importante. Il codice per generare questi nomi è disponibile su GitHub ed è scritto in Go, come la maggior parte del codice Docker; nota l'istruzione della funzione principale, che riporta:

```
if name == "boring_wozniak" /* Steve Wozniak is not boring */ {  
    goto begin  
  
}
```

Attraverso questi esempi, ci siamo fatti un'idea del funzionamento dei container: procediamo ora con l'esempio classico che dà inizio a tutti i manuali tecnici del settore: vediamo infatti nel dettaglio il container che viene utilizzato come primo step e approfondiamo anche la conoscenza dei parametri dei comandi appena visti.

“Hello world!”

Fino a poche versioni fa, eseguire Docker su macOS e Windows era una vera seccatura; per fortuna, Docker ha investito in modo significativo nel migliorare l'esperienza dei suoi utenti su questi sistemi operativi, quindi eseguire Docker ora è - quasi - un gioco da ragazzi.

Al termine dell'installazione di Docker, le guide ufficiali consigliano di sottoporre a test l'installazione di Docker eseguendo il comando `docker info`, per vedere se l'output riportato è proprio il dettaglio dell'installazione Docker, oppure di eseguire quanto segue.

Listato 3.22 Hello world.

```
1.      root@vbox:/home/utente/Documenti/hello-world# docker run hello-world
2.      Unable to find image 'hello-world:latest' locally
3.          latest: Pulling from library/hello-world
4.  1b930d010525: Pull complete
5.          Digest:
sha256:f9dfddf63636d84ef479d645ab5885156ae030f611a56f3a7ac7f2fdd86d7e4e
6.  Status: Downloaded newer image for hello-world:latest
7.
8. Hello from Docker!
9.      This message shows that your installation appears to be working
correctly.
10.
11.     To generate this message, Docker took the following steps:
12.         1. The Docker client contacted the Docker daemon.
13.         2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
14.             (amd64)
15.         3. The Docker daemon created a new container from that image which runs
the
16.             executable that produces the output you are currently reading.
17.         4. The Docker daemon streamed that output to the Docker client, which
sent it
18.             to your terminal.
19.
20.     To try something more ambitious, you can run an Ubuntu container with:
21. $ docker run -it ubuntu bash
22.
23.     Share images, automate workflows, and more with a free Docker ID:
24.     https://hub.docker.com/
25.
26.     For more examples and ideas, visit:
27.     https://docs.docker.com/get-started/
```

Il comando pull

Analizziamo nel dettaglio il Listato 3.22: nelle righe 1-6 ci viene restituito un messaggio che spiega come Docker non sia riuscito a trovare un'immagine chiamata `hello-world` già scaricata localmente; questo perché, se non partiamo da un Dockerfile da noi definito, dovremmo cercare l'immagine da utilizzare tramite uno dei registri a disposizione, come Docker Hub. A destra della Figura 3.11 notiamo infatti che ci viene suggerito di copiare e incollare nel terminale il comando `pull`, per estrarre l'immagine in locale; questo comporterebbe il download dell'immagine, che a quel punto sarà disponibile per essere eseguita (usiamo il comando

`docker images` per verificare se c'è un'immagine `hello-world` nell'elenco di quelle disponibili).

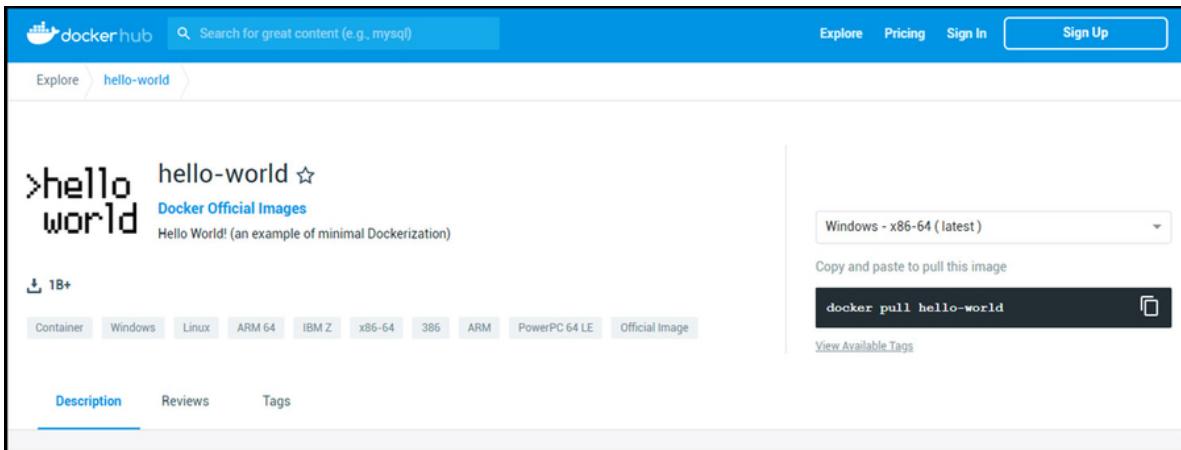


Figura 3.11 Il registro di hello-world su Docker Hub.

Il risultato del comando `pull`, ammettendo di non aver scaricato l'immagine, è riportato nel Listato 3.23.

Listato 3.23 Il comando `pull`.

```
root@vbox:/home/utente/Documenti# docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:f9dfddf63636d84ef479d645ab5885156ae030f611a56f3a7ac7f2fdd86d7e4e
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest
```

NOTA

Il comando `run` crea prima un livello di container scrivibile sull'immagine specificata, e poi lo avvia utilizzando il comando specificato. Se l'immagine che stiamo tentando di eseguire non viene ancora scaricata, verrà automaticamente recuperata con `pull`. La maggior parte delle immagini verrà creata sopra un'immagine di base dal registro del Docker Hub, il quale contiene molte immagini predefinite, che possiamo estrarre e provare senza doverne definire e configurare di customizzate.

Per scaricare una particolare immagine o un insieme di immagini (per esempio un repository), utilizziamo il comando `pull`; in questo caso, Docker scaricherà o aggiornerà un'immagine.

Tornando all'esempio `hello-world`, nella riga 8 vediamo il messaggio `Hello from Docker!`; questo è il messaggio restituito dal container, così come le righe che seguono. Qui vengono specificate, in maniera molto sintetica, le linee guida che dovrebbe seguire un utente alle prese con

Docker per la prima volta; nelle righe 20-21 ci viene data un’ulteriore indicazione: per provare qualcosa di più “ambizioso”, basta provare a eseguire il container indicato. Di seguito troviamo l’output di esecuzione del comando riportato.

Listato 3.24 Pull dell’immagine ubuntu.

```
root@vbox:/home/utente/Documenti# docker run -it ubuntu bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
5bed26d33875: Pull complete
f11b29a9c730: Pull complete
930bda195c84: Pull complete
78bf9a5ad49e: Pull complete
Digest: sha256:bec5a2727be7fff3d308193cfde3491f8fba1a2ba392b7546b43a051853a341d
Status: Downloaded newer image for ubuntu:latest

root@bdce2b32c4a0:/#
```

In questo caso, notiamo che se l’utente iniziale era `vbox`, ora abbiamo un utente definito come `bdce2b32c4a0`; questo perché il parametro `-it` indica a Docker di allocare uno pseudo-terminal collegato all’input del container; creando una shell bash interattiva nello stesso. Questo ci permette, in altre parole, di entrare all’interno del container ed eseguire dei comandi *bash*.

```
root@bdce2b32c4a0:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@bdce2b32c4a0:/# █
```

Figura 3.12 Il terminale del container.

Questa opzione ci apre la possibilità di avere a disposizione la riga di comando all’interno del container; il comando `/bin/bash` fornisce una shell *bash* con cui comunicare.

Avviare e arrestare i container

Quando si esce dalla shell, il container si arresterà e i container funzioneranno solo finché il loro processo principale è attivo; questo significa che qualsiasi cambiamento apportato all’interno del container stesso, una volta riavviato il container tramite comando `run`, verrà “perso” all’arresto del container. Per verificare ciò, proviamo a creare un file

all'interno della cartella `/home`, digitiamo `exit` per uscire dalla riga di comando del container.

Listato 3.25 Creazione di un file all'interno del container.

```
root@bdce2b32c4a0:/# cd home/
root@bdce2b32c4a0:/home# touch miofile.txt
root@bdce2b32c4a0:/home# ll
total 8
drwxr-xr-x 1 root root 4096 Apr  6 08:57 .
drwxr-xr-x 1 root root 4096 Apr  6 08:49 ../
-rw-r--r-- 1 root root     0 Apr  6 08:57 myfile.txt
root@bdce2b32c4a0:/home# exit

root@vbox:/home/utente#
```

Il container viene arrestato; dal momento che quella che abbiamo eseguito è un'istanza del nostro container, per poter recuperare quell'istanza e non perdere le modifiche fatte, nel nostro caso, la creazione del file, possiamo eseguire il comando `start` insieme al comando `attach`.

Listato 3.26 Comandi start e attach.

```
docker start `docker ps -q -l` && docker attach `docker ps -q -l`
```

Qui il parametro `docker ps -q -l` viene sostituito dall'ID dell'ultimo container Docker creato. Inoltre, possiamo usare opzioni come `-it` per avviare il container in modalità interattiva, e vedremo che il file è sempre lì.

Utilizziamo il comando `start` per avviare uno o più container arrestati, e usiamo il comando `attach` per collegare l'output del terminale a un container in esecuzione utilizzando l'ID o il nome del container. Ciò consente di visualizzare l'output in corso o di controllarlo in modo interattivo, come se i comandi fossero in esecuzione direttamente nel terminale.

Con un normale container Docker, i dati verrebbero “persi” quando il container viene arrestato, e questo proprio perché il container rappresenta un'istanza della nostra immagine. Questo ovviamente non è l'ideale per molte applicazioni, ma si tratta di una funzionalità e non di un bug. La completa distruzione di tutti i dati nel container garantisce che in qualsiasi momento sia possibile distruggere e ricreare il container nel suo stato iniziale; questo ci dà una perfetta coerenza tra le varie esecuzioni dell'applicazione.

Per le applicazioni che devono conservare i dati, il modo più comune è utilizzare un volume Docker per montare un'altra directory nel proprio container; questa directory verrebbe riportata sull'host e sarebbe fruibile e permanente, permettendo così al container di leggere o scrivere sul volume direttamente come qualsiasi altro file system.

Rimozione di container e immagini

Un'altra opzione molto comune quando viene avviato un container è utilizzare il parametro `-rm`; questo ci permette di rimuovere automaticamente il container all'uscita dello stesso. Docker, infatti, non rimuove gli oggetti inutilizzati, come container, immagini o volumi, a meno che non gli venga esplicitamente indicato di farlo. Lavorando con Docker, è facile accumulare un gran numero di oggetti inutilizzati, che consumano spazio su disco in modo significativo; esistono in realtà diversi comandi per la gestione dei container e delle immagini, come il comando `system prune`: questo comando rimuove tutti i container arrestati, tutte le immagini sospese e tutte le reti non utilizzate. Per rimuovere anche tutti i volumi non utilizzati, basterà utilizzare il parametro `--volumes`.

Listato 3.27 Il comando prune.

```
root@vbox:/home/utente# docker system prune --volumes
WARNING! This will remove:
 - all stopped containers
 - all networks not used by at least one container
 - all volumes not used by at least one container
 - all dangling images
 - all dangling build cache

Are you sure you want to continue? [y/N]
```

Allo stesso modo, possiamo rimuovere anche solo una determinata classe di oggetti, come tutti i container o tutte le immagini, usando come parametro `prune`.

Listato 3.28 Il comando container con l'opzione prune.

```
root@vbox:~# docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
d67d1aece023c82ce7ae0faab5e136f5d5104007d808b5793a370a91beba692c
137efebbb2c3f5700df120fc6ed24142aadcc65a81bf3986a6f24c136060ad0c2f
bdce2b32c4a0a2e2f944eed2e292ec2409aefef6759fba65196e496896cc398f

fd44de9ba200ad2586d9b733df90a8fcfc1c26a1cdf07b5f8dc6d8e35d9b1694
```

*****ebook converter DEMO

Watermarks*****

Abbiamo detto che i container non vengono rimossi automaticamente una volta arrestati; per rimuovere una o più immagini Docker, utilizziamo il comando `docker container rm` seguito dall'ID dei container da rimuovere; è possibile ottenere un elenco di tutti i container presenti (attivi e non) passando al comando `docker container ls` il parametro `-a` (Figura 3.13).

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
d67d1aece023	ubuntu	"bash"	3 hours ago	Exited (0) 19 minutes ago
137efeb2c3f	hello-world	"/hello"	3 hours ago	Exited (0) 3 hours ago
bdce2b32c4a0	ubuntu	"bash"	4 hours ago	Exited (127) 3 hours ago
fd44de9ba200	hello-world	"/hello"	4 hours ago	Exited (0) 4 hours ago

Figura 3.13 L'elenco dei container.

Una volta che conosciamo l'ID dei container da eliminare, basterà usarli come parametri del comando `rm`; per esempio, ecco come rimuovere i primi due container elencati nel Listato 3.28.

```
root@vbox:/home/utente# docker rm d67d1aece023 137efeb2c3f
```

Attenzione, però: i container possono essere rimossi solo se arrestati; potremmo quindi incorrere in un errore come questo:

```
Error response from daemon: You cannot remove a running container ...
```

che ci impedirebbe di procedere, fintantoché non arrestiamo tutti i container. Per arrestare un container, è sufficiente utilizzare il comando `docker stop`.

Listato 3.29 Il comando `stop`.

```
root@vbox:/home/utente# docker stop d67d1aece023 137efeb2c3f
```

Analizzare un container

Finora abbiamo visto come eseguire, scaricare, arrestare e riavviare dei container. Siamo quindi pronti per approfondire il “dietro le quinte” di un container in esecuzione (o non), avendo molti strumenti a disposizione che ci permettano di osservare più da vicino che cosa avviene.

Questo paragrafo spiega come utilizzare il comando `docker inspect` per controllare un’immagine prima di eseguirla, per vedere come è configurata. Lo stesso comando di ispezione può essere utilizzato sui container per visualizzare le informazioni di *runtime* (come l’ID del

processo del container, le interfacce di rete e i volumi montati) insieme ad altre informazioni.

Dopo che le immagini e i container sono stati ispezionati, è possibile utilizzare altri comandi per esaminarli ulteriormente. Esistono ulteriori opzioni al comando `docker` da utilizzare, come per esempio quelle che possono essere utilizzate direttamente con il processo di un dato container, oppure per eseguire un nuovo processo, visualizzare i file di registro, copiare dei file ed elencare i processi all'interno di un container, ma procediamo per gradi.

Ogni immagine contiene informazioni che includono impostazioni di configurazione predefinite, informazioni su chi l'ha creata e dettagli su quando e come è stata creata. Dopo aver eseguito un container, vengono incluse informazioni aggiuntive, come le impostazioni di rete, se il container è ancora in esecuzione e informazioni sui volumi montati.

Eseguendo il comando `docker inspect` con un'immagine o un container, è possibile visualizzare i dati di basso livello a esso associati. Questo può aiutarci a utilizzare o eseguire il debug di problemi con l'immagine o il container. Per esempio, vedere l'indirizzo IP di un container può darci informazioni su come un'applicazione client può raggiungere il servizio in esecuzione.

Per visualizzare l'output in modo che sia facile da leggere, `docker inspect` visualizza informazioni in formato JSON; ciò semplifica la visualizzazione di ogni singola chiave e coppia di valori e come si adatta alla struttura delle informazioni associate all'immagine o al container.

Per vedere come funziona questo comando, i seguenti esempi illustrano come esaminare i dati di basso livello associati a un'immagine Ubuntu che esegue un semplice server web usando il comando `python`.

Il comando di seguito riportato controlla l'ultimo container Ubuntu.

Listato 3.30 Il comando `docker inspect`.

```
root@vbox:/ docker inspect ubuntu:latest | less
```

Se l'immagine non è già stata estratta nel sistema, questa azione estrae l'immagine e la controlla. Si noti che ogni volta che si estrae un'immagine, l'output del comando e l'immagine stessa potrebbero cambiare. Ecco l'output del comando `docker inspect` sull'immagine Ubuntu:

```
[{"Architecture": "amd64",  
 "Author": "",  
 "Comment": ""},
```

```

    "Config": {
      "AttachStderr": false,
      "AttachStdin": false,
      "AttachStdout": false,
      "Cmd": [
        "/bin/bash"
      ],
      ...
      "Env": [
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
      ],
      ...
      "NetworkDisabled": false,
      ...
      "Created": "2015-03-20T06:16:47.003636554Z",
      "DockerVersion": "1.4.1",
      "Id": "d0955f21bf24f5bffffd32d2d0bb669d0564701c271bc3dfc64cf5adfdec2d07",
      "Os": "linux",
      ...
    }
  ]
}

```

Dall'output, è possibile notare che l'architettura del container è `amd64` (compatibile con PC a 64 bit). Le righe `Comment` e `Author` non sono state definite al momento della creazione di questa immagine; la sezione `Config` configura l'ambiente in cui viene eseguito il container.

Come abbiamo già detto, per impostazione predefinita viene eseguito il comando `/bin/bash`, se in fase di esecuzione non viene identificato nessun altro comando. Poiché il collegamento di input, output ed errori standard (rispettivamente, `AttachStdin`, `AttachStdout` e `AttachStderr`) sono impostati su `false`, è necessario specificare le opzioni sulla riga di comando quando si esegue il comando `bash` predefinito di questo container. Nella sezione `Env` è impostata solo la variabile `PATH` per definire quali directory vengono ricercate per i comandi eseguiti. `NetworkDisabled` impostato su `false` indica che l'interfaccia di rete deve essere avviata all'interno del container.

L'ultima sezione mostra le informazioni di base su come è stato creato il container: è possibile vedere la data in cui è stata creata l'immagine del container, la versione di Docker utilizzata per crearla (1.4.1), la forma estesa dell'ID del container e il sistema operativo utilizzato (Linux).

Mentre le immagini di base devono essere generiche, le immagini create per eseguire applicazioni specifiche tendono a includere impostazioni di configurazione di livello più basso. Un esempio di un'immagine che include una maggiore personalizzazione è l'immagine `wordpress`, disponibile sul Docker Hub. Ecco l'output del comando `docker inspect` su questa immagine:

*****ebook converter DEMO

Watermarks*****

```
[{
  "Config": {
    ...
    "Cmd": [
      "apache2-foreground"
    ],
    ...
    "Entrypoint": [
      "/entrypoint.sh"
    ],
    "Env": [
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
      "PHP_INI_DIR=/usr/local/etc/php",
      "PHP_EXTRA_BUILD_DEPS=apache2-dev",
      "PHP_EXTRA_CONFIGURE_ARGS=--with-apxs2",
      "GPG_KEYS=6E4F6AB321FDC07F2C332E3AC2BF0BC433CFC8B3
      0BD78B5F97500D450838F95DFE857D9A90D90EC1",
      "PHP_VERSION=5.6.8",
      "WORDPRESS_VERSION=4.2.1",
      "WORDPRESS_UPSTREAM_VERSION=4.2.1",
      "WORDPRESS_SHA1=c93a39be9911591b19a94743014be3585df0512f"
    ],
    "ExposedPorts": {
      "80/tcp": {}
    },
    ...
    "Volumes": {
      "/var/www/html": {}
    },
    "WorkingDir": "/var/www/html"
  },
  ...
}
```

Il comando rivela che nell’immagine `wordpress` sono state aggiunte più informazioni specifiche del container rispetto a quella vista in precedenza: nella sezione `Config`, lo script `entrypoint.sh` nella directory principale del container è impostato per essere eseguito all’avvio del container. Lo script `apache2-foreground` è definito come un’opzione che deve essere passata allo script `entrypoint.sh` quando questo viene eseguito.

Viene anche definito un set di variabili d’ambiente: queste variabili impostano aspetti come la directory nel percorso della shell, la directory che contiene script PHP `ini` e informazioni sulla versione utilizzata. La variabile `ExposedPorts` espone la porta TCP 80 (protocollo http standard) dal container all’host. La variabile `volumes` dice a Docker di montare `/var/www/html` dall’host all’interno del container; quindi imposta `WorkingDir` su quella directory, da utilizzare come directory di lavoro per il container.

Dopo aver eseguito un container, possiamo eseguire una query su quel container per trovare gran parte delle informazioni impostate durante l’interrogazione dell’immagine originale. Oltre alle informazioni sull’immagine, è possibile vedere anche molti dati che sono stati

impostati dal comando `docker run`, se l’immagine è in esecuzione, e quindi aggiunti alle impostazioni di configurazione sul container in fase di costruzione.

In questo caso, il comando `docker inspect` può aiutarci a risolvere i problemi con un container in esecuzione o semplicemente a capire come sta funzionando. Possiamo eseguire questo comando su qualsiasi container, purché non sia stato eliminato; in altre parole, su qualsiasi container che è possibile vedere con i comandi `docker ps` o `docker ps -a`.

Riprendendo l’esempio dell’immagine Ubuntu che contiene un server web `nginx` ed eseguendo il comando `docker inspect` mentre è in esecuzione, un esempio di output potrebbe essere il seguente.

```
"ContainerConfig": {
    "Hostname": "07626abd8663",
    "Domainname": "",
    "User": "",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
    "ExposedPorts": {
        "80/tcp": {}
    },
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": [
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
        "NGINX_VERSION=1.17.10",
        "NJS_VERSION=0.3.9",
        "PKG_RELEASE=1~buster"
    ],
    "Cmd": [
        "/bin/sh",
        "-c",
        "#(nop) ",
        "CMD [\"nginx\" \"-g\" \"daemon off;\"]"
    ],
    "Labels": {
        "maintainer": "NGINX Docker Maintainers <docker-maint@nginx.com>"
    }
}

...
```

Questo output ci mostra qualche informazione in più: intanto, nella sezione `Labels`, possiamo vedere tutti i metadati che vengono normalmente impostati all’interno del Dockerfile tramite l’istruzione `LABEL` (anche se abbiamo visto che non è l’unico modo) e in questo caso riguardano le informazioni di contatto per chi gestisce l’immagine; nella sezione `Cmd` vediamo il comando che viene eseguito per far partire il container, ossia `nginx -g`. Anche qui la porta esposta per raggiungere il server è la 80 e le

variabili d'ambiente riguardano le configurazione di Nginx; se avessimo eseguito un container con un'immagine Python e avessimo immesso il comando `docker inspect`, vedremmo che la sezione `Cmd` corrisponderebbe a quanto specificato nel Dockerfile nell'istruzione omologa, così come vedremmo la cartella di lavoro impostata come `/app`:

```
...
"Cmd": [
    "python",
    "./test.py"
],
"Image":
"sha256:f3b5d82654205f2037edbffa17c177b09f3675630ccaca621d45b22c2db5597e",
"Volumes": null,
"WorkingDir": "/app",
...
```

Le origini di Docker

Le immagini vengono create costruendo un'immagine di base; ogni volta che viene eseguito un nuovo comando sull'immagine, viene creato un nuovo livello. Se questi livelli vengono salvati con l'immagine, è possibile visualizzare tali informazioni in un secondo momento utilizzando il comando `docker`.

Ci sono buoni motivi per salvare un'immagine in livelli invece di comprimere quei livelli in una singola immagine: se, per esempio, si stanno eseguendo più container sullo stesso computer, se hanno tutti la stessa immagine di base (identificata da un ID), è possibile risparmiare spazio di archiviazione. Sulla parte superiore dell'immagine di base, è possibile aggiungere un determinato set di librerie di sviluppo e quindi sarebbe possibile aggiungere un set standard di servizi.

Per visualizzare la cronologia di un'immagine, esiste il comando `docker history`, da utilizzare su una data immagine. Riprendendo l'immagine base di Ubuntu, vediamo il comando in esecuzione.

Listato 3.31 Il comando docker history.

```
root@vbox:/# docker history ubuntu
IMAGE           CREATED          CREATED          SIZE
BY              13 days ago      /bin/sh -c
                0B
8a5c7be2e9c6   13 days ago      /bin/sh -c #(nop) WORKDIR
pwd
4d2144c492c0   13 days ago      /bin/sh -c 0B
/app
fbdc545395b4   13 days ago      /bin/sh -c
pwd            0B
```

*****ebook converter DEMO

Watermarks*****

```

f4dce8394a0c    13 days ago      /bin/sh -c #(nop) WORKDIR
/a/b/c          0B
b2e5522dalfc    13 days ago      /bin/sh -c #(nop) WORKDIR
/a/b            0B
98627be8993c    13 days ago      /bin/sh -c #(nop) WORKDIR
/a              0B
4e5021d210f6    4 weeks ago       /bin/sh -c #(nop) CMD
["/bin/bash"]
<missing>        4 weeks ago       /bin/sh -c mkdir -p /run/systemd && echo
'do...           7B
<missing>        4 weeks ago       /bin/sh -c set -xe && echo '#!/bin/sh' >
/...             745B
<missing>        4 weeks ago       /bin/sh -c [ -z "$(apt-get indextargets)"
]              987kB
<missing>        4 weeks ago       /bin/sh -c #(nop) ADD
file:594fa35cf803361e6...   63.2MB

```

Da qui possiamo vedere che l’immagine risale a quattro settimane fa, anche se non si tratta di quella originale; possiamo anche vedere che alla creazione dell’immagine sono stati eseguiti diversi comandi tramite la shell. Qualsiasi container che in futuro utilizzi uno qualsiasi dei livelli dell’immagine elencati per nome nella colonna `IMAGE`, non dovrà estrarre quel particolare livello, ma potrà semplicemente usare quello già presente nel sistema.

Gestione dei processi

Il container è attivo e funzionante e abbiamo ispezionato a fondo le sue impostazioni di basso livello. Utilizzando il comando `docker` con alcuni parametri è possibile ottenere ulteriori informazioni sui container in esecuzione; un container in genere esegue un processo. Tuttavia, usando `docker exec` è possibile aprire altri processi all’interno di quel container, in particolare è possibile aprire una shell per guardare che cosa sta succedendo al suo interno. Con il comando `docker top` è possibile visualizzare tutti i processi in esecuzione in un container; nel Listato 3.32 vediamo un esempio (il tuo output sarà sicuramente differente).

Listato 3.32 Il comando `docker top`.

```

root@vbox:/# docker top unruffled_williamson
UID      PID  PPID  C   STIME   TTY    TIME     CMD
Root     2294  2279  0  22:13    ?  00:00:00  nginx: master process nginx
-g daemon off;

systemd+  2360  2294  0  22:13    ?  00:00:00  nginx: worker process

```

Dall’output è possibile vedere che all’interno del container sono in esecuzione due processi; il primo riguarda il comando `nginx -g` che ha

avviato il server web e il secondo è il processo di script per l'avvio dello stesso.

Oltre a vedere i processi in esecuzione, è anche possibile avviare un nuovo processo per interagire con un container; possiamo eseguire qualsiasi comando desiderato all'interno del container, tramite il comando `docker exec`.

Per esempio, supponiamo di voler eseguire il comando `ls` su uno dei container a disposizione. La prima cosa da fare è identificare il nome del container o il suo ID. Ora, per eseguire il comando `ls` su questo container, è sufficiente aggiungere il comando `ls` all'ID del container. L'uso più popolare del comando `docker exec` è infatti proprio quello di avviare un terminale bash all'interno di un container.

Per avviare una shell bash in un container Docker, basta eseguire il comando con l'opzione `-it` e specificare l'ID e il percorso della shell bash.

Listato 3.33 Il comando docker exec.

```
root@vbox:/# docker exec -it <container> /bin/bash
```

Quando si esegue questo comando, si avrà un terminale bash interattivo in cui è possibile eseguire tutti i comandi desiderati.

Avendo familiarità con i sistemi operativi Linux, probabilmente avrai sentito parlare del concetto di *descrittori di file*. Ogni volta che si esegue un comando, si creano tre descrittori di file, già citati in precedenza.

- `STDIN`: chiamato anche “standard di input”, viene utilizzato per digitare e inviare i comandi (per esempio una tastiera, un terminale e così via).
- `STDOUT`: chiamato “standard di output”, rappresenta il luogo in cui verranno scritti gli output del processo (il terminale stesso, un file, un database e così via).
- `STDERR`: chiamato “standard di errore”, è correlato allo standard output e viene utilizzato per visualizzare gli errori.

Quando si esegue `docker exec` con l'opzione `-it`, si associa lo standard di input dell'host a quello del processo in esecuzione nel container.

Che cosa abbiamo imparato

- Qual è la differenza tra una *virtual machine* e Docker.
- Come funziona l'architettura Docker e come vengono costruite le immagini.
- Come scrivere un Dockerfile, definendo le istruzioni che compongono l'immagine.
- Come funzionano le istruzioni `FROM`, `ENTRYPOINT`, `CMD`, `RUN`, `ARG`, `VOLUME`, `LABEL`, `WORKDIR`, `CP`, `ADD`, `ARG` ed `ENV`.
- Qual è la differenza tra la *modalità di esecuzione* e la *modalità shell*.
- Cos'è un container, come costruirlo e come avviarlo, arrestarlo o rimuoverlo.
- Come rimuovere e pulire immagini e volumi.
- Come scaricare, tramite Docker Hub, un'immagine tramite il comando `docker pull`.
- Utilizzando il comando `docker inspect` sulle immagini di base, è possibile visualizzare le istruzioni principali su come deve essere eseguita l'immagine.
- L'ispezione delle immagini permette di rivelare dettagli specifici su quel container, come le variabili d'ambiente, i punti di ingresso, i volumi montati e le directory di lavoro, così come le attività in esecuzione, l'ID del processo in esecuzione, le interfacce di rete, i volumi montati e altre informazioni.
- Come controllare la storia di un container, tramite il comando `docker history`.
- Come si controllano e gestiscono i processi all'interno di un container, tramite il comando `docker top`.
- Il comando `docker exec` consente di eseguire un nuovo comando all'interno di un container in esecuzione.

Gestire i dati

Per impostazione predefinita, tutti i file creati all'interno di un container sono archiviati su un livello scrivibile del container. Ciò significa che:

- *i dati non persistono* quando quel container non esiste più e può essere difficile estrarre dati dal container, se sono utilizzati da un altro processo;
- *il livello scrivibile del container è strettamente accoppiato al computer host* su cui è in esecuzione; non è possibile spostare facilmente i dati altrove;
- la scrittura nel livello scrivibile di un *container* richiede un driver di archiviazione per gestire il file system; il driver di archiviazione fornisce un file system `union`, che usa il kernel Linux; questa ulteriore astrazione riduce le prestazioni rispetto all'utilizzo di volumi di dati, che scrivono direttamente nel file system *host*.

Per questa ragione, Docker offre due opzioni che permettono ai container di archiviare i file nel sistema ospite, in modo che i file siano persistenti anche dopo l'arresto del container: *volumi* e *bind mount*. Indipendentemente dal tipo di mount che sceglieremo di utilizzare, i dati sembreranno uguali all'interno del container; il file system viene infatti esposto come una directory o un singolo file nel file system del container.

La differenza tra i due è nelle funzionalità: i volumi sono creati e gestiti da Docker; è possibile creare un volume in modo esplicito, utilizzando il comando di creazione del volume; oppure Docker può creare il volume durante la creazione di un container o di un servizio; quando si crea un volume, questo viene archiviato in una directory

sull'host Docker. Quando si monta il volume in un container, la directory associata è quella che viene montata nel container. Il funzionamento, come vedremo, è molto simile al modo in cui funzionano i *bind mount*, tranne per il fatto che i volumi sono gestiti da Docker e isolati dalle funzionalità principali del computer host.

Un determinato volume può essere montato contemporaneamente su più container. Quando nessun container in esecuzione utilizza un volume, il volume è ancora disponibile e non viene rimosso automaticamente.

Quando si monta un volume, può essere definito tramite un nome a scelta oppure lasciato come anonimo. A quest'ultimo tipo di volume non viene assegnato un nome esplicito quando viene montato per la prima volta in un container, quindi Docker gli assegna un nome casuale che è garantito essere univoco all'interno dell'host. I volumi supportano anche l'uso di driver di volume, che consentono, tra l'altro, di archiviare i dati su host remoti o su cloud.

I *bind mount* sono disponibili sin dai primi tempi di Docker; hanno però funzionalità limitate rispetto ai volumi. Quando si utilizza un bind mount, un file o una directory sul computer host viene montato in un container. Il file o la directory sono indicati dal suo percorso completo sul computer host e non è necessario che il file o la directory esistano già sul sistema ospite. I bind mount sono molto performanti, ma si basano sul file system della macchina host, con una struttura di directory specifica per quel sistema.

Come vedremo a breve, nei casi d'uso più frequenti di utilizzo di Docker, è consigliabile impiegare i volumi: ci sono diverse ragioni che portano uno sviluppatore a scegliere questo tipo di persistenza, come le seguenti.

- *La condivisione dei dati tra più container in esecuzione.* Se non viene creato esplicitamente, un volume viene creato la prima volta che viene montato in un container. Quando quel container si arresta o viene rimosso, il volume continua a esistere. Più container, come già accennato, possono montare contemporaneamente lo stesso volume, sia in lettura sia in scrittura o anche in sola lettura. I volumi vengono rimossi solo esplicitamente.
- *Quando il sistema host non ha la garanzia di avere una determinata directory o struttura di file a disposizione;* i volumi aiutano a

disaccoppiare la configurazione dell'host dall'esecuzione del container.

- *Quando si desidera archiviare i dati del proprio container su un host remoto o un provider cloud*, caso sempre più frequente, come vedremo verso la fine del volume, anziché in locale.
- In ultimo, ma non meno importante, *quando è necessario eseguire il backup, il ripristino o la migrazione dei dati da un host Docker a un altro*, i volumi sono la scelta migliore. È infatti possibile interrompere i container e, utilizzando il volume, eseguire il backup della directory senza alcun problema di gestione.

Fatte queste premesse, vediamo come creare e lavorare con i volumi, e come gestire i dati al loro interno.

Gestione dei volumi

Fino a questo punto, i nostri container sono stati solo oggetti monouso che possiamo eseguire e poi distruggere. Abbiamo avuto modo di vedere e analizzare ciò che è all'interno del container fintanto che questo non viene arrestato o eliminato (sebbene, tecnicamente, sia possibile leggervi e copiarvi dati). La maggior parte delle cose che vorremmo fare, specialmente come sviluppatori, ci imporranno di trasferire dei dati all'interno dei container o di poter accedere ai dati all'interno di un container; potremmo arrivare al punto di essere in grado di condividere dati tra container.

Fortunatamente, Docker ha il concetto di volume, che è una sezione dell'immagine che contiene dati che dovrebbero essere montati nell'immagine, in qualche modo. È molto simile a quando si collega una chiavetta USB al PC: il sistema operativo sa che ci sono punti di ingresso (usando macOS, è possibile trovarli nel punto `/volumi`, o come lettere di unità inutilizzate utilizzando Windows) in cui “collegheremo” o memorizzeremo i dati al di fuori della memoria del sistema con cui stiamo lavorando. Possiamo dire a Docker che un volume è solo una directory all'interno dell'immagine dove memorizzare le informazioni o che i dati del volume provengono da una fonte esterna, che può essere un'altra immagine o un computer host. Entrambi hanno vantaggi e

svantaggi, ma ne discuteremo più avanti. Per ora, iniziamo a lavorare con i dati.

Per tutti i comandi che abbiano a che fare con i volumi, utilizzeremo il parametro `-v` nei vari comandi Docker. È anche possibile specificare più opzioni `-v`, per montare o creare più volumi di dati; alcune applicazioni necessiteranno di uno, due o più volumi e potrebbe essere necessario usare sia volumi montati direttamente nel sistema ospitante, sia volumi di dati normali.

Esistono diversi modi per inizializzare i volumi, con alcune sottili differenze che è importante conoscere e che analizzeremo a breve. Il modo più diretto è dichiarare un volume in fase di esecuzione con l'opzione `-v`.

Listato 4.1 Inizializzare un volume.

```
root@vbox:/home/utente# docker run -it --name vol-test -h CONTAINER -v /data
debian /bin/bash
```

Ciò renderà la directory `/data` all'interno del container disponibile anche al di fuori del file system Union e direttamente accessibile sull'host. Tutti i file che l'immagine conteneva nella directory `/data` verranno copiati nel volume. Possiamo scoprire dove risiede il volume sul sistema ospitante usando il comando `docker inspect`.

Listato 4.2 Ispezione del volume.

```
root@vbox:/home/utente# docker inspect -f "{{json .Mounts}}" vol-test | jq .
```

Il risultato dovrebbe essere il seguente output:

```
docker inspect -f "{{json .Mounts}}" vol-test | jq .
[
  {
    "Name": "9c8e3e7d6a4...",
    "Source": "/var/lib/docker/volumes/9c8e3e7d6a4.../_data",
    "Destination": "/data",
    "Driver": "local",
    "Mode": "",
    "RW": true,
    "Propagation": ""
  }
]
```

In questo caso, essendo il sistema ospite Unix-based, vediamo che la cartella in cui viene montato il volume è sotto `/var/lib/docker/volumes`; se andassimo a creare un file al suo interno e poi tornassimo al terminale ed eseguissimo il comando per elencare i file presenti nella cartella `/data`, vedremmo che il file è presente. Le modifiche si riflettono

immediatamente, poiché la directory sul container è semplicemente montata in una directory sull'host.

Lo stesso effetto si può ottenere usando un'istruzione `VOLUME` in un Dockerfile, come abbiamo accennato in precedenza: avendo un file a disposizione come il seguente.

Listato 4.3 Uso di VOLUME in un Dockerfile.

```
FROM debian:wheezy
VOLUME /data
```

Il volume verrà creato esattamente come prima, e sarà possibile accedere alla directory nel sistema ospite sempre analizzando le informazioni riportate dal comando `docker inspect`.

Il principale svantaggio di montare i volumi in un ambiente di produzione è che il container a quel punto avrà accesso al file system host; ciò vuol dire che qualsiasi cosa scritta sui volumi condivisi sarà accessibile e questo significa anche che i percorsi dei file devono corrispondere al 100% tra produzione e sviluppo, il che non è sempre garantito. Questo non è sicuramente un grosso problema in fase di sviluppo, ma è qualcosa da tenere a mente quando ci si sposta in ambiente di produzione, dove accedere a un file system dovrebbe richiedere una certa gestione dei permessi.

Se sei un utente Windows, quanto sopra non funzionerà esattamente allo stesso modo. Ciò è dovuto al fatto che spesso Docker è in esecuzione all'interno di una macchina virtuale, il che vuol dire che non può vedere le directory host sul tuo computer. Esiste però una soluzione: invece di utilizzare il terminale di avvio, è possibile usare *Kitematic*, fornito all'interno di *Docker Toolbox*. In fondo all'elenco dei container, nell'angolo in basso a sinistra, c'è un pulsante *Docker CLI*. Facendo clic su di esso si ottiene una riga di comando personalizzata. Questa finestra sarà preconfigurata per utilizzare quanto visto finora, insieme alla possibilità di gestire cartelle, proprio come il terminale usato in precedenza.

Il secondo avvertimento che è doveroso fare è che, a causa della macchina virtuale, solo alcune cartelle sono “montabili”, di default. Tutto ciò che vive sotto `C:\Users` può essere montato all'interno di Docker, anche se dobbiamo usare un nome di percorso speciale per farlo funzionare. Per montare `C:\Users\Pippo\Progetti\webapp`, dovremo usare la sintassi seguente.

Listato 4.4 Montare un volume Windows.

```
root@vbox:/home/utente# docker run --ti --rm -v  
//c/Users/Pippo/Progetti/webapp:/opt/webapp ubuntu
```

Il percorso avrà tutti i simboli \ convertiti in /, e i due punti saranno rimossi e verrà aggiunto un doppio slash. Ciò consentirà di montare correttamente le directory su sistemi Windows.

Abbiamo detto che nella maggioranza dei casi d'uso di Docker è necessario conservare i dati tra i container; quando si arresta un container, tutti i dati al suo interno “tornano” al modo in cui erano all'avvio. Questo è ovviamente intenzionale!

Se vogliamo che i dati persistano, dobbiamo specificarlo. Possiamo farlo passando `-v [container]`, come visto nel primo esempio, con il comando `create` o `run`, e Docker selezionerà quel percorso come persistente. I volumi creati in questo modo rimarranno attivi per tutta la durata del container; se distruggiamo quest'ultimo, verrà distrutto anche il volume sottostante, quindi è bene fare molta attenzione.

Questo tipo di installazione funziona in maniera semplice, e ciò è perfetto per casi in cui i container devono ospitare più tipi di archiviazione, come nel caso dei database, o per conservare i log di un server. Come accennato in precedenza, è possibile combinare questi volumi di dati persistenti con volumi montati sul sistema host.

Condivisione di dati tra container

Tuttavia, possiamo portare l'idea di un volume di dati a un nuovo livello; dal momento che i container Docker sono solo strati di un file system e i volumi di dati sono persistenti, possiamo condividere i volumi tra container Docker.

Questa è la prima vera dimostrazione della potenza di Docker e dei container in generale. È possibile creare interi container che non fanno altro che archiviare i dati e non occupano più risorse che il solo spazio su disco.

È possibile utilizzare il comando `docker create` direttamente per generare un volume senza eseguirlo e specificare volumi di dati da generare contemporaneamente. Il comando seguente crea un container che specifica un volume montato nella cartella /root del sistema.

Listato 4.5 Creazione di un container con un volume.

```
root@vbox:/home/utente# docker create -v /root --name data busybox
```

Non dobbiamo preoccuparci di eseguire un comando all'interno di questo container al momento, quindi lasciamo quella parte in sospeso, poiché stiamo usando questo container per la gestione di alcuni dati, e ci basterà un container molto piccolo. In tal senso, il container `busybox` è perfetto per scopi come questo, proprio per la sua natura minimale, ma l'applicazione di questo esempio vale anche per casi più complessi.

Quanto appena visto non esegue il container, quindi non lo vedremo all'interno dell'elenco di quelli attivi, se eseguissimo `docker ps`.

Docker ci consente di collegare i volumi con altri container aggiungendo l'opzione `--volumes-from [name]`. Per esempio potremmo eseguire un'istruzione simile alla seguente.

Listato 4.6 Collegamento di un volume.

```
root@vbox:/home/utente# docker run --rm -ti --volumes-from data ubuntu /bin/bash
```

Qui abbiamo generato un container che ha utilizzato il volume `/data` e ha scritto un file all'interno della cartella `/root`, che in realtà è il volume del nostro container di dati. Quando usciamo dal container, possiamo comunque vedere il file che abbiamo archiviato nel volume di dati nella directory `/data`.

Una parentesi molto importante sulla questione relativa ai permessi: spesso sarà necessario impostare le autorizzazioni e la proprietà su un volume o inizializzare un volume con alcuni dati o file di configurazione predefiniti. Il punto fondamentale da tenere presente è che qualsiasi istruzione successiva all'istruzione `VOLUME` in un Dockerfile non sarà in grado di apportare modifiche a quel volume. Per esempio, il seguente Dockerfile non funzionerà come previsto.

Listato 4.7 Dockerfile con configurazione delle autorizzazioni errata.

```
FROM debian:wheezy
RUN useradd pippo
VOLUME /data
RUN touch /data/x
RUN chown -R pippo:pippo /data
```

Vogliamo che i comandi `touch` e `chown` vengano eseguiti sul file system dell'immagine, ma in realtà verranno eseguiti all'interno del volume di un container intermedio, utilizzato per creare il layer. Questo volume

verrà rimosso una volta completati i comandi, rendendo inutile l’istruzione. Il seguente Dockerfile, invece, funzionerà correttamente.

Listato 4.8 Dockerfile con configurazione delle autorizzazioni corretta.

```
FROM debian:wheezy
RUN useradd pippo
RUN mkdir /data && touch /data/*
RUN chown -R pippo:pippo /data
VOLUME /data
```

Quando un container viene avviato da questa immagine, Docker copia tutti i file dalla directory del volume nell’immagine nel volume del container. Ciò non accadrà se viene specificata una directory host per il volume (in modo che i file host non vengano sovrascritti accidentalmente).

Se per qualche motivo non è possibile impostare autorizzazioni e proprietà in un’istruzione `RUN`, è necessario farlo utilizzando uno script `CMD` o `ENTRYPOINT`, che viene eseguito dopo la creazione del container.

Esiste ancora un altro modo per inizializzare un volume tra il sistema host e un container, che abbiamo utilizzato prima senza però soffermarci troppo sulla sintassi; questo può essere fatto utilizzando il formato `-v HOST_DIR:CONTAINER_DIR`, come nel seguente esempio.

Listato 4.9 Montare un volume.

```
root@vbox:/home/utente# docker run -v /home/utente/data:/data debian ls /data
```

Questo monterà la directory `/home/utente/data` sul sistema host come volume `/data` all’interno del container. Tutti i file già esistenti nella directory `/home/utente/data` saranno disponibili all’interno del container; se la directory `/data` esiste già all’interno del container, il suo contenuto verrà nascosto dal volume. A differenza degli altri comandi, nessun file dall’immagine verrà copiato nel volume e il volume non verrà eliminato da Docker (ovvero, `docker rm -v` non rimuoverà un volume montato in una directory scelta dall’utente).

In questo caso, ossia quando in un volume viene utilizzata una directory host specifica, questo viene definito *bind mount*, come. Ciò è in qualche modo fuorviante, poiché tutti i volumi sono tecnicamente montati tramite un processo di *binding*: la differenza è che il punto di montaggio è reso esplicito anziché nascosto in una directory di proprietà di Docker.

Una pratica comune consiste nel creare container per la gestione dei dati, ossia il cui unico scopo è condividere dati con altri container. Il vantaggio principale di questo approccio è che fornisce un *namespace* semplice per i volumi che possono essere facilmente caricati usando l'opzione `--volumes-from`.

Per esempio, possiamo creare un container di dati per un database PostgreSQL con il seguente comando.

Listato 4.10 Creare un container per la gestione dei dati.

```
root@vbox:/home/utente# docker run --name datidb postgres      echo "Container per  
la gestione dei dati di Postgres!"
```

Questo creerà un container partendo dall'immagine `postgres` e inizializzerà tutti i volumi definiti nell'immagine prima di eseguire il comando `echo` e uscire. Non è necessario lasciare in esecuzione questi container, poiché ciò sarebbe solo uno spreco di risorse. Possiamo quindi usare questo volume da altri container con l'argomento `--volumes-from`, come nel seguente esempio.

Listato 4.11 Opzione `-volumes-from`

```
root@vbox:/home/utente# docker run -d --volumes-from datidb --name database1  
postgres
```

Normalmente non è necessario utilizzare un'immagine “minima” come `busybox` per il container di dati, ma è sufficiente usare la stessa immagine che viene utilizzata per il container che consumerà i dati; nell'esempio precedente, si potrebbe quindi utilizzare l'immagine di Postgres per creare un container di dati da utilizzare con il database Postgres.

L'uso della stessa immagine non occupa spazio aggiuntivo, se questa è già stata scaricata; inoltre dà all'immagine la possibilità di inizializzare il container con qualsiasi set di dati e garantisce che le autorizzazioni siano impostate correttamente.

Copiare i dati

Ci possono essere momenti in cui vuoi guardare un file all'interno di un container senza interrompere la sua esecuzione o le attività che sta svolgendo; un modo per farlo è quello di copiare i file da un container tramite l'istruzione `docker cp`. Per esempio, per copiare il file `access.log` da

un container Nginx in esecuzione, possiamo digitare quanto segue in una scheda qualsiasi della riga di comando.

Listato 4.12 Istruzione docker cp.

```
root@vbox:/home/utente# docker cp nginx /var/log/nginx/error/access.log /tmp
```

Che cosa abbiamo imparato

- La differenza tra un volume e un *bind mount*.
- Come creare e montare volumi a un container, sia tramite Dockerfile, sia tramite riga di comando.
- Come condividere i dati tra il volume di un container e un volume nel sistema host.
- Come controllare i dati di basso livello associati a immagini e container; ciò costituisce un modo eccellente per vedere che cosa sta succedendo al loro interno e come sono costruiti.
- Utilizzando il comando `docker inspect`, è possibile controllare le informazioni riguardanti un volume.
- Con il comando `docker attach` è possibile attaccare un processo al processo in esecuzione di un container.
- Come porre attenzione ai permessi per l'esecuzione di determinate istruzioni in un Dockerfile.
- Con `docker cp` è possibile copiare file da un container.

Gestire la rete

Le connessioni tra container sono un'attività piuttosto comune e vengono implementate utilizzando le reti; questo paragrafo farà riferimento, infatti, a tutto ciò che è inerente alla gestione della rete Docker, concetto introdotto nella versione 1.9. La gestione della rete consente di configurare le proprie reti, attraverso le quali i container possono comunicare tra loro, e non solo. In altre parole, questo integra la rete `docker0` creata di default con nuove reti gestite dall'utente. La configurazione della rete è altamente personalizzabile e permette a un diverso numero di container di lavorare in sinergia.

Affinché comunichino tra loro e con il mondo esterno tramite la macchina host, è necessario predisporre una configurazione di rete; Docker ne supporta diverse tipologie, ciascuna adatta a specifici casi d'uso.

Per esempio, la creazione di un'applicazione che viene eseguita su un singolo container Docker avrà un'impostazione di rete diversa rispetto a un'applicazione web con un cluster con database, applicazioni e bilanciatori di carico che si estendono a più container, i quali devono poter comunicare tra loro; inoltre, sarà quasi sicuramente necessario che numerosi clienti possano accedere ai servizi esposti dai diversi container, come per esempio nel caso di un'applicazione web.

Controllo delle interfacce

Quando si installa Docker, viene creata una rete bridge predefinita denominata `docker0`; ogni nuovo container viene automaticamente collegato a questa rete, a meno che non venga specificata una rete personalizzata.

Oltre a `docker0`, Docker crea automaticamente altre reti, come vedremo di seguito.

Driver di rete: host

Su questa rete non esiste alcun isolamento con i container, perché verso l'esterno è come se si trovassero tutti sulla stessa rete; in altre parole, se si utilizza questa modalità di rete, lo stack di rete di quel container non viene isolato dall'host Docker e il container non riceve un proprio indirizzo IP. Per esempio, se si eseguisse un container che si collega alla porta 80 e si utilizza la rete host, l'applicazione del container sarebbe disponibile sulla porta 80 dell'indirizzo IP del sistema host.

La rete in modalità host può essere utile per ottimizzare le prestazioni e anche in situazioni in cui un container deve gestire una vasta gamma di porte, in quanto non richiede la traduzione dell'indirizzo di rete (tramite NAT) e non viene creato alcun “userland-proxy”.

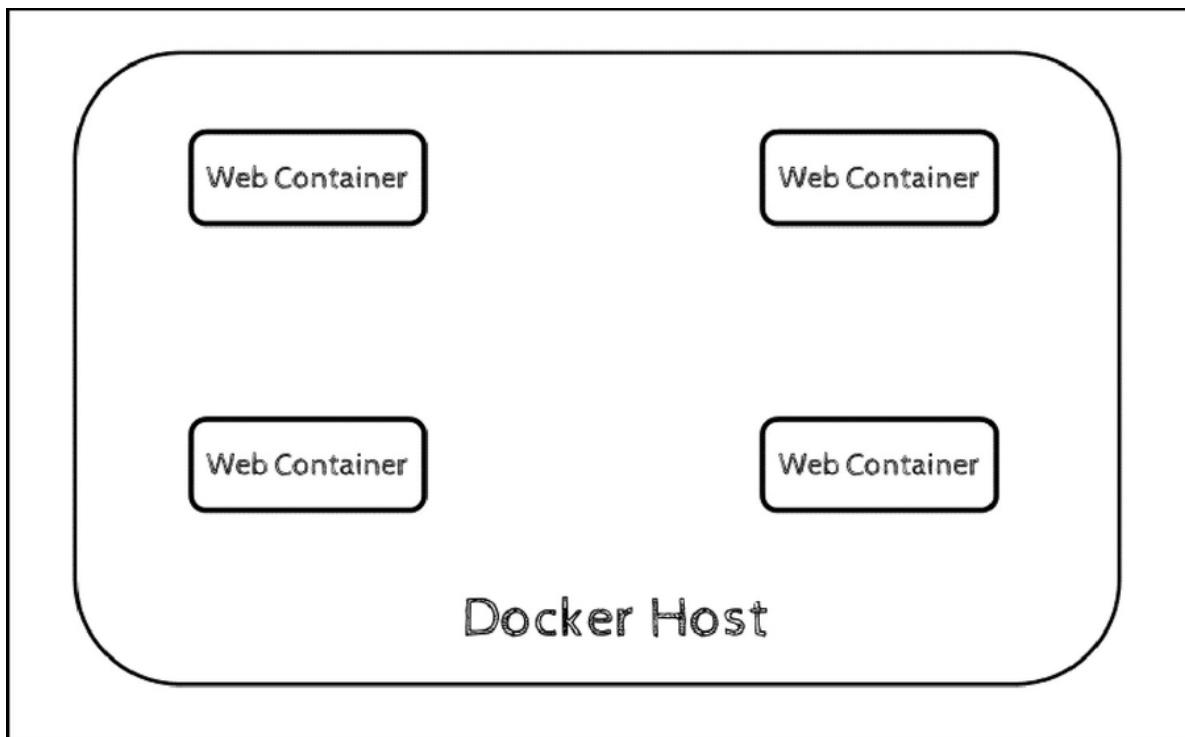


Figura 5.1 Esempio di una rete con driver host.

Il driver di rete dell'host funziona però solo su sistemi *Unix-based* e quindi non è supportato su Docker Desktop per macOS o per Windows.

Il caso più semplice di applicazione è quello di un container con un server Nginx; dopo che questo è messo in esecuzione, è possibile accedere alla pagina principale tramite <http://localhost:80> dal sistema su cui Docker è installato e visualizzare la pagina iniziale del server; questo è possibile proprio perché la rete tra il sistema host e il container è praticamente la stessa.

Driver di rete: bridge

È il tipo di rete più comune; è limitata ai container all'interno di un singolo host che esegue il Docker Engine. Questa tipologia è facile da creare, gestire e ha una risoluzione semplice dei problemi.

Affinché i container sulla rete bridge comunichino o siano raggiungibili dal mondo esterno, è necessario configurare la mappatura delle porte. Per esempio, immaginiamo di avere un container che esegue un servizio web sulla porta 80; poiché questo container è collegato alla rete bridge su una sottorete privata, una porta sul sistema host come 8000 deve essere mappata sulla porta 80 sul container, per consentire al traffico esterno di raggiungere il servizio web.

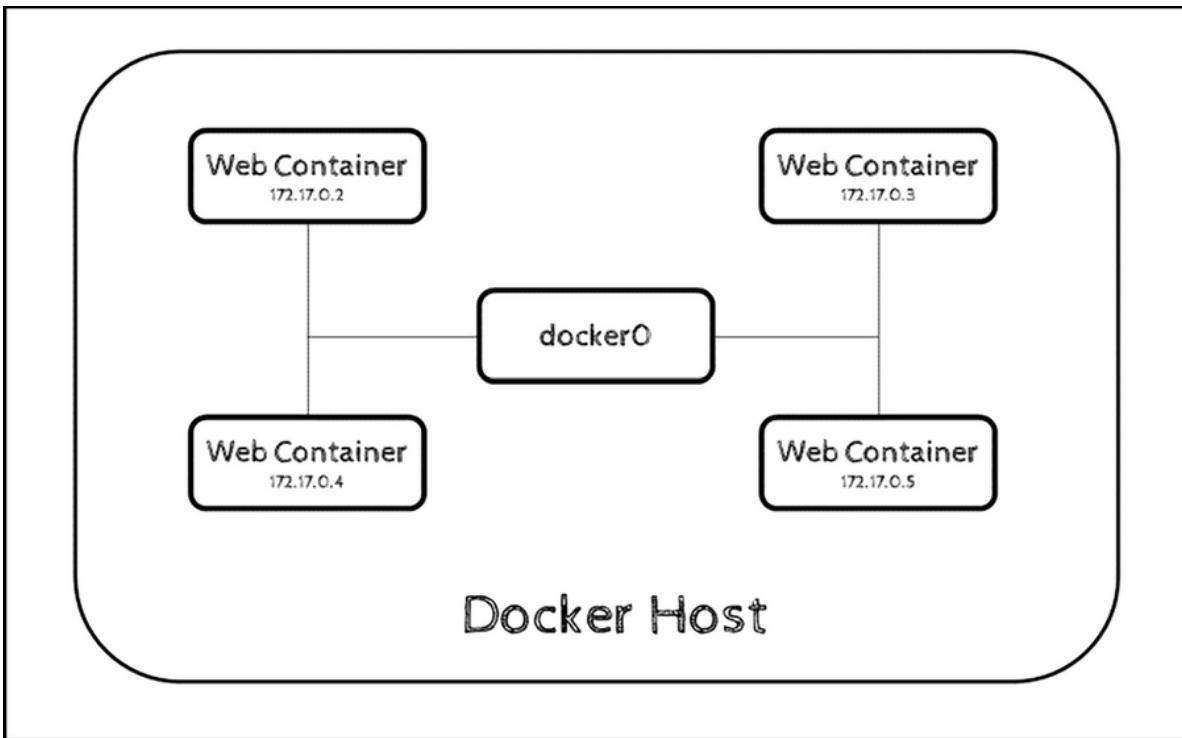


Figura 5.2 Esempio di una rete con driver bridge.

La *rete bridge* è dunque una rete interna predefinita e privata creata da Docker; pertanto, tutti i container ottengono un indirizzo IP interno e possono comunicare senza problemi, utilizzando tale IP. Le reti bridge vengono generalmente utilizzate quando le applicazioni vengono eseguite in container autonomi che devono poter comunicare; questo comporta che le reti bridge si applichino solo ai container in esecuzione sullo stesso sistema host.

Driver di rete: overlay

Le *reti overlay* collegano insieme più daemon Docker e quindi consentono ai servizi di comunicare tra loro. È possibile utilizzare delle reti overlay per facilitare la comunicazione tra un servizio e un container o tra due container su diversi daemon Docker. Questa strategia elimina la necessità di eseguire il routing a livello di sistema operativo tra questi container e consente di gestire il tutto tramite un unico driver di rete.

Questa rete si pone sopra le reti specifiche dell'host, consentendo ai container collegati alla rete del sistema di comunicare in modo sicuro

anche quando è abilitata la crittografia. Docker gestisce in modo trasparente il routing di ciascun pacchetto da e verso l'host e il container di destinazione.

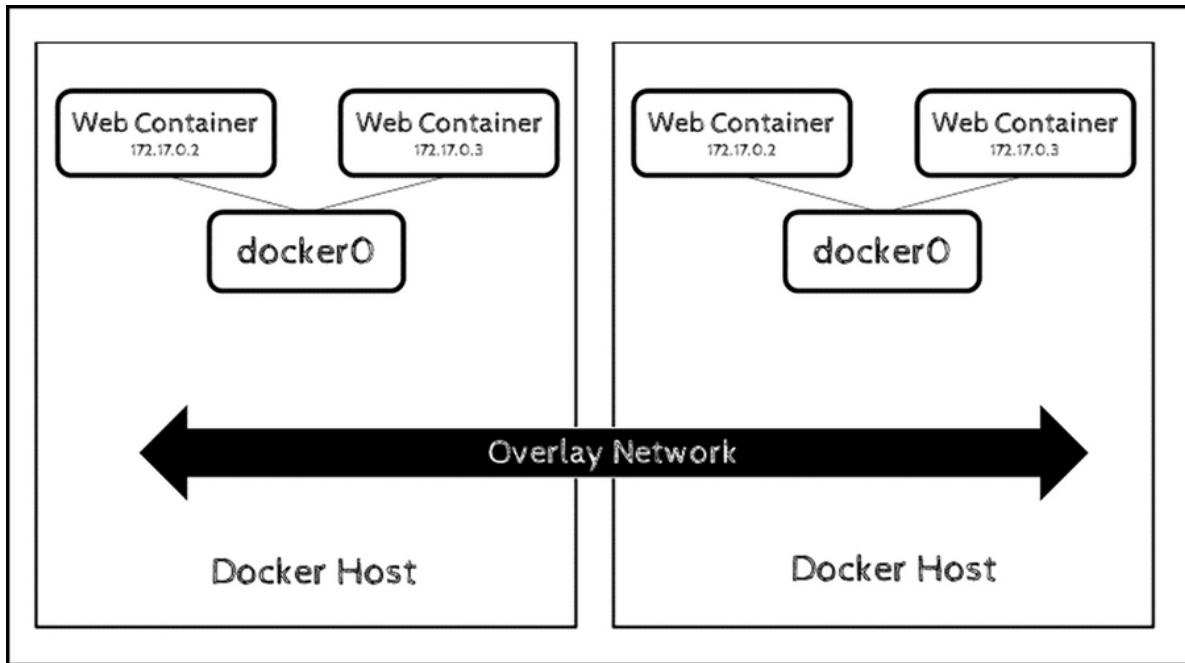


Figura 5.3 Esempio di una rete con driver overlay.

Driver di rete: macvlan

Questa tipologia consente di assegnare un indirizzo MAC a un container, facendolo apparire come fosse un dispositivo fisico sulla rete; in questo modo, il daemon Docker indirizza il traffico verso i container in base ai loro indirizzi MAC. Non è necessario l'uso della mappatura delle porte o della traduzione degli indirizzi di rete (NAT) e ai container può essere assegnato un indirizzo IP pubblico, accessibile dall'esterno. La latenza nelle reti macvlan è bassa, poiché i pacchetti vengono instradati ai container direttamente dal controller dell'interfaccia di rete (NIC) dell'host Docker.

Occorre notare che macvlan deve essere configurato *ad hoc* per il sistema host e offre il supporto per le interfacce di rete fisiche, secondarie e anche le interfacce di gruppo. Il traffico viene esplicitamente filtrato dai moduli del kernel host per ragioni di isolamento e sicurezza.

L'utilizzo di questo tipo di driver è la scelta migliore quando si ha a che fare con *applicazioni legacy* che prevedono di essere direttamente connesse alla rete fisica, anziché instradate attraverso lo stack di rete dell'host Docker.

Driver di rete: none

Se desideriamo disabilitare completamente lo stack di rete su un container, possiamo utilizzare l'opzione `--network none` all'avvio. All'interno del container, viene creato solo il dispositivo di *loopback*.

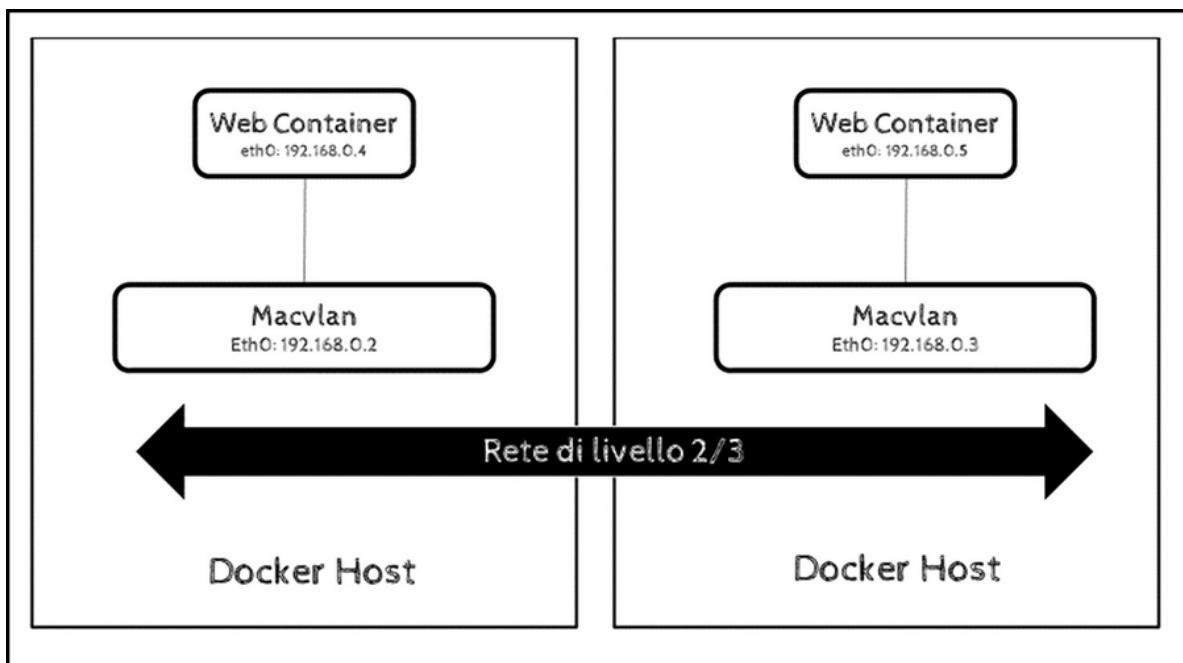


Figura 5.4 Esempio di una rete con driver macvlan.

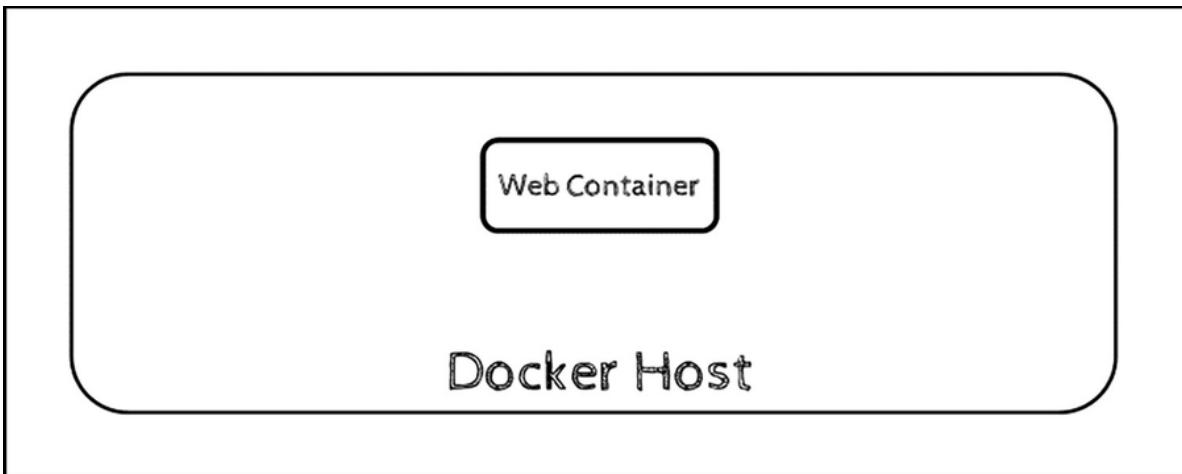


Figura 5.5 Esempio di una rete con driver none.

Fatte queste premesse, vediamo come applicare i diversi driver di rete in Docker.

Esportare in rete un container

Abbiamo già detto che un modo con il quale Docker crea l'illusione di un container che opera su una propria macchina è fornendo al processo il proprio indirizzo IP. Docker lo fa impostando un'interfaccia virtuale e collegandola alla rete della macchina host; come abbiamo accennato, questo bridge lato host si chiama `docker0`. A ciascun container gestito da questo bridge viene assegnato un proprio indirizzo IP. Tuttavia, Docker fornisce comunque l'isolamento del container.

Affinché un container fornisca un servizio basato su IP ad altri container o alle applicazioni, deve esporre la porta utilizzata dal servizio. Per esempio, un container che abbia un server web Apache predefinito deve esporre le porte 80 e 443, come illustrato nell'esempio Dockerfile seguente.

Listato 5.1 Dockerfile di un server web Apache.

```
FROM fedora:latest
RUN yum -y update; yum clean all
RUN yum -y install httpd
EXPOSE 80 443

ENTRYPOINT /usr/sbin/httpd -DFOREGROUND
```

Il Dockerfile precedente utilizza la parola chiave `EXPOSE` per definire una porta che verrà esposta dal container in esecuzione. Ma il solo uso della parola chiave `EXPOSE` non espone immediatamente la porta definita ad altri container o alle applicazioni sul sistema host.

Affinché tali entità abbiano accesso a questa porta, sono necessari dei passaggi aggiuntivi, per collegare o mappare la porta all'esterno del container.

- *Porte esposte*: il collegamento consente a un container di accedere alla porta esposta di un altro container operante sulla stessa macchina.
- *Mappatura delle porte*: la mappatura fornisce un meccanismo per mappare una porta esposta alle porte esterne della macchina *host*.

Immaginando, infatti, di eseguire questo primo container e di eseguirne un secondo (i due container saranno identici), abbiamo bisogno di creare un collegamento tra i due, per far sì che possano comunicare: possiamo farlo, per esempio, usando l'opzione `--link`, come nei Listati 5.2 e 5.3.

Listato 5.2 Esecuzione del primo container.

```
root@vbox:/home/utente# docker run -d --name=container-uno my-apache
```

Listato 5.3 Esecuzione del secondo container.

```
root@vbox:/home/utente# docker run -it --link=container-uno:cuno --  
name=container-due my-apache bash
```

Durante l'esecuzione del secondo container, avviamo una sessione TTY tramite bash per utilizzare la riga di comando; se provassimo a elencare tutte le variabili d'ambiente presenti in questo container, vedremmo che ve ne sono diverse che iniziano con “`cuno`”, ossia l'alias che è stato assegnato al container; in questo modo, ogni porta esposta dal primo container risulta accessibile dal secondo.

Questo però ci permette di mettere in comunicazione due container tra loro, ma non di esporre le porte al di fuori del sistema host: nella maggior parte dei casi, le reti Docker utilizzano sottoreti senza accesso dal mondo esterno. Per consentire alle richieste Internet di raggiungere un container, è necessario mappare le porte del container alle porte sull'host. Per esempio, una richiesta all'indirizzo `miaapp:8000`, verrà inoltrata a qualsiasi

servizio in esecuzione all'interno del container sulla porta 80, se in precedenza era stato definito un mapping dalla porta del sistema host 8000 alla porta del container 80.

Questo perché i container Docker possono connettersi al mondo esterno senza ulteriore configurazione, ma il mondo esterno non può connettersi ai container per impostazione predefinita.

Per esporre una porta da un container, in modo che sia disponibile tramite una porta sull'host, possiamo utilizzare l'opzione `-p` con il comando `docker run`; questo permette di esporre la porta del container alla stessa porta sull'host o a una diversa. Se un container è collegato a un altro container, solo le porte mappate esplicitamente all'host saranno però disponibili per i client. In altre parole, mentre il collegamento a un container crea porte in ascolto per l'altro container, tale azione non rende automaticamente tali porte disponibili per il sistema host.

I paragrafi seguenti illustrano come le porte dei container sono mappate con l'host su cui sono in esecuzione; vedremo anche come le porte mappate influenzano il modo in cui alcune porte all'interno dei container sono protette dall'accesso esterno.

Immaginiamo di avere due container con due web server qualsiasi (per semplicità useremo l'immagine di Nginx, ma lo stesso vale per qualunque altra immagine) collegati tra loro su un host chiamato `Host1`; `Host1` ha a disposizione il default bridge `docker0`. La porta 8080 del container C1 è esposta e la porta 80 del container C2 è mappata con la porta 80 dell'host.

Per far sì che `Host1` sia raggiungibile da un secondo host, chiamato `Host2`, e che `Host2` riesca a visualizzare la pagina web esposta dal container C2 sulla porta 80, dobbiamo compiere una serie di operazioni.

Come prima cosa, è necessario avviare il container C1 ed esporre la porta 8080 tramite il comando seguente.

Listato 5.4 Esecuzione del primo container.

```
root@vbox:/home/utente# $ docker run -d --name=C1 -p 8080:80 -w /var/www/html      -v /var/www/html:/var/www/html nginx
```

Poi dobbiamo avviare il container C2 esponendo la porta 80 mappata con la stessa porta dell'host, creando anche un collegamento tra i due container.

Listato 5.5 Esecuzione del secondo container.

```
$ docker run --name=C2 -d --link=C1:C1 -p 80:80 nginx
```

A questo punto verifichiamo che C1 abbia accesso a C2 tramite il bridge di default: utilizzando il comando `docker exec` all'interno del container C2 e usando il comando `curl` per verificare se è possibile visualizzare la pagina principale del server, dovremmo avere il seguente risultato.

Listato 5.6 Verifica di accesso da C2 alla porta di C1.

```
$ docker exec -it C2 bash  
bash-4.2# curl http://C1:80  
<!DOCTYPE html>  
<html>  
<head>  
<title>Welcome to nginx!</title>
```

...

Ora dobbiamo controllare che `Host1` sia accessibile da `Host2` tramite la porta 80 e che quindi anche `Host2` visualizzi correttamente la pagina di default; sarà sufficiente utilizzare la riga di comando ed eseguire il comando `curl Host1` sulla porta 80. Se il risultato è uguale a quello del precedente comando, quindi si visualizza il codice HTML della pagina di default di Nginx, si ha la dimostrazione che C2 ha accesso a C1 tramite la porta esposta.

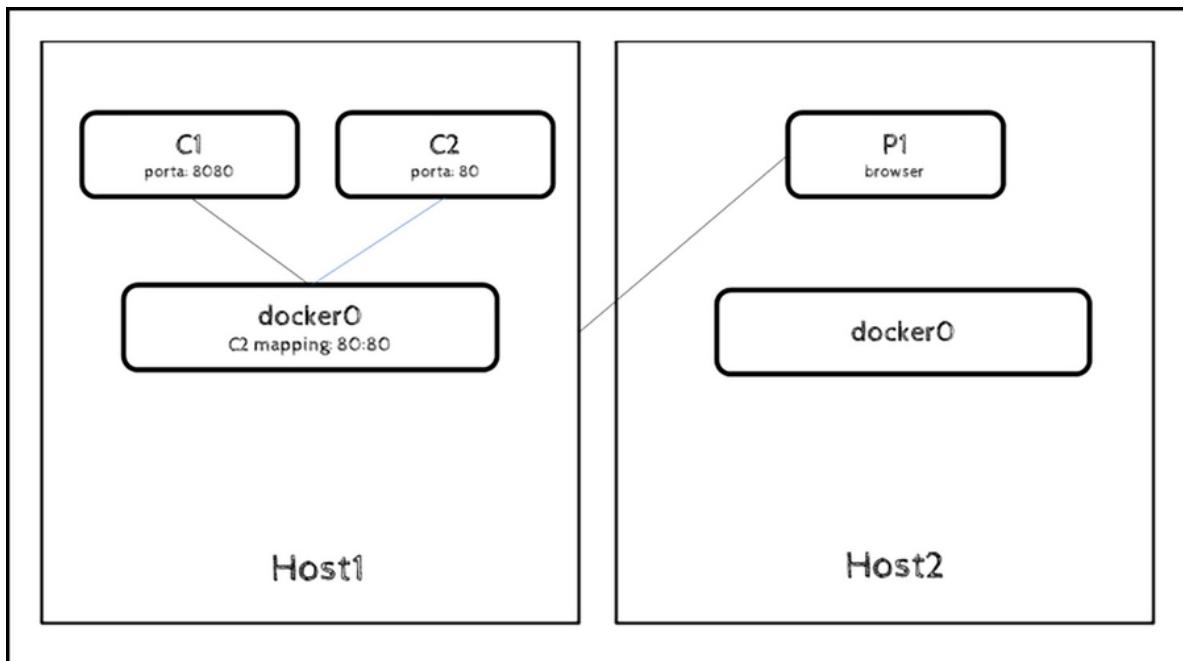


Figura 5.6 Schema della rete composta dai due sistemi host.

Usare il bridge di default

Per fare chiarezza su quanto accennato circa il bridge di default, vediamo che cos'è e come possiamo sfruttarlo appieno.

In termini di rete, una *rete bridge* è un dispositivo che inoltra il traffico tra i diversi segmenti di rete. Un bridge può essere un dispositivo hardware oppure un software in esecuzione all'interno del kernel di una macchina host.

Nel contesto di Docker, una rete bridge utilizza un software che consente ai container collegati alla stessa rete bridge di comunicare, fornendo al contempo l'isolamento dai container che non sono connessi a quella rete bridge. Il driver bridge di Docker configura automaticamente le regole per la connessione nel sistema host in modo che container operanti su reti bridge diverse non possano comunicare direttamente tra loro.

Le reti bridge si applicano ai container in esecuzione sullo stesso host su cui viene eseguito il daemon Docker. Per la comunicazione tra container in esecuzione su host diversi, è possibile gestire il routing a livello di sistema operativo oppure utilizzare una rete overlay, come vedremo a breve.

Per vedere che tramite la rete bridge di default due container sullo stesso host riescono a comunicare, eseguiamo due container di base e verifichiamo che sia possibile effettuare il `ping` dall'uno all'altro.

Per poter vedere la differenza tra il prima e il dopo l'esecuzione dei due container e quindi la configurazione della rete, eseguiamo il comando `docker network`, per mostrare che reti sono disponibili prima che i due container siano eseguiti.

Listato 5.7 Il comando `docker network` (prima).

```
root@vbox:~# docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
4989cf2dfef0    bridge    bridge      local
6e0bfb77dbc9    host      host       local
ae7d83c1b3e8    none      null      local
```

Viene elencata la rete `bridge` predefinita, insieme a `host` e `none`. Le ultime due non sono reti a tutti gli effetti, come abbiamo già visto, ma vengono utilizzate per avviare un container collegato direttamente allo stack di

rete del sistema host o per avviare un container senza dispositivi di rete collegati.

A questo punto, avviamo i due container a partire dall'immagine Ubuntu, che ci consente di utilizzare bash per verificare la connessione tra i due, e lo facciamo con i seguenti comandi: nei primi due avviamo i container con il nome di c1 e c2 e nel terzo verifichiamo che siano effettivamente stati avviati.

Listato 5.8 Il comando docker network (dopo).

```
root@vbox:~# docker run -dit --name c1 ubuntu bash
5fc801d56eee411930d8096c17a43037f1ee41574a42aee0c22ed34bc2ba6b32
root@vbox:~# docker run -dit --name c2 ubuntu bash
a3b8b5d037f151fe1eb3fbfc8f85820724a55771dee4e836c05130bf97e21a0b
root@vbox:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES
a3b8b5d037f1        ubuntu              "bash"              3 seconds ago     Up  3 seconds      c2
5fc801d56eee        ubuntu              "bash"              8 seconds ago     Up  8 seconds      c1
```

A questo punto, ispezioniamo il bridge, impiegando il comando `docker network inspect bridge`.

Listato 5.9 Il comando docker network inspect bridge.

```
root@vbox:~# docker network inspect bridge
[{"Name": "bridge", "Id": "4989cf2dfef0a54f2613e8e27b4b8bb90c947e59ac04f763699d34d6e5709a0e", "Created": "2020-04-05T17:55:08.473951603+02:00", "Scope": "local", "Driver": "bridge", "EnableIPv6": false, "IPAM": {"Driver": "default", "Options": null, "Config": [{"Subnet": "172.17.0.0/16", "Gateway": "172.17.0.1"}]}, "Internal": false, "Attachable": false, "Ingress": false, "ConfigFrom": {"Network": ""}, "ConfigOnly": false, "Containers": {""5fc801d56eee411930d8096c17a43037f1ee41574a42aee0c22ed34bc2ba6b32": {"Name": "c1", "EndpointID": "5b10b0de43409a61f231f45fc85213ccf023af83fd44361f1c4b7ae13c127"}}
```

```

        "b30",
        "MacAddress": "02:42:ac:11:00:02",
        "IPv4Address": "172.17.0.2/16",
        "IPv6Address": ""
    },
    "a3b8b5d037f151fe1eb3fbfc8f85820724a55771dee4e836c05130bf97e21a0b": {
        "Name": "c2",
        "EndpointID": "c987365e964f2ee85419493a2191ff217a861de4fbb163e50d95d0ed69398
de5",
        "MacAddress": "02:42:ac:11:00:03",
        "IPv4Address": "172.17.0.3/16",
        "IPv6Address": ""
    }
},
"Options": {
    "com.docker.network.bridge.default_bridge": "true",
    "com.docker.network.bridge.enable_icc": "true",
    "com.docker.network.bridge.enable_ip_masquerade": "true",
    "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
    "com.docker.network.bridge.name": "docker0",
    "com.docker.network.driver.mtu": "1500"
},
"Labels": {}
}
]

]

```

Come possiamo vedere, ai due container `c1` e `c2` sono stati assegnati due IP, rispettivamente `172.17.0.2` e `172.17.0.3`, mentre il gateway predefinito ha l'IP `172.17.0.1`.

Se lanciamo il comando `docker exec` per utilizzare la riga di comando nel primo container e eseguiamo il comando `ip addr show`, vedremo che le interfacce di rete sono come descritte dal comando precedente.

Listato 5.10 Il comando docker exec su c1.

```

root@vbox:~# docker exec -it c1 bash
root@5fc801d56eee:/app# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
62: eth0@if63: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
        valid_lft forever preferred_lft forever

```

A questo punto, possiamo interrogare con `ping` il container `c2` sull'indirizzo IP `172.17.0.3` e verificare che si connettano.

Listato 5.11 Il comando ping su c1.

```
root@5fc801d56eee:/app# ping 172.17.0.3
PING 172.17.0.3 (172.17.0.3) 56(84) bytes of data.
64 bytes from 172.17.0.3: icmp_seq=1 ttl=64 time=0.059 ms
--- 172.17.0.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.059/0.059/0.059/0.000 ms
```

Effettuando la stessa prova, ma a partire dal container `c2` verso il container `c1`, il risultato è lo stesso.

NOTA

I comandi `ping` e `ip` non sono installati all'interno dell'immagine utilizzata; sarà tuttavia sufficiente utilizzare i comandi `apt install -y iputils-ping` e `apt install -y iproute2` per installarli, dopo aver effettuato gli aggiornamenti di sistema tramite `apt update` e `apt upgrade`.

Associare le porte a una rete

Abbiamo visto che i container Docker possono connettersi al mondo esterno senza ulteriore configurazione, ma il mondo esterno non può connettersi ai container per impostazione predefinita.

Una *rete bridge* viene creata di default quando si installa Docker; ogni connessione in uscita sembra provenire dall'host, ma in realtà Docker crea una delle regole di mascheramento personalizzate che ne gestiscono la redirezione.

Se al comando `docker run` aggiungiamo l'opzione `-P` (oppure `--publish-all=true`), Docker identifica ogni porta esposta tramite Dockerfile con le istruzioni `EXPOSE` ed esegue il mapping di tutte queste porte su una porta host tra quelle a disposizione all'interno di un determinato range di porte. È possibile trovare la configurazione di queste porte (di solito vanno dalla 32768 alla 61000) nel file `/proc/sys/net/ipv4/ip_local_port_range` (in un sistema *Unix-based*). Per vedere in che modo queste sono state gestite, possiamo eseguire il comando `docker port`, il quale ispeziona le porte create da Docker.

È anche possibile specificare le porte, senza che queste vengano associate a delle porte dell'host in maniera casuale rispetto al range previsto; nel fare ciò, supponiamo di voler esporre la porta 8080 del container sulla porta 80 dell'host (supponendo che la porta non sia in uso). Possiamo aggiungere al comando `docker run` le opzioni `-p 80:8080` (oppure `--publish=80:8080`), come nell'esempio seguente.

Listato 5.12 Associare una porta host a un container.

```
root@vbox:~# docker run -p 80:8080 nginx
```

Per impostazione predefinita, Docker espone le porte del container sull'indirizzo IP 0.0.0.0 (che corrisponde a qualsiasi IP sul sistema). Chiaramente, è possibile specificare a quale IP deve essere associato il container: per esempio, per associare l'indirizzo IP 10.1.2.3:80, è possibile lanciare il seguente comando.

Listato 5.13 Associare una porta host e un indirizzo IP a un container.

```
root@vbox:~# docker run -p 10.1.2.3:80:8080 nginx
```

Definire una rete custom

È possibile creare più reti tramite Docker e aggiungere dei container a una o più reti; questi container potranno comunicare all'interno di una stessa rete, ma non tra reti diverse. Un container collegato a più reti può connettersi con tutti i container che appartengono alle diverse reti, e permettere così la connessione tra diverse reti, creando una sorta di "hub".

Le reti *bridge* offrono la soluzione più semplice per creare la propria rete Docker; supponendo di dover creare una rete che isoli un container dagli altri, possiamo eseguire il comando `docker network create`, così da creare una nuova rete, per poi associarvi un container a scelta. Nell'esempio seguente, vediamo come creare una rete chiamata `my_network` e come associare il container `nginx`.

Listato 5.14 Creare una rete bridge.

```
root@vbox:~# docker network create --driver bridge my_network  
123ab...
```

```
root@vbox:~# docker run --net=my_network --name=nginx
```

Eseguendo il comando `docker network ls`, vedremo elencata la rete appena creata tra quelle disponibili, in aggiunta a quelle presenti di default descritte in precedenza.

Listato 5.15 Elenco delle reti.

```
root@vbox:~# docker network ls  
NETWORK ID      NAME      DRIVER
```

```

fa1ff6106123      bridge      bridge
803369ddc1ae     host        host
123abcdef556     my_network  bridge
01cc882aa43b     none        null

```

Per associare più container a una stessa rete, basta eseguire sempre il comando `docker run` passando come parametro la rete alla quale si desidera che questi vengano associati; per poter verificare in qualunque momento le informazioni della rete e l'elenco dei container a essa associati, possiamo utilizzare l'istruzione `docker network inspect`, così da avere a disposizione tutto il dettaglio della rete creata.

Listato 5.16 Dettaglio della rete `my_network`.

```

root@vbox:~# docker network inspect my_network
[{"Name": "my_network", "Id": "123abcdef556...", "Scope": "local", "Driver": "bridge", "EnableIPv6": false, "IPAM": {"Driver": "default", "Options": {}, "Config": [{"Subnet": "172.18.0.0/16", "Gateway": "172.18.0.1/16"}]}, "Internal": false, "Containers": {"b4ba8821a2fa3d602ebf2ff114b4dc4a9dbc178784dad340e78210a1318b717b": {"Name": "nginx", "EndpointID": "4434c2c253afed44898aa6204a1ddd9b758ee66f7b5951d93ca2fc6dd610463c", "MacAddress": "02:42:ac:12:00:02", "IPv4Address": "172.18.0.2/16", "IPv6Address": ""}}, "Options": {}, "Labels": {}}]

```

Qualsiasi altro container creato su questa rete può connettersi immediatamente a qualsiasi altro container presente su questa stessa rete, e lo vedremo elencato nella lista dell'attributo `Containers`; la rete isola i container da altre reti (comprese quelle esterne), ma è possibile esporre e

pubblicare le porte dei container sulla rete, consentendo ad alcune porzioni della rete di accedere a una rete esterna.

Creare una rete multi-host

Fintantoché vengono creati e utilizzati dei container su un singolo host, le tecniche introdotte finora sono sufficienti. Tuttavia, se la capacità di un singolo host non è sufficiente per gestire il carico di lavoro oppure si desidera una maggiore resilienza, è necessario *ridimensionare in maniera orizzontale i container*.

Quando si esegue il ridimensionamento in questo modo, si crea una rete di macchine, nota anche come *cluster*: in che modo però permettere il dialogo tra container operanti su host distinti? Come controllare le comunicazioni tra i container e il mondo esterno? Come mantenere lo stato, le assegnazioni di indirizzi IP in modo coerente in un cluster? Quali sono i punti di integrazione con l'infrastruttura di rete esistente? Come gestire le politiche di sicurezza?

Per rispondere a queste domande, nel resto di questo paragrafo esamineremo le tecnologie per la gestione di rete di container multi-host. Poiché diversi casi d'uso e ambienti hanno requisiti diversi, non verranno fornite raccomandazioni per un particolare progetto o prodotto; verranno forniti solo gli strumenti necessari per essere consapevoli dei compromessi da attuare durante la creazione di una rete come questa, così da prendere una decisione ragionata e consapevole.

In breve, Docker stesso offre il supporto per *reti overlay*, ossia per la creazione di una rete distribuita tra host posti sopra rispetto alla rete specifica dell'host stesso, oltre che plugin di rete per provider di terze parti.

Vediamo come creare una *rete overlay*, ossia una rete sopra un'altra rete.

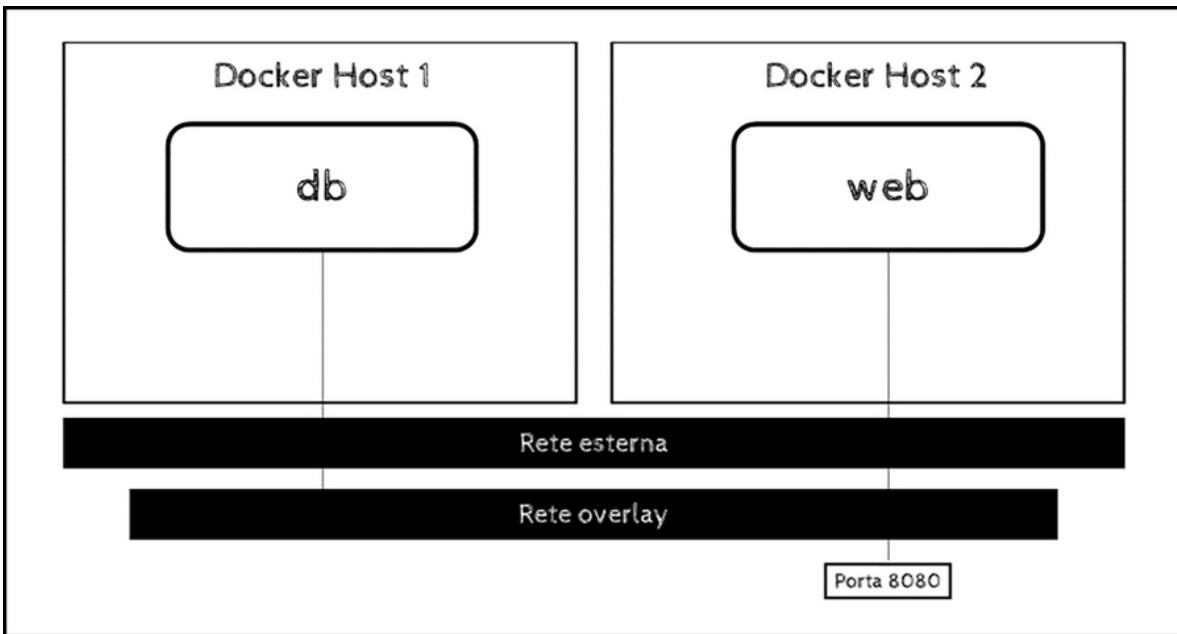


Figura 5.7 Rappresentazione di una rete overlay.

Se si desidera creare una rete multi-host nativa, è necessario creare una rete overlay e questa tipologia di rete richiede un servizio di gestione per sistemi distribuiti che mantenga le informazioni relative a coppie chiave-valore per l'associazione di porte fra host diversi, come Consul, Etcd o ZooKeeper. È necessario installare e configurare questo servizio *prima* di creare la rete, e gli host Docker devono poter comunicare con il servizio scelto.

Questo tipo di rete è particolarmente adatta quando vengono creati dei *cluster*, come nel caso di Docker Swarm o Kubernetes; rimandiamo quindi gli esempi pratici ai paragrafi che li tratteranno.

Esistono numerose opzioni di rete di container multi-host che vengono spesso utilizzate nella pratica, soprattutto nel contesto di Kubernetes (che vedremo più avanti). Queste includono:

- *Flannel* di CoreOS;
- *Metaswitch's Project*;
- *OpenVPN*;
- *Open vSwitch*;
- *Weave Net*.

Flannel

Si tratta di una rete virtuale che assegna una sottorete a ciascun host da utilizzare durante l'esecuzione dei container. Ogni container, o pod, nel caso di Kubernetes, ha un IP unico e inistradabile all'interno del cluster. Il vantaggio è che riduce la complessità della mappatura delle porte; il progetto Atomic di Red Hat utilizza questa opzione.

Metaswitch's Project

Il progetto Calico di Metaswitch utilizza il routing IP standard - per essere precisi, il Border Gateway Protocol, spesso abbreviato in BGP - e strumenti di rete per fornire una soluzione di livello 3 (vedi il modello OSI), mentre la maggior parte delle soluzioni di rete che vedremo crea una rete overlay incapsulando il traffico di livello 2 in un livello superiore.

La modalità operativa primaria non richiede incapsulamento ed è progettata soprattutto per l'utilizzo nei *data center* in cui l'organizzazione ha il controllo sulla struttura della rete fisica.

OpenVPN

OpenVPN consente di creare reti private virtuali usando il protocollo TLS; queste VPN possono anche essere utilizzate per connettere in modo sicuro dei container tramite la rete pubblica.

Open vSwitch

È uno switch virtuale multistrato progettato per consentire l'automazione della rete attraverso un'estensione programmatica, ossia che supporti interfacce e protocolli di gestione standard, come NetFlow, IPFIX, LACP e 802.1ag. Inoltre, è progettato per supportare la distribuzione su più server fisici e viene utilizzato nella distribuzione OpenShift Kubernetes di Red Hat e altre.

WeaveNet

Viene creata una rete virtuale che collega i container Docker distribuiti su più host. Le applicazioni utilizzano la rete come se i container fossero tutti collegati allo stesso switch di rete, senza la necessità di configurare la mappatura o i collegamenti delle porte. I servizi forniti dai container di applicazioni sulla rete Weave possono essere resi accessibili al mondo esterno, indipendentemente da dove sono in esecuzione.

Allo stesso modo, i sistemi interni esistenti possono essere esposti ai container dell'applicazione indipendentemente dalla loro posizione; Weave può gestire i firewall e operare in reti parzialmente connesse. Il traffico può essere crittografato, consentendo agli host di essere connessi attraverso reti non sicure.

Che cosa abbiamo imparato

- Come funziona la rete all'interno di Docker.
- Quali tipi di driver esistono e che differenze vi sono.
- Come esporre un container in una rete e renderlo visibile ad altri container nella stessa rete.
- Come associare delle porte a una rete.
- Come definire una propria rete e associarvi dei container.
- Cos'è una rete multi-host e quali alternative sono disponibili.

Capitolo 6

Docker Hub

Docker Hub è un servizio di registro basato su cloud che consente di scaricare immagini Docker create dalla community; oltre a scaricare immagini, è anche possibile caricare proprie immagini e renderle pubbliche. Oltre ai repository pubblici e privati, fornisce anche build automatizzate, account dell'organizzazione e l'integrazione con soluzioni di controllo del codice sorgente come Github e Bitbucket. DockerHub può anche scansionare automaticamente le immagini nei repository privati, alla ricerca di vulnerabilità, producendo un rapporto che descrive nel dettaglio i problemi rilevati in ogni livello dell'immagine, elencate per gravità (critica, maggiore o minore).

Diversi repository privati, build parallele e scansioni della sicurezza delle immagini sono disponibili solo con abbonamenti a pagamento.

In questo capitolo vedremo che cosa sono i registri, le principali differenze, come scaricare e utilizzare diverse immagini presenti su Docker Hub, e come gestire i repository.

Registri pubblici

Docker fornisce un registro di immagini pubbliche che la community desidera condividere; tra queste vi sono le immagini ufficiali per le distribuzioni Linux, container WordPress pronti all'uso e molto altro.

Se uno sviluppatore ha lavorato su immagini che possono essere pubblicate, il posto migliore è sicuramente un registro pubblico, come Docker Hub. Fornisce infatti migliaia registri di immagini Docker centralizzati ai quali è possibile accedere in qualsiasi momento, oltre a garantire un ottimo metodo per archiviare immagini private.

Docker Hub ha un'interfaccia utente molto semplice (vedi la Figura 6.1) e la possibilità di separare le autorizzazioni di accesso al team o di gestire gli utenti che possono accedere ai registri privati.

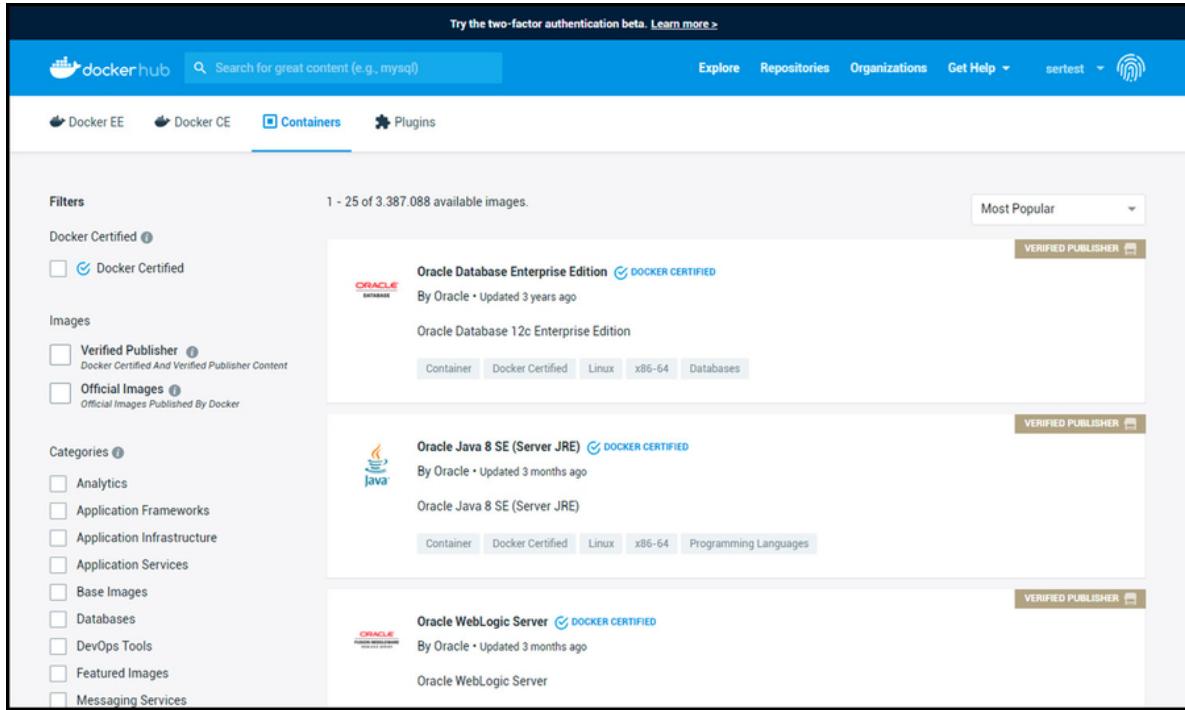


Figura 6.1 Pagina principale di Docker Hub.

Definiamo però che cos’è un registro: un registro Docker è un sistema di archiviazione e distribuzione per le immagini Docker, dove la stessa immagine potrebbe avere versioni diverse, identificate tramite tag.

Un registro Docker è organizzato in repository, e ognuno di essi contiene tutte le versioni di un’immagine. Il registro consente agli utenti Docker di estrarre le immagini localmente, nonché di inviare nuove immagini al registro (quando sono applicabili le autorizzazioni di accesso adeguate).

Per impostazione predefinita, il Docker Engine interagisce con Docker Hub, istanza del registro pubblico di Docker; tuttavia, è possibile eseguire *on-premise* il registro Docker open source, nonché una versione supportata commercialmente denominata *Docker Trusted Registry*.

Un repository pubblico è accessibile a chiunque esegua Docker e i nomi delle immagini includono l’organizzazione o il nome utente. Per esempio, eseguendo il comando `docker pull jenkins/jenkins`, verrà estratta l’immagine del server `jenkins` con il tag più recente appartenente

*****ebook converter DEMO

Watermarks*****

all’organizzazione Jenkins. Su Docker Hub sono disponibili centinaia di migliaia di immagini pubbliche; i repository privati limitano l’accesso al creatore del repository o ai membri della sua organizzazione.

Docker Hub supporta repository ufficiali, che includono immagini verificate per sicurezza e best practice; questi non richiedono la definizione un’organizzazione o di nome utente, come nel caso di nginx.

Docker Hub può eseguire build automatizzate di immagini, se il repository è collegato a un repository di controllo del codice sorgente che contiene un contesto di build (ossia il Dockerfile e tutti i file presenti nella stessa cartella). L’operazione di *commit*, che vedremo a breve, nel repository di origine attiverà un’operazione di build su Docker Hub.

Per le aziende che utilizzano fortemente Docker, il principale svantaggio di questi registri è che non sono locali alla rete su cui viene distribuita l’applicazione: ciò significa che potrebbe essere necessario riportare ogni livello di ogni distribuzione su Internet per distribuire un’applicazione. Le latenze di Internet hanno un impatto reale sulle distribuzioni di software e le interruzioni che incidono su questi registri potrebbero avere conseguenze sulla capacità di un’azienda di implementare senza problemi la soluzione nei tempi previsti. Ciò è mitigato da una buona progettazione delle immagini, in cui si creano strati sottili e facili da distribuire.

Alternative disponibili

Docker Hub non è l’unico registro pubblico a disposizione, se stai cercando dove ospitare i tuoi repository privati nel cloud. Al momento della scrittura di questo manuale, il principale concorrente è *Quay.io*, che offre alcune funzionalità in più rispetto a Docker Hub. Quando i principali strumenti Docker stavano guadagnando popolarità, Docker Hub, infatti, non esisteva. Per colmare questo vuoto evidente nella community, è stato creato Quay.io. Da allora, Quay.io è stato replicato da CoreOS Quay ed è stato utilizzato per creare il prodotto *CoreOS Enterprise Registry*.

Altre società ospitano registri Docker online a pagamento per uso pubblico; un esempio è rappresentato dai fornitori di servizi cloud, come

AWS e Google, che offrono anche servizi di hosting di container, commercializzando l'elevata disponibilità dei loro registri.

Di seguito vengono riportate alcune alternative a disposizione tra cui scegliere.

- *Amazon Elastic Container Registry* (ECR) si integra con il servizio IAM (*AWS Identity and Access Management*) per l'autenticazione. Supporta solo repository privati e non fornisce la creazione automatica di immagini.
- L'autenticazione di *Google Container Registry* (GCR) si basa sulle autorizzazioni del servizio di archiviazione cloud di Google. Supporta solo repository privati e fornisce build di immagini automatizzate tramite l'integrazione con i repository di Google Cloud Source, GitHub e Bitbucket.
- *Azure Container Registry* (ACR) supporta i registri multi-area e si autentica con Active Directory. Supporta solo repository privati e non fornisce la creazione automatica di immagini.
- *CoreOS Quay* supporta l'autenticazione OAuth e LDAP. Offre repository pubblici e privati , gratuiti e a pagamento, scansione automatica della sicurezza e build automatizzate di immagini tramite l'integrazione con GitLab, GitHub e Bitbucket.
- Il registro *Docker privato* supporta l'autenticazione OAuth, LDAP e Active Directory. Offre repository sia privati sia pubblici, gratuitamente fino a tre repository (privati o pubblici).

Registri privati

L'altra opzione che molte aziende considerano è quella di ospitare internamente dei registri privati delle immagini Docker. Prima che esistesse il registro pubblico per Docker, gli sviluppatori Docker hanno rilasciato il progetto del registro docker su GitHub. Il registro docker è un daemon Python senza GUI che può interagire con il client Docker per supportare il *push*, il pull e la ricerca di immagini. Inizialmente non supportava alcuna forma di autenticità e, oltre all'archiviazione di file in locale, il registro Docker open source ora supporta S3, Azure e alcuni altri sistemi di archiviazione.

Un altro forte concorrente nello spazio del registro privato è il *CoreOS Enterprise Registry*; quando CoreOS acquistò Quay.io, prese rapidamente il codice sorgente e lo rese disponibile come container Docker, facilmente implementabile. Questo prodotto offre sostanzialmente le stesse funzionalità su Quay.io, ma può essere distribuito internamente. Viene encapsulato come una macchina virtuale che è possibile eseguire come fosse un sistema fisico, e supporta la stessa interfaccia utente e le stesse interfacce del Quay.io pubblico.

Nel dicembre 2014, Docker ha annunciato che stava lavorando allo sviluppo di *Docker Hub Enterprise* (DHE), che consentirà alle organizzazioni di avere un registro delle immagini *on-premise* supportato da Docker nel loro data center o ambiente cloud.

Creare un account

Per poter pubblicare le proprie immagini o per creare dei registri privati su Docker Hub, è necessario avere un account; per registrarsi, è sufficiente collegarsi al sito <https://hub.docker.com> e fare clic su *Sign Up*. Dopo aver inserito le informazioni richieste nella Figura 6.2, sarà possibile effettuare il login.

Creare un repository

A questo punto siamo pronti per creare il primo repository; per farlo, facciamo clic su *Create repository* in alto a destra nella pagina *Repository* del sito Docker Hub e compiliamo le informazioni richieste, come il nome, la descrizione e il tipo di visibilità; quest'ultima informazione è estremamente importante, perché riguarda il modo in cui verrà gestito il repository e se questo sarà messo a disposizione di altri utenti o meno (Figura 6.3).

Chi ha un account gratuito, avrà a disposizione un solo repository privato; per aggiungerne altri, è necessario pagare una quota mensile di circa 7 dollari ogni cinque repository privati. Inoltre è possibile collegare un account GitHub o BitBucket, che permette di effettuare la build automatica delle immagini a partire dal codice sorgente presente in un repository esterno di uno di questi servizi, così come di effettuare

automaticamente l'operazione di *push* dell'immagine sui repository Docker.

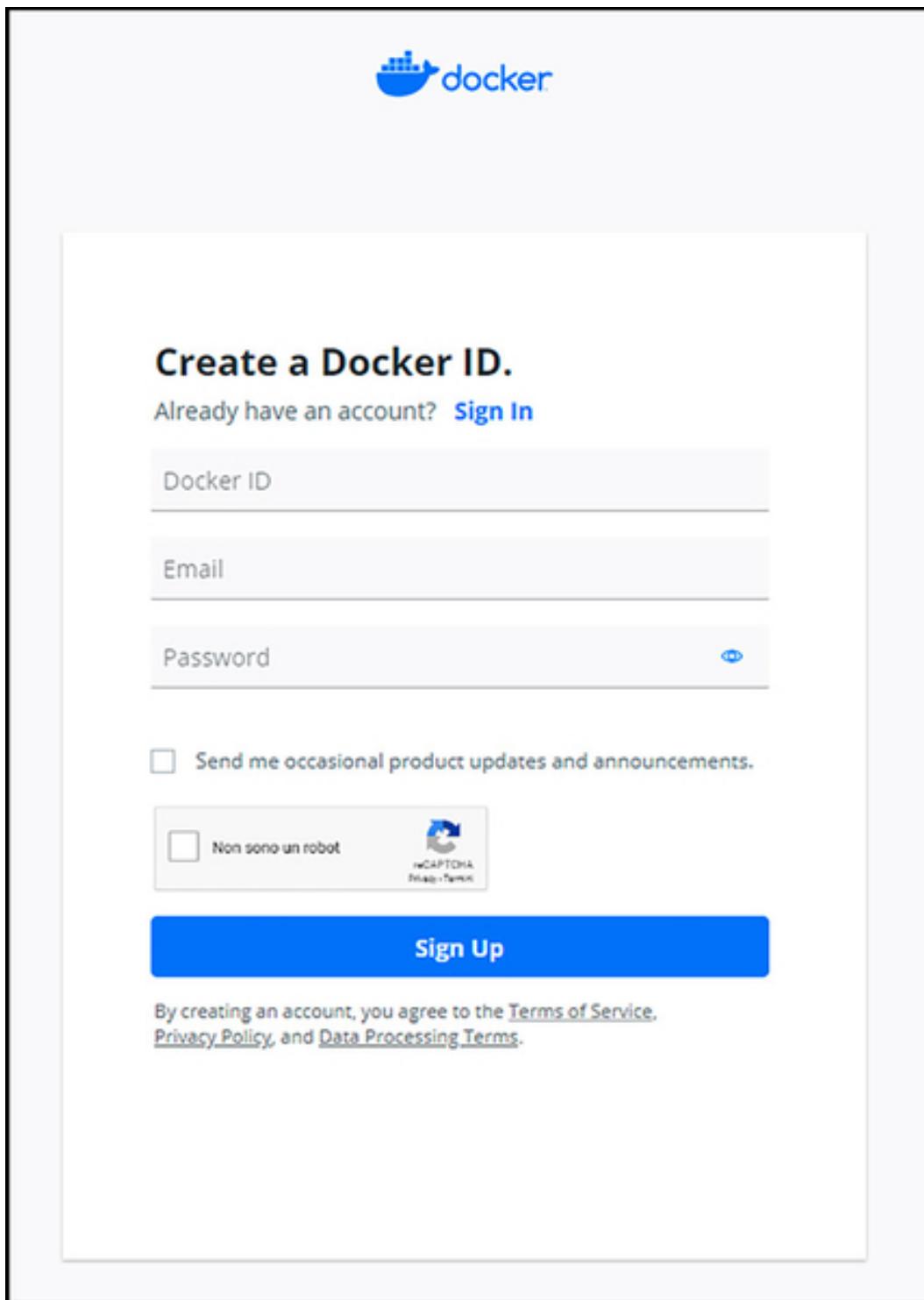


Figura 6.2 Creazione di un account su Docker Hub.

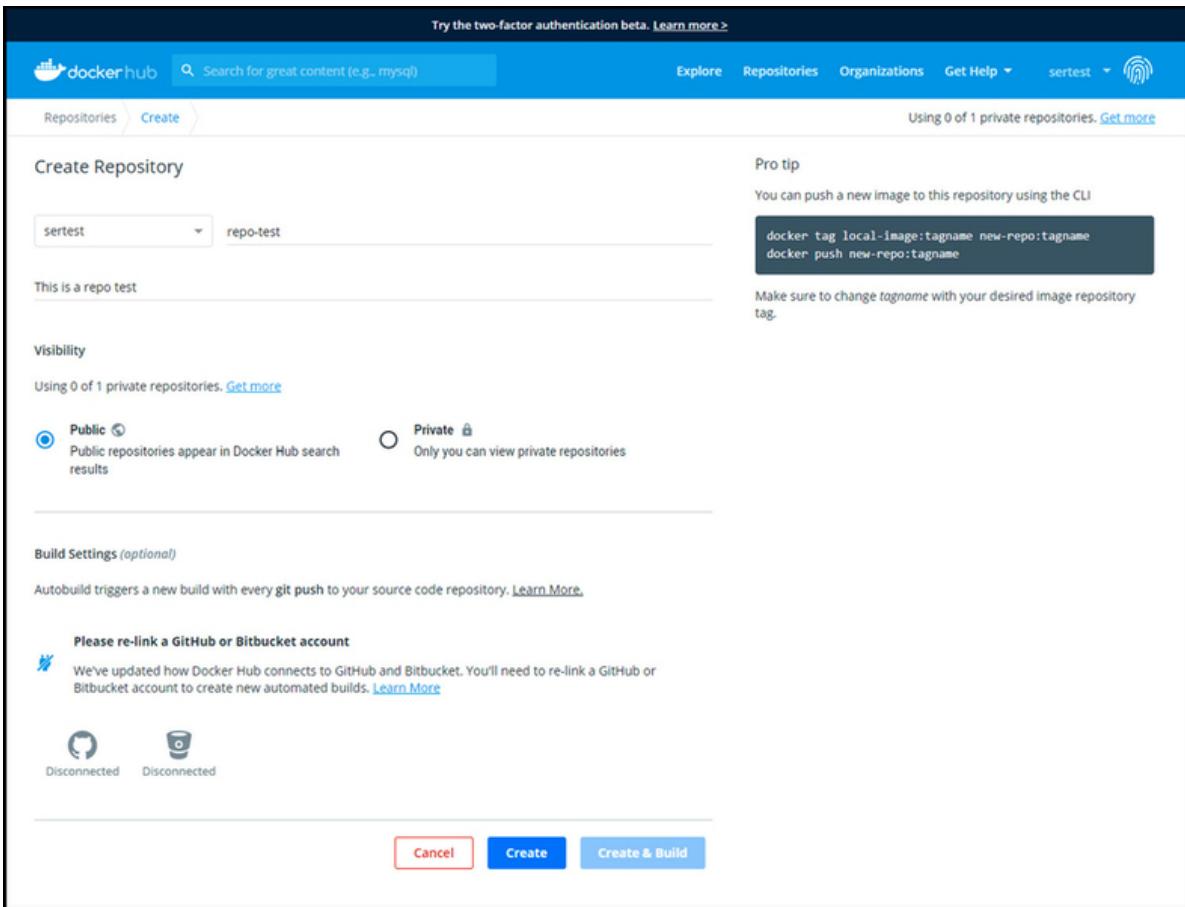


Figura 6.3 Creazione di un repository.

Dopo aver fatto clic su *Create*, il repository sarà disponibile nell’elenco principale associato all’utente.

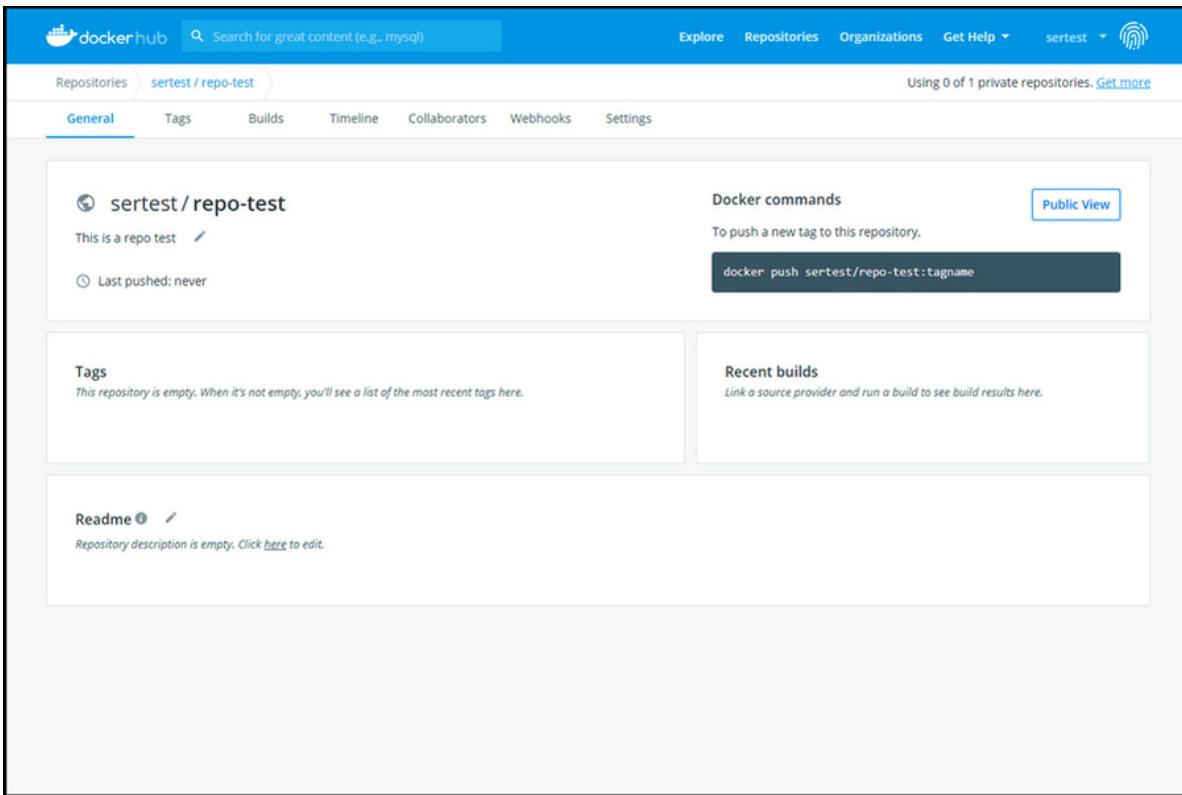


Figura 6.4 Riepilogo del repository creato.

Push di un'immagine

È possibile inviare una nuova immagine al repository utilizzando l'interfaccia a riga di comando; è fondamentale assicurarsi di cambiare *tagname* con il tag del repository di immagini desiderato.

Immaginiamo di avere a disposizione un semplice Dockerfile come il seguente.

Listato 6.1 Esempio di Dockerfile.

```
FROM busybox
CMD echo "Ciao! Questa è la mia prima immagine Docker."
```

Dopo aver effettuato la build dell'immagine e aver avviato il relativo container per verificare che non ci siano errori, è possibile effettuare il *push* dell'immagine sul repository Docker Hub creato nel paragrafo precedente. Per farlo, andremo a lanciare i seguenti comandi.

*****ebook converter DEMO

Watermarks*****

Listato 6.2 I comandi docker build e docker run.

```
root@vbox:/home/busybox# docker build -t USERNAME/repo-test .

Sending build context to Docker daemon 2.048kB

Step 1/2 : FROM busybox

    latest: Pulling from library/busybox
    e2334dd9fee4: Pull complete

    Digest: sha256:a8cf7ff6367c2afa2a90acd081b484cbded349a7076e7bdf37a05279f276bc12

    Status: Downloaded newer image for busybox:latest

    --> be5888e67be6

Step 2/2 : CMD echo "Ciao! Questa è la mia prima immagine Docker."
    --> Running in 6014ebce2511

    Removing intermediate container 6014ebce2511
    --> 5aa6baf3bb2c

    Successfully built 5aa6baf3bb2c

    Successfully tagged USERNAME/repo-test:latest

root@vbox:/home/busybox# docker run USERNAME/repo-test
Ciao! Questa è la mia prima immagine Docker.
```

Listato 6.3 Il comando docker push.

```
root@vbox:/home/busybox# docker push USERNAME/repo-test

The push refers to repository [docker.io/USERNAME/repo-test]
5b0d2d635df8: Preparing
denied: requested access to the resource is denied
```

Che cosa è andato storto? Naturalmente dobbiamo effettuare il login, altrimenti non sarà possibile effettuare il *push* sul repository; possiamo farlo tramite il seguente comando, e poi eseguire nuovamente il *push* dell'immagine.

Listato 6.4 Il comando docker login.

```
root@vbox:/home/busybox# docker login -u "USERNAME" -p "PASSWORD" docker.io
WARNING! Using --password via the CLI is insecure. Use --password-stdin.
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning.

See https://docs.docker.com/engine/reference/commandline/login/#credentials-store
```

*****ebook converter DEMO

Watermarks*****

```
Login Succeeded
```

```
root@vbox:/home/busybox# docker push USERNAME/repo-test
The push refers to repository [docker.io/ USERNAME/repo-test]
5b0d2d635df8: Mounted from library/busybox
latest: digest:
sha256:002802729c73f0f46b0dc9e4c58dacea05f4b9c4ed350e29d9fd09a63178feb6 size: 527
```

A questo punto, nel Docker Hub, il repository dovrebbe avere il tag `latest` sotto la voce `Tags`, come possiamo vedere nella Figura 6.5.

Questo vuol dire che in qualunque momento volessimo accedervi, anche da un host diverso da quello in cui abbiamo creato l'immagine, potremo effettuare il *pull* dell'immagine e potremo utilizzarla localmente.

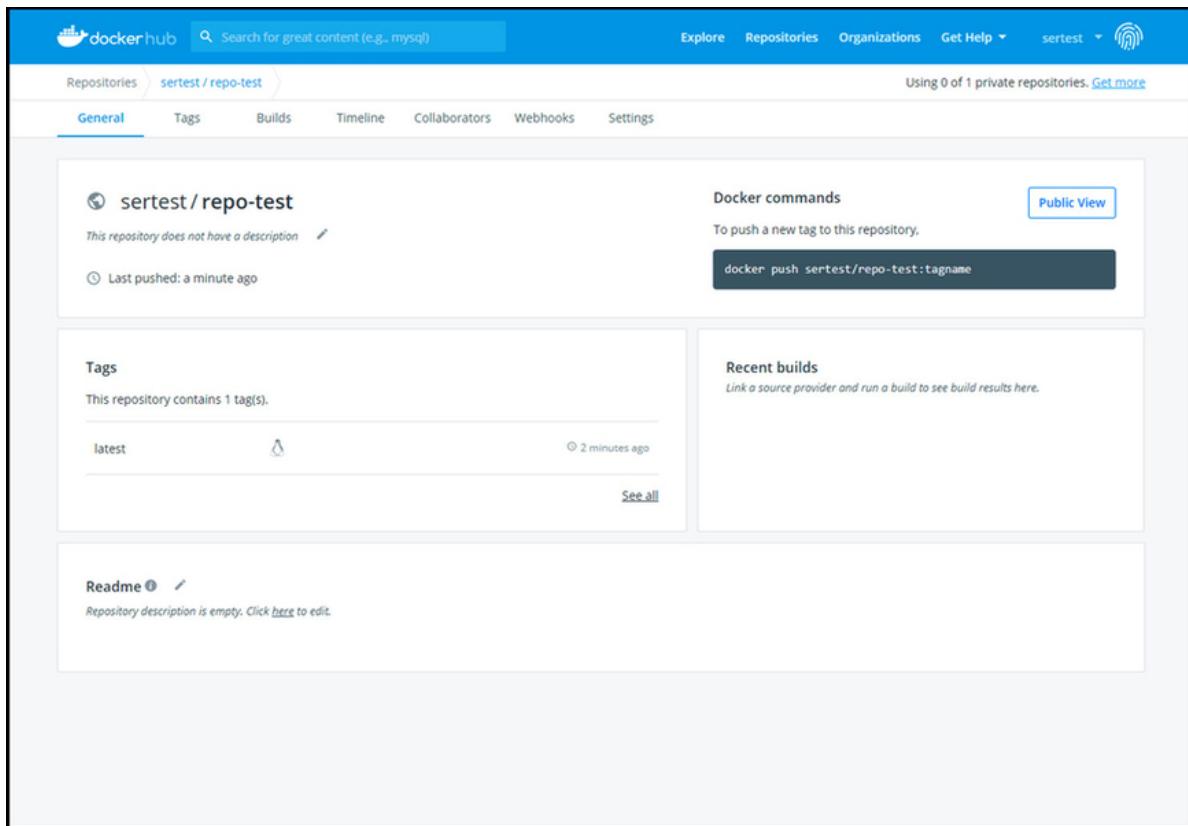


Figura 6.5 Repository aggiornato dopo l'esecuzione del push.

Namespacing

Così come Internet utilizza il *Domain Name System* (DNS) per avere un set univoco di nomi riferiti a tutti gli host presenti in rete, Docker si è proposto di creare un *namespace* per consentire di denominare in modo univoco ogni immagine presente sui suoi registri. In quest’ottica, il comando `docker run` andrebbe a eseguire lo stesso identico tipo di immagine che è stato scaricato sul sistema locale, indipendentemente dalla posizione o dal tipo di sistema su cui lo si esegue.

Per alcuni utenti Docker ciò però potrebbe rappresentare un problema: alcune installazioni di Docker sono disconnesse da Internet e dunque i requisiti di sicurezza consentono loro di cercare ed estrarre immagini solo dai registri di loro proprietà.

La discussione sul funzionamento del *namespace Docker* va avanti da diverso tempo e, a mano a mano che la tecnologia cresce, possiamo aspettarci che le modalità d’uso evolvano. Allo stato attuale, tuttavia, è necessario sapere che un sistema che esegue Docker esclusivamente dal codice del progetto Docker ufficiale ha i seguenti attributi.

- *Ricerca di un’immagine*: tramite il comando `docker search`, la ricerca viene effettuata nel *Docker Hub Registry* di default.
- *Blocco dei registri*: Docker non dispone di una funzione per bloccare un registro; pertanto, se estraiamo un’immagine senza identificare un registro, Docker cerca quell’immagine nel *Docker Hub* (se non è già presente nel sistema locale).
- *Modifica del registro predefinito*: Docker non dispone di una funzione per modificare il registro predefinito rispetto a Docker Hub, ma è necessario effettuare operazioni di *push* e *pull* specificando l’host su cui è disponibile il registro da utilizzare.
- *Conferma prima di effettuare il push*: Docker non richiede di confermare una richiesta di *push* prima che inizi a inviare un’immagine.

A parte queste specifiche, per ciò che concerne il *namespace*, quando si crea un nuovo repository è necessario tenere a mente le seguenti regole.

- È possibile scegliere di inserire il repository solo all'interno del proprio namespace Docker o in quello dell'organizzazione di cui siamo proprietari. Per esempio, avendo un account con username “*m.rossi*”, sarà possibile inserire un repository solo sotto questo nome.
- *Il nome del repository deve essere univoco*: ciò significa che il nome del repository deve essere unico per l'account utilizzato; qualcun altro potrebbe avere un repository chiamato `test`. Tuttavia, non è possibile avere nel proprio account due repository con lo stesso nome. È fondamentale scegliere un nome descrittivo, basato sul progetto. Può essere composto da 2 a 255 caratteri e può contenere solo lettere minuscole, numeri o i simboli “-” e “_”. Per esempio, se l'utente *m.rossi* ha creato un repository `test`, non può creare un secondo, ma un utente *g.verdi* può comunque creare un proprio repository `test`.
- *La descrizione può contenere fino a 100 caratteri* e viene utilizzata nei risultati della ricerca, quindi è molto importante prestare attenzione a fornire informazioni che, nel caso in cui il repository sia pubblico, possano essere d'aiuto anche ad altri utenti per capire quali sono le intenzioni e lo scopo dell'immagine.

Creare un registro privato

Abbiamo visto all'inizio del capitolo che un registro privato consente di condividere all'interno di un'organizzazione le immagini di base personalizzate, mantenendo una sorgente coerente, privata e centralizzata per ogni blocco di una certa soluzione. Un registro Docker privato offre prestazioni migliori per cluster di grandi dimensioni e con roll-out molto frequenti, oltre a funzionalità aggiuntive come l'autenticazione dell'accesso. Per migliorare la disponibilità, sarebbe sicuramente opportuno ospitare il registro su un server esterno; immaginando quindi di avere a disposizione un server nel quale appoggiare il registro, vediamo come impostarne uno privato; non verrà trattata invece la questione di sicurezza legata alla protezione del registro sul server, perché non rientra negli scopi di questo manuale; è comunque necessario sottolineare come sia necessario adottare le giuste misure in termini di

sicurezza, soprattutto se le immagini utilizzate contengono informazioni sensibili.

Un registro è solo un’immagine Docker, quindi la prima cosa da fare è quella di configurare Docker sul server (l’installazione è descritta nel Capitolo 2). Terminata l’installazione e verificatone il corretto funzionamento, è il momento dell’installazione del registro Docker. Questa parte potrebbe essere la più semplice dell’intero processo di installazione; come abbiamo detto, un registro Docker è un container specifico in esecuzione, chiamato `registry`.

Il comando da eseguire per crearne uno è il seguente.

Listato 6.5 Esecuzione di un container `registry`.

```
root@vbox:/home/busybox# docker run -d -p 5000:5000 --restart=always --name registry
registry:2
```

Abbiamo avviato un container a partire dall’immagine `registry` con tag `2`, di cui esponiamo la porta `5000` sul sistema che lo ospita, dandogli il nome di `registry`.

Per verificare che il container sia stato avviato correttamente, possiamo eseguire il comando `docker ps`.

Listato 6.6 Il comando `docker ps`.

CONTAINER	ID	IMAGE	COMMAND	PORTS	NAMES
	50b0ee97d6ab	registry:2	"/entrypoint.sh..."	0.0.0.0:5000-	

>5000/tcp registry

Ora, per verificare che il registro si comporti come ci aspettiamo, effettuiamo il *push* di un’immagine di base nel nostro nuovo registro; per semplicità, useremo l’immagine `hello-world`.

Per farlo, dobbiamo porre l’URL del registro (in questo caso, l’IP del server o il dominio utilizzato per raggiungerlo) davanti al nome dell’immagine, specificando anche la porta esposta; per esempio, supponendo che l’IP sia `192.168.0.3`, utilizzeremo come indirizzo `192.168.0.3:5000/hello-world`.

Listato 6.7 Il comando `tag`.

```
root@vbox:/home # docker tag hello-world 192.168.0.3:5000/hello-world
```

In questo esempio, abbiamo creato un tag denominato `hello-world` che fa riferimento all’immagine di base `hello-world`; eseguendo il comando `docker images`, dovremmo avere un risultato simile al seguente.

Listato 6.8 Il comando images.

```
root@vbox:/home # docker images
192.168.0.3:5000/hello-world      latest      5f7a8c027bb7b      2 days
ago          1.84kB
hello-world                         latest      5f7a8c027bb7b      2 days
ago          1.84kB
```

Sebbene siano riportate due immagini con due etichette diverse, si fa riferimento alla stessa immagine `hello-world` (basta controllare l'ID dell'immagine, ossia `5f7a8c027bb7b` nel nostro caso).

Le etichette sono in effetti piuttosto simili ai collegamenti creati tramite il comando `ln`; rimuovendo una delle etichette, l'immagine sarà comunque disponibile.

A questo punto è possibile effettuare il *push* dell'immagine sul server, tramite il comando omonimo.

Listato 6.9 Il comando push.

```
root@vbox:/home # docker push 192.168.0.3:5000/hello-world
```

Eseguendo questo comando si potrebbe incorrere nel seguente errore:

```
The push refers to repository [192.168.0.3:5000/hello-world]
```

```
Get https://192.168.0.3:5000/v2/: http: server gave HTTP response to HTTPS client
```

Docker prevede l'utilizzo di un canale protetto per impostazione predefinita, e questa è naturalmente una buona pratica in termini di sicurezza; il protocollo TLS però aggiunge un altro livello di complessità e nuovi possibili problemi, quindi al momento verrà saltato e andremo a gestire la questione in maniera differente.

La configurazione di Docker per accettare connessioni a registri non sicuri dipende dal sistema operativo in uso, ma in ogni caso è abbastanza semplice: è infatti necessario aggiornare un file `daemon.json`, che contiene le informazioni relative al daemon Docker, tra cui anche i registri su cui eseguire il *push* delle immagini.

In un sistema Linux il file si trova sotto `/etc/docker/daemon.json` e, supponendo che nel file non sia presente alcuna altra impostazione, dovrebbe avere il seguente aspetto.

Listato 6.10 File `daemon.json`.

```
{  
  "insecure-registries" : ["my_registry_address:5000"]}
```

}

È possibile creare il file, se non esiste e, per rendere effettive le modifiche, una volta terminato l'inserimento del registro sarà necessario riavviare Docker. Sempre nel caso di un sistema Linux, per far ripartire il daemon, sarà sufficiente lanciare il seguente comando.

Listato 6.11 Riavvio del daemon Docker su Linux.

```
$ sudo systemctl restart docker
```

Avendo invece a disposizione *Docker Desktop*, come nel caso di un sistema Windows o macOS, è possibile farlo usando l'interfaccia utente e le modifiche riavvieranno automaticamente il daemon. Occorre:

- fare clic sull'icona *Docker*;
- selezionare *Preferenze* nel menu principale;
- selezionare la scheda *Daemon*;
- selezionare la casella di controllo *Funzioni sperimentali*;
- nella prima casella di riepilogo, inserire l'indirizzo (URL o IP) del registro non sicuro, per esempio 192.168.0.3:5000.

Poi occorre attendere il riavvio del daemon Docker ed effettuare nuovamente il *push* dell'immagine nel registro con lo stesso comando utilizzato in precedenza. Questa volta il risultato dovrebbe essere simile al seguente:

```
The push refers to repository [192.168.0.3:5000/hello-world]
428c97da766c: Pushed
latest: digest:
sha256:1a6fd470b9ce10849be79e99529a88371dff60c60aab424c077007f6979b4812 size: 524
```

L'immagine dovrebbe ora essere archiviata in modo sicuro nel registro Docker che abbiamo impostato. Per accertarci di ciò, possiamo chiedere a questo registro remoto quali immagini contiene; fortunatamente, il registro offre anche un'API web per eseguire query sulle immagini memorizzate.

Da qualsiasi macchina che possa raggiungere il server remoto, basta digitare il seguente comando.

Listato 6.12 Il comando curl.

```
$ curl -X GET http:// 192.168.0.3:5000/v2/_catalog
```

*****ebook converter DEMO

Watermarks*****

```
{"repositories": ["hello-world"]}
```

Se il risultato è simile a questo, allora l'immagine è stata caricata con successo nel registro privato.

Che cosa abbiamo imparato

- Che cos'è Docker Hub e che cosa sono i registri.
- La differenza tra un registro pubblico e uno privato.
- Come creare un account su Docker Hub e come creare il nostro primo repository.
- Come funziona un account gratuito e uno a pagamento su Docker Hub.
- Quali sono le alternative disponibili sul mercato, come CoreOS Quay, ECR o ACR.
- Come effettuare il *push* di un'immagine sul repository creato.
- Come funziona il *namespacing* delle immagini.
- Come creare un registro privato e come eseguire il *push* di un'immagine su di esso.

Best practice

Ottimizzare il Dockerfile

L'ordine delle istruzioni in un Dockerfile è davvero importante; adottando alcune strategie di configurazione, è possibile ridurre le dimensioni dell'immagine, ridurre il numero di livelli creati e sfruttare al massimo le funzionalità di memorizzazione nella cache Docker. Le conseguenze che si avrebbero non adottando questi piccoli trucchi sono sicuramente negative: tempi di costruzione e di distribuzione più lunghi oppure immagini che non entrano nella cache e creano livelli non necessari.

Sono problemi importanti? Dipende, ma se l'intenzione è quella di distribuire soluzioni sviluppate e di farlo in modo rapido ed efficiente, il download di un aggiornamento di 5 MB che contiene solo le modifiche al codice sorgente è ovviamente più veloce del download di 400 MB di aggiornamenti. Vediamo dunque alcune regole da seguire per lavorare con un Dockerfile ottimizzato al massimo e che quindi ci permetta di operare al massimo delle prestazioni possibili.

L'ordine conta

L'importanza dell'ordine delle istruzioni in un Dockerfile è fondamentale, soprattutto nel momento in cui ogni istruzione `RUN` è in relazione con altre istruzioni, come `CMD`, `ENV`, `ARG`, `ADD` o `COPY`; è importante perché influisce sulla funzione di memorizzazione nella cache Docker e quindi sul tempo di costruzione e sulla dimensione dell'immagine.

Per esempio, se si inserisce un'istruzione `ADD` sopra ad alcune istruzioni `RUN` e cambia qualcosa nei file o nelle directory aggiunte, la cache di tutti i

livelli successivi viene automaticamente invalidata. Tuttavia, questo è un caso specifico, che non può essere generalizzato. Esistono però regole generiche che permettono di fare alcune considerazioni importanti, come:

- *collocare le istruzioni statiche nella parte iniziale del Dockerfile;* per esempio, istruzioni come, ma non solo, `EXPOSE`, `VOLUME`, `CMD`, `ENTRYPOINT` e `WORKDIR`, il cui valore non cambierà una volta impostato;
- posizionare le istruzioni dinamiche più in basso; parliamo di istruzioni come `ENV` (quando si utilizza la sostituzione variabile) `ARG` e `ADD`;
- inserire le istruzioni `RUN` che hanno dipendenze con altri comandi sempre prima delle istruzioni `ADD` o `COPY`;
- inserire le istruzioni `ADD` e `COPY` dopo le istruzioni `RUN` che servono per l'installazione delle dipendenze, ma sempre prima delle istruzioni dinamiche.

Un caso è l'esempio del Listato 7.1.

Listato 7.1 Dockerfile ottimizzato.

```
FROM ubuntu
WORKDIR /opt
EXPOSE 5000
VOLUME ["/data"]
ENTRYPOINT ["node"]
CMD ["app.js"]
ADD package.json /opt/package.json
RUN npm install --production
COPY . /opt
ARG BUILD_REVISION=master
ENV REVISION=${BUILD_REVISION}
```

`EXPOSE`, `WORKDIR`, `VOLUME` ed `ENTRYPOINT` sono istruzioni statiche e, in quanto tali, sono state poste all'inizio; l'istruzione `ADD` viene posta prima dell'istruzione `RUN`, in modo che quando verrà eseguita l'installazione di tutte le dipendenze, il file `package.json` sia pronto. L'istruzione `COPY` non è coinvolta nel comando eseguito tramite l'istruzione `RUN`, pertanto può

essere posta anche dopo di essa; le istruzioni `ARG` ed `ENV` sono poste per ultime, in quanto sono dinamiche e modificano l’ambiente che verrà avviato con il container.

Istruzione RUN

Abbiamo già visto in precedenza questa istruzione, ma vale la pena riprendere un argomento: l’istruzione `RUN`, spesso utilizzata per eseguire l’installazione di dipendenze di sistema o l’installazione di pacchetti legati a una determinata applicazione, ottiene il massimo risultato se si combinano quante più istruzioni possibili in una riga di comando. Per esempio, un Dockerfile come il seguente può essere facilmente ottimizzato.

Listato 7.2 Istruzioni RUN prima dell’ottimizzazione.

```
...
RUN apt-get -y update
RUN apt-get install -y python
...
```

In questo caso, vengono creati due layer, ciascuno dei quali esegue il rispettivo comando; se combiniamo le due istruzioni, verrà creato un unico livello nell’immagine finale, permettendoci di risparmiare tempo nella costruzione dell’immagine.

Listato 7.3 Istruzioni RUN dopo l’ottimizzazione.

```
...
RUN apt-get -y update && apt-get install -y python
...
```

Un altro esempio è il seguente, dove vi sono diverse istruzioni `RUN` non ottimizzate, che creano a ogni step un livello diverso.

Listato 7.4 Istruzioni RUN prima dell’ottimizzazione.

```
...
RUN apt-get update
RUN apt-get install -y wget sudo supervisor
RUN apt-get install -y python
```

```
RUN apt-get update  
RUN apt-get install -y openssh-server  
EXPOSE 4000 5000  
...
```

Questo esempio non è necessariamente negativo, ma non è ottimizzato: ogni volta che viene eseguito, se le istruzioni della prima esecuzione cambiano, tutte le cache dei livelli successivi vengono invalidate, portando a tempi di costruzione più lunghi e immagini più grandi. Inoltre, non combinando le istruzioni di esecuzione, vengono per l'appunto creati più livelli; per correggere l'esempio precedente, possiamo combinare i vari comandi in un'unica istruzione `RUN`, come di seguito riportato.

Listato 7.5 Istruzioni RUN dopo l'ottimizzazione.

```
...  
RUN \  
    apt-get update && \  
    apt-get install -y wget sudo supervisor python && \  
    apt-get update && \  
    apt-get install -y openssh-server  
EXPOSE 4000 5000  
...
```

I dodici fattori

La metodologia conosciuta come i *dodici fattori* è stata ideata e fortemente voluta dagli sviluppatori per il corretto funzionamento e la distribuzione di applicazioni di tipo SaaS (*Software as a Service*) o per applicazioni web che pongono particolare attenzione all'utilizzo di microservizi.

Esiste un sito che riporta queste linee guida, *12factor.net*: i dodici fattori che vengono descritti parlano della gestione delle dipendenze, di come rilasciare un'applicazione, come gestirne la configurazione o altri fattori come concorrenza e disponibilità.

Questi punti possono essere riassunti brevemente come segue, immaginando che un'applicazione ideale debba essere:

- automatizzata dal punto di vista dello sviluppo e della distribuzione;
- portatile in diversi ambienti di esecuzione;
- distribuibile su ambienti con tipologie differenti, senza che questo richieda la manutenzione del server;
- a prova di *continuous delivery*;
- scalabile in modo semplice;
- gestibile tramite *versioning* del codice.

Con l'avvento del *cloud computing*, le attività di sviluppo delle applicazioni web sono cambiate in maniera drastica e la fase di transizione non è stata di certo facile. Gli sviluppatori che prima erano a loro agio con la distribuzione su un server *bare-metal*, si sono trovati ad affrontare l'utilizzo e la coesione di più server, con la relativa distribuzione del carico.

Adam Wiggins, co-fondatore di Heroku, e il suo team hanno proposto un processo che unificasse alcune buone regole di progettazione e implementazione di una soluzione software che hanno appunto chiamato “La metodologia delle applicazioni a dodici fattori”. Poiché la metodologia è il risultato dello sviluppo di un gran numero di applicazioni, le sue linee guida garantiscono produttività e scalabilità. Il formato è principalmente influenzato dai libri di Martin Fowler *Patterns of Enterprise Application Architecture* e *Refactoring*.

La metodologia dei dodici fattori è un insieme di linee guida che fornisce le migliori pratiche e un approccio organizzato allo sviluppo di moderne applicazioni web complesse. I principi che suggerisce non sono vincolati ad alcun linguaggio di programmazione o database, in quanto questi principi sono abbastanza flessibili da poter essere utilizzati con qualsiasi linguaggio di programmazione o database.

Container effimeri

Per definizione, i container sono considerati *effimeri*; con ciò si intende che un container può essere fermato e distrutto e che uno nuovo può essere costruito a partire dalla stessa immagine Docker ed eseguito con poche attività di installazione e configurazione.

Pertanto, è bene tenere presente che quando si crea un container a partire da un'immagine Docker, ogni modifica apportata, come per esempio l'aggiunta di un file o l'installazione di una libreria, andrà persa una volta arrestato o eliminato il container. In altre parole, quando un container viene eliminato, tutti i dati scritti in tale container che non sono memorizzati in un volume di dati vengono eliminati insieme al container.

L'ideale è dunque quello di provare a progettare container facili da creare e distruggere; se i container dipendono troppo da fattori esterni e dalle configurazioni, sono più difficili da mantenere. La progettazione di container senza stato può aiutare a semplificare il sistema; se questi avessero bisogno di mantenere delle informazioni, è bene progettare una soluzione in cui l'istanza mantenga la sua proprietà effimera e che tutto ciò che concerne lo stato persistente dell'istanza sia memorizzata tramite un volume che ne conservi le informazioni.

Uso di `.dockeignore`

Se abbiamo a che fare con una build complessa, che attraversa ricorsivamente più directory, tutti i file e le directory vengono comunque elaborati dal daemon Docker. Questo può comportare la creazione di immagini più grandi e tempi di costruzione più lenti, soprattutto se nel processo di avvio del container abbiamo istruzioni che aggiungono massivamente file a partire da directory dove sono presenti file non indispensabili per l'istanza stessa. Ma esiste un file che è possibile utilizzare per escludere file e cartelle non necessari e che complicano il processo di compilazione; il file `.dockeignore` funziona come l'omonimo utilizzato per Git, ossia `.gitignore`: ignora una serie di file o directory specificati al suo interno. Perché è importante?

Ci sono molti vantaggi nell'uso di `.dockeignore`: può aiutare a ridurre le dimensioni dell'immagine Docker, accelerare la creazione del container ed evitare l'esposizione indesiderata di dati sensibili (a breve vedremo che cosa si intende). Prima però, per capire perché `.dockeignore` abbia un'efficacia così importante, è necessario capire il contesto di compilazione.

Il comando di costruzione `docker build` crea una nuova immagine Docker; è possibile passare al comando `build` un parametro che specifica

il contesto di build. Nella maggior parte dei casi di solito si passa la directory corrente, come nell'esempio sotto riportato.

Listato 7.6 Il comando docker build.

```
root@vbox:/home # docker build . -t myapp-image:1.0.
```

Innanzitutto, dobbiamo ricordare che Docker è un'applicazione client-server, costituita da client Docker e server Docker (noto anche come daemon Docker). La riga di comando del client Docker comunica con il server Docker e gli chiede di fare qualcosa; una di queste attività è la creazione di container, avendo a disposizione un'immagine di partenza oppure un Dockerfile. Il server Docker, come abbiamo visto in precedenza, può essere eseguito sulla stessa macchina dove si trova il client o in una macchina virtuale, che può anche essere locale, remota o nel cloud.

Per creare una nuova immagine Docker, il server Docker deve accedere ai file dai quali si desidera creare l'immagine Docker; quindi, è necessario in qualche modo inviare questi file al server Docker (perché, ricordiamo, il server Docker può essere anche su un sistema remoto). Questi file sono il contesto di compilazione Docker; il client comprime tutti i file di contesto di build in un archivio `.tar` e carica questo archivio sul server Docker. Per impostazione predefinita, il client prenderà tutti i file (e le cartelle) nella directory di lavoro corrente e li utilizzerà come contesto di compilazione.

È anche possibile utilizzare come contesto Docker un archivio `.tar` o un repository Git. Nel caso di un repository Git, il client lo clonerà e utilizzerà anche i moduli, che verranno compressi in una cartella temporanea, creando un archivio di contesto di build.

Immaginando di effettuare la build a partire da un Dockerfile qualsiasi, le prime righe di output dovrebbero essere simili alle seguenti.

```
root@vbox:/home# docker build . -t myapp-image:1.0.
```

```
Sending build context to Docker daemon 45.3 MB
```

```
Step 1: FROM ...
```

Questo ci dice che ogni volta che si esegue il comando `docker build` il client Docker crea un nuovo archivio del contesto di build e lo invia al server Docker. Significa dunque che quest'operazione ha un "costo fisso": il tempo necessario per creare un archivio, per la memorizzazione, per il traffico di rete e per la latenza.

Il file `.dockerignore` è lo strumento che può aiutarci a definire il contesto di compilazione di cui si ha veramente bisogno. Utilizzando questo file, è possibile specificare le regole per ignorare dei file oppure le eccezioni per queste regole che valgono per specifici file o cartelle, che non saranno inclusi nel contesto di compilazione e quindi non verranno impacchettati in un archivio e caricati sul server Docker.

Il mondo dello sviluppo software si sta spostando sempre più verso concetti come *continuous delivery*, infrastrutture elastiche e architetture a microservizi; spesso questi sistemi sono composti da più componenti, ognuno dei quali è in esecuzione all'interno di un container. Potrebbero esserci decine o centinaia di servizi e persino più istanze di un certo servizio; queste istanze possono essere create e distribuite indipendentemente l'una dall'altra e ciò può essere fatto per ogni singolo *commit* del codice. Inoltre, un'infrastruttura elastica consente di aggiungere al sistema o rimuovere dal sistema dei nodi di calcolo e i suoi servizi possono essere spostati da un nodo all'altro, per supportare requisiti di scalabilità o disponibilità. Ciò significa che le immagini Docker verranno create e trasferite frequentemente.

Quando si applicano questi concetti, le dimensioni e il tempo di costruzione dell'immagine sono molto importanti. È chiaro che sia molto più veloce distribuire su cento server un'immagine Docker da 5 MB rispetto a un'immagine da 700 MB, ma queste regole aiutano anche nello sviluppo locale.

Non controllando il contesto di compilazione, si può arrivare a un'esposizione non intenzionale del codice, della cronologia di commit oppure di informazioni sensibili, come le chiavi e le credenziali di accesso.

Se si copiano file nell'immagine Docker con le istruzioni `ADD` oppure `COPY` in maniera massiva, è possibile includere involontariamente i file sorgente, l'intera cronologia git (contenuta nella cartella `.git`), i file segreti (come `.aws`, `.env`, oppure le chiavi private), la cache e altri file non solo nel contesto di compilazione Docker, ma anche nell'immagine finale.

Un modello comune consiste nell'iniettare in un'immagine l'intera base del codice di un'applicazione, usando un'istruzione come la seguente.

Listato 7.7 Istruzione COPY.

```
COPY . /usr/src/app
```

In questo caso, stiamo copiando l'intero contesto di compilazione nell'immagine, compresi eventuali file con dati di accesso, come quelli appena menzionati. È anche importante capire che ogni comando Dockerfile genera un nuovo livello; quindi, se uno qualsiasi dei file inclusi cambia, questa modifica invaliderà la cache di compilazione per l'istruzione `COPY` e sulla build successiva verrà generato un nuovo layer immagine, rendendo ovviamente la build molto più lenta.

Se la directory di lavoro contiene file (come registri, risultati dei test, cronologia Git, file di cache temporanei e simili) che vengono aggiornati frequentemente, questo livello verrà rigenerato per ogni esecuzione della compilazione dell'immagine.

Ora che abbiamo chiare le motivazioni per cui questo file è importante, passiamo alla sintassi; come abbiamo detto, il file `.dockerignore` è simile al file `.gitignore`; infatti, analogamente a quest'ultimo, è possibile specificare un modello per file e cartelle che deve essere ignorato dal client Docker durante la generazione del contesto di compilazione.

Per escludere, per esempio, le cartelle di cache e della cronologia Git, possiamo scrivere all'interno del file quanto segue.

Listato 7.8 Esempio di `.dockerignore`.

```
.git  
.cache
```

Un altro caso d'uso è l'esclusione di tutti i file compilati presenti nel contesto di un'applicazione, come tutti i file che hanno l'estensione

```
.class.
```

Listato 7.9 Esempio di `.dockerignore` per file compilati.

```
**/*.class
```

Oppure viene spesso utilizzato per evitare di copiare le dipendenze di un'applicazione, che possono invece essere scaricate direttamente dal container.

Listato 7.10 Esempio di `.dockerignore` per file compilati.

```
*/node_modules
```

Usare `.dockerignore` è fondamentale in ogni progetto, di piccola o grande dimensione, in cui si costruiscono immagini Docker; aiuta infatti a rendere le immagini compatte, veloci e, soprattutto, sicure, per non parlare dell'uso della cache durante lo sviluppo locale.

Evitare l'uso del tag latest

In questo caso, il suggerimento è semplice: se si utilizza l'istruzione `FROM [immagine]:latest`, è possibile riscontrare problemi ogni volta che l'immagine cambia; immaginiamo di gestire una soluzione a lungo termine per un'azienda dove la manutenzione verrà effettuata semestralmente; in caso di aggiornamenti di pacchetti importanti del sistema, il riavvio continuo delle istanze Docker e i relativi problemi legati alle librerie che nel frattempo vengono aggiornate, può diventare un problema difficile da rintracciare.

L'uso di tag specifici può garantire la conoscenza dell'esatta immagine utilizzata dal registro Docker; in questo modo, l'aggiornamento del sistema utilizzato dal container o anche dell'immagine di base stessa può essere gestito in maniera ottimale, andando a prevenire eventuali complicazioni legate a fattori come l'aggiornamento del registro Docker dell'immagine.

Scegliere l'immagine di partenza

Le immagini ufficiali disponibili su Docker Hub sono già ottimizzate; invece di crearne una, è preferibile utilizzare le immagini già disponibili. Per esempio, se abbiamo bisogno di un'immagine Redis, possiamo crearla a partire da un'immagine Ubuntu o scaricare direttamente quella ufficiale presente sul registro.

Il problema è che quando creiamo un'immagine Docker per un'applicazione e vogliamo usare un'immagine esistente, ci troviamo di fronte a molte possibili opzioni: ci sono immagini con un sistema operativo come Ubuntu e CentOS e ci sono molte diverse varianti dell'immagine base di Python.

Quale usare? Qual è la combinazione migliore? Le scelte sono diverse e potrebbe non essere così ovvio quale sia la migliore per una

determinata situazione. Andiamo dunque ad analizzare i principali criteri su cui basare la decisione e vediamo alcuni casi pratici, esaminando i vantaggi e gli svantaggi nell'uso delle immagini di base più comuni.

Innanzitutto, esistono numerosi criteri per la scelta di un'immagine di base, sebbene ogni progetto rappresenti un caso particolare che può enfatizzare, aggiungere o rimuovere alcuni di questi.

- *Stabilità*: la build deve avere lo stesso set base di librerie, la stessa struttura di directory e la stessa infrastruttura di una build replicata in tempo breve-medio.
- *Aggiornamenti di sicurezza*: immagine di base con supporto continuo, in modo da ottenere tempestivamente gli aggiornamenti di sicurezza per il sistema operativo di base.
- *Dipendenze aggiornate*: a meno che non stiamo creando un'applicazione molto semplice, probabilmente sarà necessarie che sul sistema operativo siano installate delle librerie e delle applicazioni (per esempio un compilatore per un qualche linguaggio di programmazione).
- *Dipendenze estese*: per alcune applicazioni potrebbero essere necessarie dipendenze meno diffuse: un'immagine di base che abbia l'accesso a un gran numero di librerie.
- *Immagine piccola*: di cui sia facile il trasporto e la build in diversi ambienti di sviluppo.
- *Minimizzare i possibili problemi*: ciò che non è incluso non può rompersi: questa è una delle regole più importanti per un'immagine di base. È necessario assicurarsi che sia incluso solo il software che è effettivamente necessario, e di conseguenza, è importante sapere davvero quale software è incluso e come funziona.
- *Strumenti di base*: è importante valutare ciò che già viene messo a disposizione dall'immagine, e ciò che invece richiede attività supplementari, secondo le necessità della soluzione.

Esistono tre principali sistemi operativi che soddisfano all'incirca tutti i criteri di cui sopra (date e versioni di rilascio sono aggiornate al momento della scrittura di questo manuale).

- *Ubuntu 20.04* (l'immagine `ubuntu:20.04`) è stato rilasciato a fine aprile 2020 e, poiché si tratta di una versione di supporto a lungo termine, riceverà aggiornamenti di sicurezza fino al 2025.
- *CentOS 8* (`centos:8`) è stato rilasciato nel 2019 e avrà aggiornamenti completi fino al 2024 e aggiornamenti di manutenzione fino al 2029.
- *Debian 10* (“Buster”) è stato rilasciato il 6 luglio 2019 e sarà supportato fino al 2024.

Tutte queste immagini presentano diversi vantaggi comuni.

- *Debian GNU/Linux* è uno dei sistemi operativi più longevi basati sul kernel Linux ed è in circolazione dai primi anni Novanta. Debian è una distribuzione ricca di funzionalità, grazie alle decine di migliaia di pacchetti cui può attingere, e supporta molte architetture ed esigenze diverse. Debian - e la sua distribuzione derivata *Ubuntu* - sono sicuramente tra i sistemi operativi più utilizzati, e vengono installati sui server in tutte le aziende del mondo.
- *Supportano la maggior parte dei linguaggi di programmazione*, i cui compilatori possono essere facilmente installati tramite l'istruzione `RUN`.
- *Sono costantemente aggiornate e manutenute* dalle community ufficiali, che si occupano di rilasciare una nuova versione dell'immagine ogniqualvolta esce un aggiornamento del sistema operativo.
- *Alcune di queste immagini sono disponibili in versioni minimizzate*, come nel caso di `minideb`: questo permette di godere dei vantaggi del sistema operativo con il minimo delle funzionalità necessarie ad avviarlo.
- *Hanno cicli di vita molto lunghi*, il che vuol dire che il supporto è garantito per tempi medio-lunghi e che la manutenzione alle singole immagini può essere facilmente gestita nel tempo.

A questo elenco andrebbe aggiunta un'ulteriore immagine, molto utilizzata per chi si avvicina a Docker per la prima volta, e che tuttavia viene sconsigliata per applicazioni o soluzioni che devono essere portate

in ambiente di produzione: *Alpine* descrive la propria immagine come “piccola, sempre, sicura”, ma la realtà dei fatti spesso è ben diversa.

Quando si sceglie un’immagine di base per un’applicazione Docker, spesso si consiglia di utilizzare Alpine Linux. L’uso di Alpine dovrebbe ridurre la dimensione delle immagini e quindi accelerare la build.

Immaginiamo dunque di dover installare `gcc` nella nostra immagine e paragoniamola con Ubuntu 18.04 in termini di tempo di build e dimensioni dell’immagine.

Estraendo entrambe le immagini e verificandone le dimensioni, possiamo vedere a occhio che sono molto diverse.

Alpine, Ubuntu e gli altri

Listato 7.11 Il comando docker build.

```
root@vbox:/home# docker images ls

REPOSITORY      TAG          IMAGE ID       SIZE
Ubuntu          18.04        ccc6e87d482b   64.2MB
Alpine          latest        e7d92cdc71fe   5.59MB
```

Successivamente, installando `gcc` in entrambe le immagini e comparando nuovamente le dimensioni, abbiamo il seguente risultato.

Listato 7.12 Il comando docker build dopo l’installazione.

```
root@vbox:/home# docker images ls

REPOSITORY      TAG          IMAGE ID       SIZE
ubuntu          18.04        ccc6e87d482b   150MB
alpine          latest        e7d92cdc71fe   105MB
```

Come promesso, le immagini Alpine effettuano una build più velocemente e sono più piccole: 15 secondi anziché 30 secondi e l’immagine è 105 MB anziché 150 MB. È un risultato piuttosto buono!

Provando però a ripetere l’esperimento con un compilatore come quello Python, ci troviamo a dover affrontare diversi problemi, causati dall’installazione delle librerie più comuni, spesso per via dei file binari; Alpine, al contrario di altre immagini di base, scarica il codice sorgente delle librerie, perché i file binari standard non funzionano.

Perché? La maggior parte delle distribuzioni Linux usa la versione GNU (`glibc`) della libreria C standard, che è richiesta praticamente da ogni programma C, incluso Python, ma Alpine Linux usa `musl`, ossia una versione minimizzata ma più lenta di `glibc`, che quindi disabilita l'uso dei binari.

In altre parole, dal momento che la maggior parte dei pacchetti Python include librerie basate su PyPI, accelerando notevolmente i tempi di installazione, se si utilizza Alpine Linux è necessario compilare tutto il codice C in ogni pacchetto Python utilizzato.

Analizziamo quindi i pro e i contro delle diverse immagini citate finora.

Alpine: i pro

- *Immagini molto piccole*: la community presta molta attenzione alla riduzione al minimo delle dimensioni delle immagini.
- *Funzionalità minime*: sono contenuti solo i pacchetti assolutamente necessari.
- *Sistema di inizializzazione leggero*: come Gentoo, Alpine utilizza OpenRC, un'alternativa leggera a `systemd`, che ne velocizza la build.
- *Performance musl*: in alcuni casi, `musl libc` può essere più performante di `glibc`.

Alpine: i contro

- *Documentazione piuttosto scadente*.
- *Piccolo team*: attualmente il team di sviluppatori che ne mantiene l'immagine è davvero piccolo e non sempre i bug vengono risolti in modo tempestivo.
- *Possibili incompatibilità*: `musl` può causare problemi con alcuni plugin basati su C e potrebbero essere necessarie installazioni aggiuntive (vedi esempio precedente).

Debian: i pro

- *Immagini piccole*: la dimensione delle immagini Debian ridotte (come nel caso di `minideb`) è quasi alla pari di Alpine (per esempio `minideb` con installato Python è solo 7 MB più grande).
- *Varietà di pacchetti*: non c'è quasi nessun software per Linux che non sia disponibile per Debian.
- *Ben collaudato*: grazie alla sua popolarità, Debian è ampiamente utilizzato e si riscontrano maggiori probabilità di risoluzione dei problemi segnalati.
- *Documentazione completa*; la community ha prodotto una grande quantità di documentazioni e tutorial aggiuntivi.
- *Sicurezza*: ancora una volta, grazie alla sua più ampia community, Debian ottiene più attenzione ed è più probabile che vengano scoperte vulnerabilità o possibili problemi di sicurezza.

Debian: i contro

- *Vulnerabilità maggiore*: dal momento che presenta diversi pacchetti di base installati anche nelle versioni come `minideb`, il quale è composto da circa 35 pacchetti (come `bash`, `grep`, `hostname`, `mount` e così via), ci sono possibilità maggiori di un attacco alla sicurezza di sistema (non che sia così semplice!).

Ubuntu: i pro

- *Basato su Debian*, utilizza sempre `glibc`, che permette la facile installazione dei compilatori più comuni e dei servizi di sistema.
- *I pacchetti a disposizione sono quasi gli stessi di quelli a disposizione dei sistemi Debian*.

- *Presenta una community molto estesa*, per cui è difficile che non vengano rilasciati aggiornamenti per eventuali bug o patch di sistema.

Ubuntu: i contro

- *Con l'installazione degli aggiornamenti o delle librerie, la dimensione del container aumenta abbastanza velocemente.*
- *Aggiornamenti frequenti*, il che rende il sistema più vulnerabile a possibili attacchi.

CentOS: i pro

- *Basato su RHEL (Red Hat Enterprise Linux)*, garantisce la compatibilità con librerie o programmi dello stesso sistema operativo.
- Dispone di aggiornamenti dei pacchetti meno frequenti rispetto a sistemi come Ubuntu, motivo per cui spesso è considerato più stabile.
- *Spesso utilizzato per soluzioni di hosting*, proprio per la stabilità e la sicurezza garantite.

CentOS: i contro

- *Documentazione non sempre aggiornata* e comunque in rapporto minore ad altre distribuzioni.
- *Con l'installazione degli aggiornamenti o delle librerie, la dimensione del container aumenta abbastanza velocemente.*
- *È una delle immagini di base più grandi* rispetto a quelle citate prima.

Sicuramente la conclusione che va tratta in questo caso è che non ci sia un'immagine di base adatta a tutti gli usi: è sempre bene considerare i casi d'uso specifici in ogni aspetto, valutando i criteri elencati e facendo anche uso dell'esperienza maturata con le diverse distribuzioni. Tuttavia vi sono alcuni punti fermi.

- *Alpine rappresenta una scelta valida per un'immagine di base*, anche se ci possono essere opinioni contrastanti tra gli esperti sul fatto che Alpine sia una buona scelta in termini di sicurezza e stabilità. Potrebbero esserci alcuni problemi di compatibilità o bug relativi alla rete, ma in generale Alpine funziona molto bene. Per alcuni software, non esistono pacchetti già pronti, quindi è comunque necessario del lavoro supplementare, di cui va dosato il costo.
- *Debian è una buona scelta anche per un'immagine di base*, dal momento che il risultato è paragonabile a quello ottenuto con Alpine; dispone di diverse immagini di base compatte, che contengono solo un minimo di pacchetti ben mantenuti e di cui è garantito il supporto per diversi anni.

Gli strumenti giusti (automazione della build, distribuzione automatica) e la corretta configurazione (privilegi minimi, profili di sicurezza, gestione delle policy di rete) risultano importanti per scegliere un'immagine di base solida.

Sviluppare con Docker

Se le considerazioni fatte finora valevano soprattutto per la costruzione di un Dockerfile o sulla scelta di un'immagine di base, ora vale la pena analizzare il contesto di sviluppo di un'applicazione con Docker: quali siano le buone strategie da adottare per ottenere il massimo dei risultati.

Lo sviluppo di applicazioni in container può essere considerato come parte di una più ampia transizione verso un modello di sviluppo software DevOps; i container offrono infatti agli sviluppatori un maggiore livello di controllo degli elementi che un'applicazione deve eseguire senza dover pensare troppo all'infrastruttura del data center sottostante. Allo stesso

modo, gli sviluppatori possono essere più certi che le loro applicazioni verranno eseguite a mano a mano che passano dallo sviluppo al collaudo e infine alla produzione.

Questo paragrafo descrive quello che è necessario sapere quando vogliamo sviluppare applicazioni a container; innanzitutto, espone alcuni strumenti di sviluppo disponibili, da predisporre prima di iniziare a sviluppare applicazioni per container. Successivamente, il capitolo esamina le raccomandazioni per uno sviluppo efficiente dei container, nonché le tecniche per rendere i container più facili da gestire e mantenere.

Prima di iniziare a costruire i propri container Docker, è necessario considerare gli strumenti e l'ambiente di lavoro da installare. Il progetto Docker stesso offre strumenti di sviluppo container che potrebbero tornare utili. Alcuni di questi offrono la possibilità di creare e gestire container Docker utilizzando un *front-end grafico* anziché utilizzare gli strumenti a riga di comando.

Docker Toolbox

Disponibile sul sito www.docker.com/toolbox, fornisce un programma di installazione che consente di impostare Docker su sistemi desktop o laptop. Toolbox è disponibile per Mac o Windows; utilizzando VirtualBox per creare l'ambiente Linux, Toolbox fornisce i seguenti strumenti per l'utilizzo di Docker:

- *Docker Machine*, per l'esecuzione di comandi `docker-machine`;
- *Docker Engine*, per l'esecuzione di comandi `docker`;
- *Docker Compose*, per l'esecuzione di comandi `docker-compose`;
- *Kitematic*, l'interfaccia grafica di Docker;
- una shell preconfigurata per usare la riga di comando Docker;
- *Oracle VirtualBox*.

Dopo aver scaricato l'eseguibile dal sito ufficiale, è sufficiente installarlo seguendo le indicazioni fornite dal *wizard* e al termine dell'installazione, verificarne il corretto funzionamento verificando che

applicazioni come VirtualBox o Kitematic siano installate sul proprio sistema.

Kitematic

Docker Kitematic (www.docker.com/docker-kitematic): Kitematic fornisce un’interfaccia grafica che permette di utilizzare Docker. Utilizzando Kitematic, è possibile creare ed eseguire container senza dover ricorrere al terminale.

Kitematic è un progetto open source creato per semplificare l’utilizzo di Docker su un Mac o un PC Windows. Automatizza il processo di installazione e configurazione di Docker e fornisce un’interfaccia utente grafica intuitiva per l’esecuzione di container Docker. Kitematic inoltre si integra con Docker Machine per eseguire il provisioning di una VirtualBox VM e installare Docker Engine localmente sul computer.

Una volta installata, l’interfaccia grafica di Kitematic (Figura 7.1) si avvia e, dalla schermata principale, visualizza le immagini che è possibile eseguire immediatamente. È possibile cercare qualsiasi immagine pubblica su Docker Hub da Kitematic semplicemente dalla barra di ricerca. Puoi utilizzare l’interfaccia grafica per creare, eseguire e gestire i tuoi container semplicemente facendo clic sui pulsanti. Kitematic consente di alternare tra la CLI Docker e l’interfaccia grafica, e automatizza anche alcune funzionalità avanzate, come la gestione delle porte e la configurazione dei volumi. È possibile utilizzare Kitematic per modificare le variabili d’ambiente, i log di flusso e gestire il termine con un singolo clic nel container, tutto tramite l’interfaccia grafica.

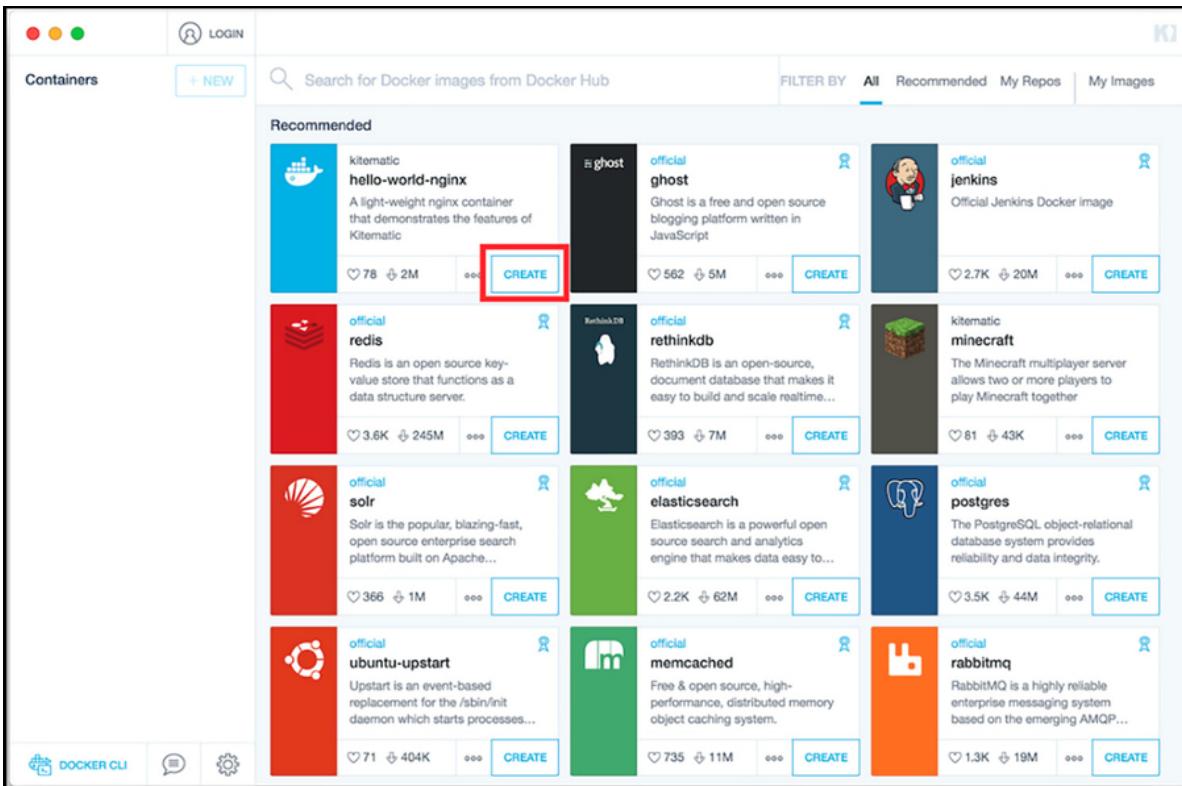


Figura 7.1 La schermata principale di Kitematic.

Kitematic è integrato con Docker Hub, che consente di cercare qualsiasi immagine direttamente tramite il registro ufficiale, e anche di estrarre e distribuire le nostre app. Attualmente include alcune funzionalità come la mappatura automatica delle porte, la modifica visiva delle variabili d’ambiente, la configurazione dei volumi, la semplificazione dei registri e molto altro.

Vediamo dunque come installarlo e come costruire un server Nginx su Windows. Prima di tutto, dovremo scaricare l’ultima versione di Kitematic disponibile per il sistema operativo Windows, tramite il sito ufficiale oppure dal repository GitHub, per esempio

<https://github.com/kitematic/kitematic/releases>. Qui, scarichiamo il file .exe e, al termine del download, potremo procedere con l’installazione.

Il programma di installazione inizierà ora a scaricare e installare le dipendenze necessarie per eseguire Kitematic. Se nel sistema hai già installato Virtualbox, questo verrà aggiornato all’ultima versione. Il programma di installazione dovrebbe terminare in pochi minuti, ma tutto dipende dalla velocità di Internet e del sistema. Dopo che le dipendenze

richieste da Docker e Virtual box saranno installate e in esecuzione, verrà richiesto di accedere a Docker Hub. Se non disponiamo di un account o non vogliamo effettuare l'accesso ora, possiamo fare clic su *Skip for now* per continuare. Al termine, verrà caricata l'interfaccia principale dell'applicazione Kitematic, come possiamo vedere nella Figura 7.2.

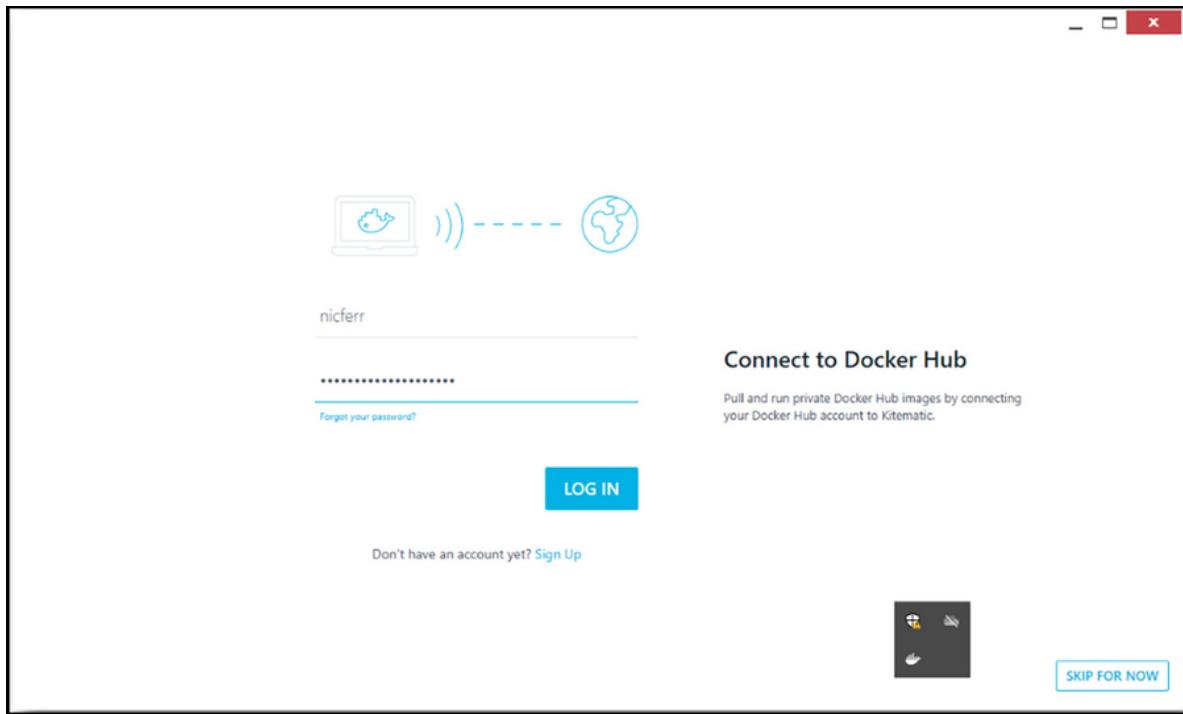


Figura 7.2 La connessione a Docker Hub.

Possiamo cercare le immagini Docker come avviene solitamente su Docker Hub, quindi per parola chiave; ora, dal momento che Kitematic è stato installato con successo, passiamo a installare l'immagine `nginx`. Per eseguire un container, possiamo semplicemente cercare l'immagine nell'area di ricerca e poi fare clic su *New* per avviare il container. Una volta che il download dell'immagine è stato completato, verrà avviato. Possiamo vedere l'esecuzione dei comandi lanciati da Kitematic per distribuire quel container tramite la finestra dei log, e anche l'anteprima della pagina web direttamente dall'interfaccia di Kitematic (Figura 7.3).

Se vogliamo passare all'interfaccia a riga di comando e gestire Docker tramite terminale, c'è un pulsante chiamato *CLI Docker* che aprirà una Powershell dove possiamo eseguire i comandi `docker`. Se invece vogliamo configurare il container e cambiare il nome del container, assegnare variabili d'ambiente, assegnare porte, configurare l'archiviazione e altre

funzionalità avanzate, possiamo farlo dalla scheda *Settings*, in alto a destra.

Docker Machine

Questo strumento fornisce un modo per eseguire il *provisioning* di Docker sul proprio computer, su un *cloud provider* o sul proprio *data center*. Utilizzando Docker Machine, possiamo eseguire il provisioning dei sistemi host e distribuire i diversi Docker Engine su tali sistemi.

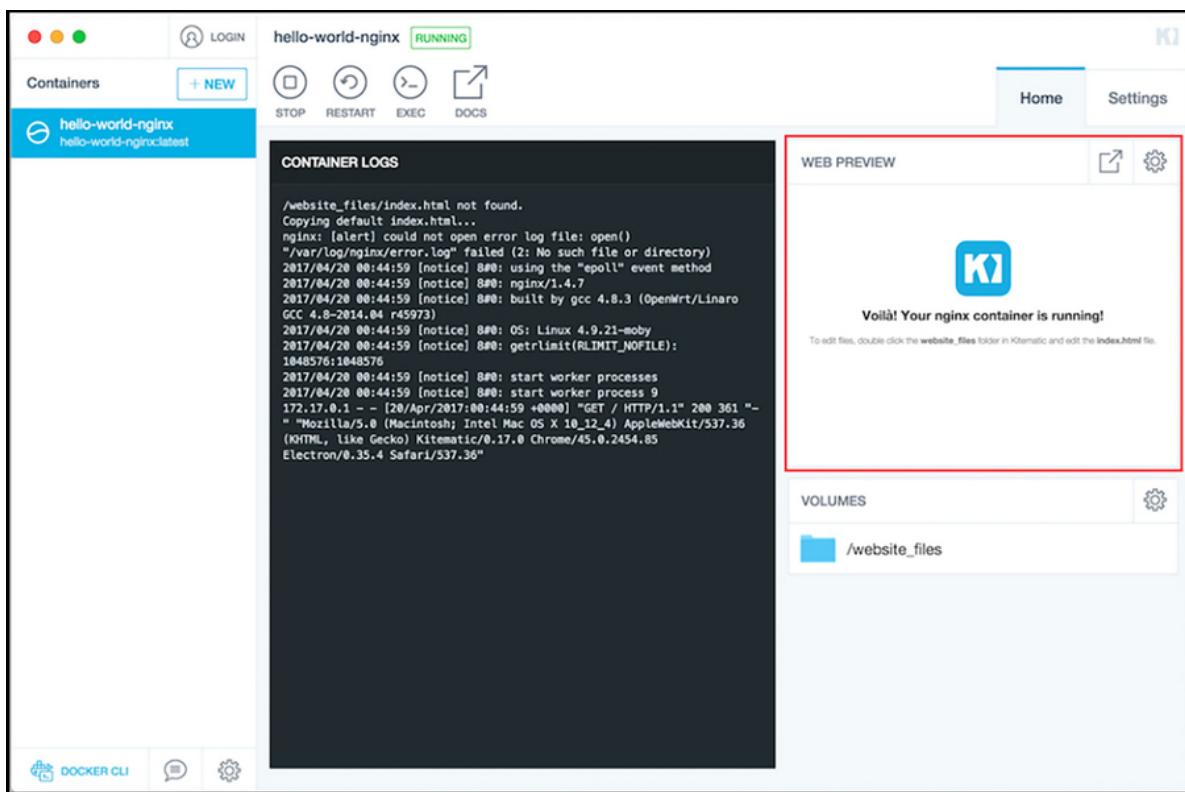


Figura 7.3 Build di un web server Nginx.

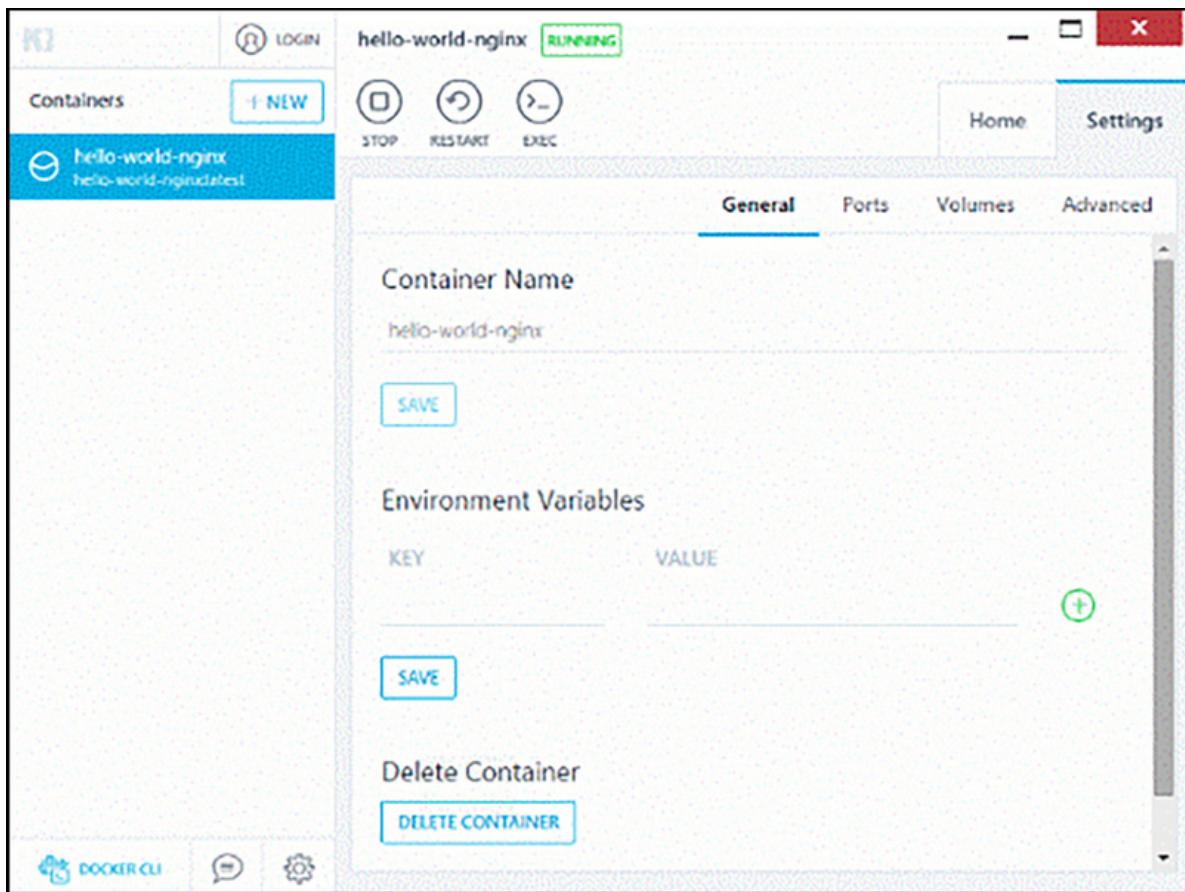


Figura 7.4 Impostazioni del container.

Docker Machine viene fornito con un numero di driver pronti all’uso. Ogni driver integra Docker Machine con una diversa tecnologia di macchina virtuale o provider di elaborazione virtuale basato su cloud. Ogni piattaforma cloud ha i suoi vantaggi e svantaggi. Non vi è alcuna differenza tra un host locale e remoto dal punto di vista di un client Docker.

Quando si dispone di un’applicazione a container, è importante essere in grado di distribuirla facilmente su ambiente cloud, non solo eseguendola localmente utilizzando Docker Desktop o tramite macchina virtuale. Lo strumento per poter creare facilmente una macchina virtuale (VM) remota e gestire tali container è proprio Docker Machine.

In altre parole, Docker Machine consente di controllare il Docker Engine di una macchina virtuale creata; permette anche di aggiornare il sistema Docker, riavviare la macchina virtuale, visualizzarne lo stato e così via.

Il motivo principale per cui si dovrebbe utilizzare Docker Machine è quando si desidera creare un ambiente di distribuzione per un'applicazione e si vogliono gestire tutti i microservizi in esecuzione su di esso. Per esempio, è possibile avere facilmente un ambiente di sviluppo, di collaudo e di produzione accessibile dalla propria macchina e aggiornarli di conseguenza in maniera altrettanto semplice.

Per la creazione di ambienti di sviluppo di container più grandi, è possibile prendere in considerazione alcuni di questi strumenti di Docker.

- *Docker Compose* (www.docker.com/docker-compose): fornito dal progetto Docker per aiutare gli sviluppatori a organizzare i container in applicazioni più grandi.
- *Docker Swarm* (<https://docs.docker.com/swarm>): fornisce strumenti per il ridimensionamento di Docker per eseguire più container e più host.

Questi strumenti possono aiutarci a creare velocemente, come vedremo nel prossimo capitolo, un ambiente per far funzionare Docker, ma estendendone l'uso, tramite la definizione gestione di più container o servizi.

Creazione di host

Questo paragrafo introduce i comandi per poter utilizzare Docker Machine. Poiché molti di questi strumenti sono simili nella loro forma e nelle loro funzioni, questo paragrafo introdurrà un piccolo insieme di esempi con i relativi comandi, per far capire come utilizzare all'atto pratico questo strumento.

La prima e più importante cosa da sapere con Docker Machine è *creare degli host*. I successivi tre comandi creeranno tre diversi host utilizzando come driver VirtualBox. Ogni comando creerà sul computer una nuova macchina virtuale, che potrà ospitare diverse tipologie di servizi.

Listato 7.13 Il comando docker-machine create.

```
root@vbox:/home# docker-machine create --driver virtualbox host1
root@vbox:/home# docker-machine create --driver virtualbox host2
root@vbox:/home# docker-machine create --driver virtualbox host3
```

Dopo aver eseguito questi tre comandi (ci potrebbe volere qualche minuto), avremo a disposizione tre host Docker gestiti dalla Docker Machine. Docker Machine tiene traccia di questi sistemi con una serie di file nella directory home (che si trova in `~/.docker/machine/`). Qui vengono descritti gli host creati, i certificati utilizzati per stabilire comunicazioni sicure con gli host e l'immagine del disco utilizzata, nel caso di host basati su VirtualBox.

Ispezionare gli host

Il comando `docker-machine` può essere utilizzato anche per elencare, ispezionare e aggiornare i sistemi; utilizzando il sottocomando `ls` è possibile ottenere un elenco di macchine presenti nel sistema e in gestione a Docker Machine.

Listato 7.14 Il comando `docker-machine ls`.

```
root@vbox:/home# docker-machine ls
```

NAME	ACTIVE	DRIVER	STATE	URL
host1		virtualbox	Running	tcp://192.168.99.100:2376
host2		virtualbox	Running	tcp://192.168.99.101:2376
host3		virtualbox	Running	tcp://192.168.99.102:2376

Questo comando elencherà ogni macchina e il relativo nome, il driver con cui è stata creata, il suo stato e l'URL da cui è possibile raggiungere il daemon Docker. Se utilizziamo Docker Machine per eseguire Docker localmente, avremo un'altra voce in elenco e quella voce sarà probabilmente contrassegnata come “attiva”. La macchina attiva è infatti quella con cui l'ambiente è attualmente configurato per comunicare. Tutti i comandi emessi con `docker` o `docker-compose`, come vedremo a breve, si collegheranno con il daemon sulla macchina attiva.

Per saperne di più su una specifica macchina o per cercare una parte della sua configurazione, possiamo usare il sottocomando `inspect` nel modo seguente.

Listato 7.15 Il comando `docker-machine inspect`.

```
root@vbox:/home# docker-machine inspect host1
```

```
{  
  "DriverName": "virtualbox",
```

```

    "Driver": {
        "MachineName": "XXX",
        "SSHPort": 55834,
        "Memory": 1024,
        "DiskSize": 20000,
        "Boot2DockerURL": "",
        "IPAddress": "192.168.99.100"
    },
    ...
}

```

Il sottocomando `inspect` per `docker-machine` è molto simile al comando analogo per `docker`.

Gestire gli host

Docker Machine consente inoltre di costruire con relativa facilità una flotta di sistemi ed è importante che sia possibile mantenerla con la stessa facilità; qualsiasi macchina può essere aggiornata con il sottocomando di aggiornamento relativo, come nell'esempio seguente.

Listato 7.16 Il comando `docker-machine upgrade`.

```
root@vbox:/home# docker-machine upgrade host1
```

Questo comporterà l'arresto della macchina, l'aggiornamento e i relativi download (se previsti) e poi il riavvio della stessa, il tutto in maniera completamente automatizzata. Con questo comando possiamo eseguire, con il minimo sforzo, aggiornamenti continui sulla flotta di sistemi a disposizione.

Potrebbe capitare di aver bisogno di modificare o di vedere i file su una delle macchine o di accedere direttamente al terminale di una di esse; oppure potrebbe essere necessario recuperare il contenuto di un volume. Altre volte potrebbe essere necessario sottoporre a test la rete dall'host o personalizzarne la configurazione. In questi casi è possibile utilizzare i sottocomandi `ssh` e `scp`, attraverso i quali è possibile sfruttare un protocollo sicuro per avviare, rispettivamente, una sessione SSH oppure un processo di copia dei file.

Quando si crea un host con Docker Machine, questo processo crea o importa il file relativo alla chiave privata, la quale può essere utilizzata per l'autenticazione come utente privilegiato sulla macchina tramite il protocollo SSH. Il comando `docker-machine ssh` eseguirà l'autenticazione con la macchina richiesta e collegherà il terminale a una shell sulla macchina.

Per esempio, per creare un file sulla macchina denominata `host1`, possiamo eseguire i seguenti comandi.

Listato 7.17 Il comando `docker-machine ssh`.

```
root@vbox:/home# docker-machine ssh host1 "echo hello > file.txt"
```

L'avvio, l'arresto (“uccisione”) e la rimozione di una macchina sono comandi equivalenti a quelli usati per lavorare con i container. Docker Machine offre infatti quattro sottocomandi: `start`, `stop`, `kill` e `rm`.

Listato 7.18 I comandi `docker-machine start`, `stop` e `rm`.

```
root@vbox:/home# docker-machine start host1  
root@vbox:/home# docker-machine stop host1  
root@vbox:/home# docker-machine rm host1
```

Se l'utilizzo di Docker Machine al momento non sembra chiaro, vedremo nel Capitolo 8 come sfruttarlo appieno per la creazione degli host che ci aiuteranno nella configurazione di uno sciame con Docker Swarm.

Docker API

Le *API Docker* sono una serie di servizi REST che sostituiscono l'interfaccia a riga di comando remota e che possono tornare utili nel caso di installazioni di Docker su server remoti.

In questo paragrafo vedremo il funzionamento delle API Docker e come utilizzarle; analizzeremo il daemon Docker su una porta di rete e poi attraverso l'API ad alto livello e ne tratteremo gli aspetti chiave. Infine, esamineremo l'autenticazione dell'API tramite TLS.

Esistono tre API specifiche nell'ecosistema Docker.

- *API di registro*: fornisce l'integrazione con il registro Docker, che memorizza le nostre immagini.
- *API Docker Hub*: fornisce l'integrazione con Docker Hub.
- *Docker Remote API*: fornisce l'integrazione con il daemon Docker.

In questa parte ci concentreremo sull'ultimo caso, l'API, perché è la chiave per qualsiasi integrazione e interazione con Docker.

Esploriamo le *Remote API* e vediamo le sue funzionalità: innanzitutto, dobbiamo ricordare che le API sono fornite dal daemon Docker. Per impostazione predefinita, i daemon sono associati a una socket, `unix:///var/run/docker.sock` sull'host su cui questo è in esecuzione. Il daemon viene eseguito con i privilegi di root, in modo da disporre dei permessi necessari per gestire le risorse appropriate.

Le chiamate che andremo a vedere funzionano senza problemi se stiamo interrogando l'API dallo stesso host che esegue Docker, ma se vogliamo l'accesso remoto a un altro sistema, dobbiamo associare il daemon Docker a un'interfaccia di rete. Possiamo farlo passando o regolando il parametro `-H` al daemon Docker.

Se desideriamo utilizzare l'API Docker localmente, utilizziamo il comando `curl` per effettuare le interrogazioni, in questo modo.

Listato 7.19 Chiamata containers.

```
$ curl http://localhost:5555/containers/json
```

Se per esempio volessimo connetterci a un daemon remoto e ottenere informazioni sul numero di container e/o di immagini presenti, potremmo digitare il seguente comando, dove occorre sostituire a `docker.example.com` l'indirizzo di riferimento del server dove è presente il daemon Docker.

Listato 7.20 Il comando `docker info` tramite API.

```
root@vbox:/home# docker -H docker.example.com:123 info
Containers: 1
Images: 3
...

```

Ora che abbiamo stabilito e confermato la connettività con il daemon Docker remoto, proviamo a connetterci direttamente tramite le API. Cominciamo con alcune nozioni di base: lavorare con le immagini

Docker. Inizieremo ottenendo un elenco di tutte le immagini sul nostro daemon Docker.

Listato 7.21 Il comando docker images tramite API.

```
root@vbox:/home# curl http://docker.example.com:123/images/json
```

```
[  
{  
  "Id": "sha256:  
...  
",  
  "ParentId": "",  
  "RepoTags": [  
    "jamtur01/distributed_app:latest"  
  ],  
  "RepoDigests": [  
    "jamtur01/distributed_app@sha256:  
...  
"  
  ],  
  "Created": 1589296057914,  
  "Size": 123456,  
  "VirtualSize": 654321,  
  "Labels": {}  
...  
]
```

Abbiamo usato l'endpoint `/images/json`, che restituirà un elenco di tutte le immagini presenti sul sistema remoto; verranno fornite le stesse informazioni del comando `docker images`. Possiamo anche interrogare immagini specifiche tramite ID, proprio come avviene usando il comando `docker inspect`.

Listato 7.22 Il comando docker images con ID tramite API.

```
root@vbox:/home# curl http://docker.example.com:123/images/ID_IMMAGINE/json
```

Per scaricare un'immagine, così come faremmo con `docker pull`, possiamo utilizzare l'endpoint `/images/create` e passare come parametro `fromImage`.

Listato 7.23 Il comando docker images create tramite API.

```
root@vbox:/home# curl http://docker.example.com:123/images/create?fromImage=alpine
{"status":"Pulling from library/alpine","id":"3.1"}
{"status":"Pulling fs layer","progressDetail":{},"id":"8f13703509f7"}
...
```

Le API a disposizione espongono anche tutte le operazioni relative ai container disponibili nella riga di comando; possiamo infatti elencare tutti i container in esecuzione usando l'endpoint `/containers`, come faremmo normalmente con il comando `docker ps`.

Listato 7.24 Il comando docker containers tramite API.

```
root@vbox:/home# curl http://docker.example.com:123/containers/json
```

In questo modo verranno restituiti tutti i container in esecuzione sull'host remoto; per vedere anche quelli arrestati, è possibile passare come *query string* alla chiamata precedente `all=1`.

Listato 7.25 Il comando docker containers (elenco completo) tramite API.

```
root@vbox:/home# curl http://docker.example.com:123/containers/json?all=1
```

Possiamo anche usare le API per creare container utilizzando una richiesta POST all'endpoint `/container/create`. Nel seguente esempio, vediamo la creazione di un semplice container a partire dall'immagine `alpine`.

Listato 7.26 Il comando docker containers create tramite API.

```
root@vbox:/home# curl -X POST -H "Content-Type: application/json" \
http://docker.example.com:123/containers/create \
-d '{
  "Image": "alpine"
}'
```

Dopo aver eseguito questa chiamata, viene restituito l'ID del container che abbiamo appena creato ed eventuali avvisi. Possiamo configurare ulteriormente la creazione del nostro container aggiungendo delle informazioni tramite coppie chiave/valore al JSON passato come parametro.

Per avviare il container utilizziamo l'endpoint `/containers/ID/start`.

Listato 7.27 Il comando docker containers start tramite API.

```
root@vbox:/home# curl -X POST -H "Content-Type: application/json" \
http://docker.example.com:123/containers/ID_CONTAINER/
start \
-d '{
"PublishAllPorts":true
}'
```

Analogamente, è possibile arrestare, riavviare o “uccidere” un container, utilizzando, rispettivamente, gli endpoint `/containers/ID/stop`, `/containers/ID/restart` e `/containers/ID/kill`.

Per recuperare tutti i log relativi a un dato container, abbiamo a disposizione l’endpoint `/containers/ID/logs`:

Listato 7.28 Il comando docker containers logs tramite API.

```
root@vbox:/home# curl http://docker.example.com:123/containers/ID_CONTAINER/logs
```

I log di Docker

Quando si creano applicazioni a container, la gestione dei log è sicuramente uno degli aspetti più importanti per chi sviluppa. La gestione dei log aiuta infatti i team a eseguire il debug e a risolvere i problemi più rapidamente, facilitando l’identificazione delle problematiche, individuando i bug e semplificandone la risoluzione.

L’importanza del *logging* si applica ampiamente nelle applicazioni Docker; quando un’applicazione in un container registra dei log, questi vengono inviati ai flussi di output o di errore dell’applicazione.

La gestione dei log serve ad accedere a questi flussi e a scrivere i log su un file, dove vengono raccolti tutti quelli segnalati durante l’esecuzione sull’host o tramite l’accesso a un servizio esterno.

Per impostazione predefinita, Docker utilizza dei file in formato JSON per strutturare i log; questi vengono salvati in un file specifico del container sull’host su cui è in esecuzione.

L’esempio seguente mostra i log registrati tramite un file JSON.

Listato 7.29 Log in formato JSON.

```
{"log":"Hello World!","stream":"stdout","time":"2020-03-29T22:51:31.549390877Z"}
```

*****ebook converter DEMO

Watermarks*****

I log di un container Docker vengono generati al suo interno, quindi devono essere raccolti direttamente accedendo al suo file system. Tutti i messaggi che un container invia allo standard di output viene registrato, quindi trasferito a un driver di log che lo inoltra a una destinazione remota di propria scelta.

Vediamo ora alcuni comandi di base Docker per scoprire l'uso dei log e delle statistiche che vengono elaborate.

Listato 7.30 Il comando logs.

```
root@vbox:/home# docker logs c1
```

Questo primo comando mostra tutti i log generati all'interno del container `c1`; qualora volessimo mostrare solo quelli più recenti, possiamo aggiungere il parametro `-f`.

Per avere informazioni sulla memoria utilizzata, le prestazioni della CPU su tutti i container in esecuzione, è sufficiente utilizzare il comando `docker stats`; qualora avessimo bisogno di visionare le informazioni relative a uno o più container specifici, è sufficiente passare i loro nomi come parametri aggiuntivi del comando.

Listato 7.31 Il comando stats.

```
root@vbox:/home# docker stats c1
```

Per mostrare gli eventi Docker registrati, è possibile usare il comando `docker events`, come mostrato di seguito.

Listato 7.32 Il comando events.

```
root@vbox:/home# docker events
2020-01-05T00:34:52 container create ... (image=alpine:latest, name=test)
2020-01-05T00:35:59 network connect ...
2020-01-05T00:36:07 container start 0fdb...ff37
```

Gli eventi che possono verificarsi sono molteplici; se parliamo di un container, possiamo avere eventi come `create, kill, rename, pause, stop` e `restart`, che sono alcuni di quelli che abbiamo visto quando abbiamo analizzato il ciclo di vita di un container; con le immagini, possono verificarsi eventi come `pull, push, delete, save, tag` e così via, quindi tutte operazioni già viste in precedenza; queste sono tutte informazioni utili a prevenire o a risolvere delle problematiche, perché permettono di

tracciare la storia dell'esecuzione di un container piuttosto che la manutenzione di un'immagine.

La maggior parte dei metodi di analisi convenzionali dei log non funziona utilizzando dei container, per diverse ragioni: prima di tutto, ricordiamo che i container sono effimeri, e quindi i log sono salvati all'interno del container in esecuzione, sotto la cartella

`/var/lib/docker/containers`. Questi si perdono una volta che il container viene riavviato o anche arrestato; è possibile però raccogliere i log con sistema di gestione e archiviarli in una cartella in cui saranno disponibili permanentemente. È sicuramente poco utile conservare i log solo sull'host Docker e questo per diversi motivi, perché questi possono accumularsi nel tempo e consumare spazio sul disco (se il container rimane in esecuzione a lungo). Ecco perché sarebbe bene utilizzare una posizione ben definita per i salvare i log, soprattutto una posizione che sia esterna al container.

Un altro motivo è il fatto che le applicazioni a container spesso sono composte da più elementi, e non da un singolo container; per quanto basilare sia una soluzione, potrebbe capitare di lavorare con due o più livelli di aggregazione. Può dunque esserci la necessità di accedere ai log di più container, ma anche a quelli del server host su cui è in esecuzione il daemon Docker; in questi casi è sicuramente utile avere a disposizione un aggregatore di log che abbia accesso all'host e al file system all'interno del container, così da permettere anche di trovare un modo per correlare gli eventi di log ai diversi processi anziché ai rispettivi container.

L'accesso ai log Docker potrebbe sembrare impegnativo, ma ci sono alcune *best practice* da tenere a mente quando si lavora con applicazioni in esecuzione in un container.

Log tramite applicazione

Con questa tecnica, l'applicazione all'interno dei container gestisce i propri log utilizzando un *framework ad hoc*; per esempio, nelle applicazioni Java si potrebbe utilizzare `Log4j2` per raccogliere e inviare i log dall'app a una cartella remota, *bypassando* sia Docker sia il sistema operativo dell'host.

Questo approccio offre agli sviluppatori il massimo controllo sulla gestione dei log; tuttavia, crea un carico aggiuntivo sul processo dell'applicazione. Se il framework di registrazione è limitato al container stesso, considerata la natura transitoria dei container, tutti i log memorizzati nel file system verranno cancellati, se questo viene chiuso o arrestato.

Per conservare questi dati, sarà però necessario configurare un sistema di memorizzazione oppure inoltrare i log a una destinazione remota, utilizzando uno dei tanti sistemi di gestione a disposizione. Inoltre, questo tipo di logging diventa difficile quando si distribuiscono più container identici, poiché costringe a trovare un modo per capire quale evento catturato dai log appartiene a quale container.

Utilizzare un volume

Come accennato in precedenza, un modo per aggirare il problema consiste nell'utilizzare dei volumi; con questo approccio si crea una directory all'interno del proprio container, la quale si collega a una directory sul computer host, in cui i dati verranno archiviati in modo permanente e possono anche essere condivisi, indipendentemente da ciò che accade al proprio container. In questo modo è possibile effettuare copie e backup e accedere ai log anche da altri container.

È inoltre possibile condividere il volume su più container; il rovescio della medaglia è che l'uso di volumi di dati rende difficile spostare i container su host diversi senza causare una perdita di dati.

Logging driver

Un'altra opzione per la gestione dei log quando si lavora con Docker è utilizzare il driver apposito; a differenza dei volumi, il driver Docker legge i dati direttamente dall'output del container. La configurazione predefinita scrive i log in un file del computer host, ma modificando il driver è possibile inoltrare gli eventi registrati in diversi modi.

Poiché i container non dovranno più scrivere e leggere dai file di registro, è probabile che siano visibili dei miglioramenti in termini di prestazioni; tuttavia, ci sono alcuni svantaggi nell'utilizzo di questo

approccio: i comandi di registro Docker funzionano solo con un driver che usa file JSON; inoltre offre funzionalità limitate, dal momento che consente solo l'invio dei log senza poterli analizzare.

Log container

Un'altra soluzione è quella di avere un container dedicato esclusivamente alla registrazione e alla raccolta dei log, il che lo rende più adatto all'architettura a microservizi. Il vantaggio principale di questo approccio è che non dipende da una macchina host; al contrario, avere a disposizione un container dedicato consente di gestire i file di log all'interno dell'ambiente Docker. Aggregherà automaticamente i log di altri container, e li potrà monitorare, analizzare e memorizzare o inoltrare a una destinazione remota.

Questo approccio di registrazione semplifica lo spostamento dei container tra host e il ridimensionamento dell'infrastruttura di log semplicemente aggiungendo nuovi container di cui sia necessario salvare i log. Allo stesso tempo, consente di raccogliere i log attraverso vari flussi di eventi e fornire anche delle statistiche.

Questo è l'approccio che viene sempre consigliato utilizzare. È possibile configurare *Logagent* come container per il logging e fare in modo che tutti i log registrati vengano inviati ai log di Sematext in pochi minuti, come vedremo a breve.

I *logging driver* abbiamo visto essere dei meccanismi di Docker per la raccolta di dati da container e dai servizi in esecuzione per renderli disponibili per l'analisi. Ogni volta che viene creato un nuovo container, Docker configura automaticamente il log driver JSON, se non è stata specificata alcuna altra opzione; in questo modo consente di implementare e utilizzare i plugin disponibili se si desidera integrare altri tipi di cattura dei log.

Nel caso volessimo eseguire un container con un logging driver customizzato, per esempio *syslog*, possiamo digitare il comando seguente, dove avviamo un container che riporterà i log registrati all'indirizzo specificato.

Listato 7.33 Logging driver personalizzato.

```
root@vbox:/home# docker run --log-driver syslog --log-opt syslog-
address=tcp://syslog-server:123 \
```

*****ebook converter DEMO

Watermarks*****

```
alpine echo hello world
```

Quando si tratta di configurare questo tipo di driver, sono disponibili due opzioni:

- impostare un driver di log predefinito per *tutti i container*;
- specificare *un driver per ciascun container*.

Nel primo caso, il driver predefinito è un file JSON, ma, come menzionato sopra, sono disponibili molte altre opzioni; è possibile passare a un altro driver modificando il file di configurazione e cambiando il valore del parametro `-log-driver` e `-log-opt`, come nell'esempio sopra riportato.

Riportiamo qui alcune opzioni che è possibile utilizzare, oltre il file JSON di default e *syslog*.

- `logagent`: gestione dei log per scopi generici. L'immagine di *logagent* è preconfigurata per la raccolta dei log su container; raccoglie non solo i log, ma aggiunge anche metadati come il nome dell'immagine, ID del container, nome del container, servizio *Swarm* o metadati *Kubernetes*. Inoltre, gestisce i log multilinea e può analizzarne le informazioni.
- `splunk`: scrive i log su Splunk utilizzando *HTTP Event Collector* (HEC).
- `syslog`: invia i dati a un server *syslog*; questa opzione è piuttosto diffusa per la registrazione dei log delle applicazioni.
- `journald`: invia i log del container al journal di `systemd`.
- `fluentd`: invia i log al raccoglitore *Fluentd* sotto forma di dati strutturati.
- `elf`: scrive i log del container su un endpoint GELF (*Graylog Extended Log Format*) come Graylog o Logstash.
- `awslogs`: invia i log ai registri *AWS CloudWatch*.
- `cplogs`: invia i log al sistema *Google Cloud Platform Logging*.

Il modo più affidabile e conveniente per la raccolta dei log rimane quello di utilizzare il driver di default e impostare un *log shipper* per

salvare queste informazioni; in questo modo, si ha sempre una copia locale dei log sul server e si ha anche il vantaggio della gestione centralizzata.

Per utilizzare *Sematext Logagent* è necessario seguire alcuni semplici passaggi, per iniziare a inviare i log. Infatti, dopo aver creato un'applicazione per la sola registrazione dei log, occorre eseguire questi comandi in un terminale.

Listato 7.34 Usare Sematext Logagent.

```
root@vbox:/home# docker pull sematext/logagent
docker run -d --restart=always --name st-logagent \
-e LOGS_TOKEN=YOUR_LOGS_TOKEN \
-e LOGS_RECEIVER_URL="https://logsene-receiver.sematext.com" \
-v /var/run/docker.sock:/var/run/docker.sock \
sematext/logagent
```

In questo modo, il container invierà a Sematext i log di tutti i container presenti; Logagent è uno *shipper* abbastanza comune, dal momento che l'immagine Docker Logagent è preconfigurata per la raccolta dei log su container e funziona come un piccolo container in esecuzione su ogni host, dove raccoglie i log per tutti i nodi presenti in un cluster e i loro container.

Il driver predefinito di Docker e la modalità di invio dei log possono influire notevolmente sulle prestazioni delle applicazioni a container. Si consiglia infatti di utilizzare un driver basato su file JSON per ottenere una registrazione affidabile, prestazioni costanti e osservabilità utilizzando uno strumento di registrazione centralizzato come Sematext Cloud.

Questa soluzione offre la gestione e l'analisi dei log senza problemi per qualunque infrastruttura o applicazione, e consente di filtrare, analizzare e creare avvisi sui log di tutte le tue applicazioni.

Sicurezza

Per utilizzare Docker in modo sicuro, è necessario essere consapevoli dei potenziali problemi di sicurezza e dei principali strumenti atti a proteggere i nostri sistemi basati su container. In questo paragrafo

andremo a considerare la sicurezza dal punto di vista dell'esecuzione di Docker in sede di produzione, ma la maggior parte dei consigli è ugualmente applicabile agli ambienti di sviluppo.

Quando capita di leggere su qualche blog o rivista online degli articoli su Docker, si può avere l'impressione che Docker sia intrinsecamente non sicuro e inadatto all'uso in un ambiente di produzione. Sebbene sia certamente necessario essere consapevoli dei problemi relativi all'utilizzo sicuro dei container, se utilizzati correttamente, essi possono fornire un sistema più sicuro ed efficiente rispetto all'utilizzo di macchine virtuali o ai classici server fisici.

Principali aspetti

La domanda che dovremmo porci, a questo punto, è a quali tipi di aspetti in materia di sicurezza dovremmo pensare in un ambiente basato su container. Vediamo nel seguente elenco una serie di tematiche su cui è necessario lavorare.

- *Exploit del kernel.* A differenza di quanto accade in una VM, il kernel è condiviso da tutti i container, e questo aumenta l'importanza di eventuali vulnerabilità presenti nel kernel. Se un container causa un *kernel panic*, verrà persa ogni informazione presente sul sistema host. Nelle macchine virtuali, la situazione è sicuramente diversa: un utente dovrebbe predisporre un attacco attraverso il kernel della macchina virtuale e passare per l'hypervisor prima di poter arrivare al kernel dell'host.
- *Accesso a dati sensibili.* Quando un container accede alle informazioni presenti in un database o servizio, deve utilizzare delle credenziali per farlo, oppure una chiave. Un utente malintenzionato che possa accedere a questi dati avrà anche accesso al servizio.
- *Immagini corrotte.* Se un utente inviasse a un altro un'immagine da lui creata per entrare nel sistema e copiare informazioni sensibili, sia l'host sia i dati sono a rischio. È necessario, come abbiamo visto in precedenza, essere sicuri che le immagini in esecuzione siano aggiornate e non contengano versioni di software con vulnerabilità.

note, sempre cercando di affidarsi a quelle disponibili sui registri ufficiali.

- *Attacchi DoS*. Tutti i container condividono le risorse del kernel; questo vuol dire che se un container può monopolizzare l'accesso a determinate risorse, inclusa la memoria e altre risorse come gli ID utente (UID), può lasciare senza risorse altri container, causando la non disponibilità del servizio, e gli utenti che ne usufruiscono regolarmente non saranno più in grado di accedere a parte del sistema o, peggio, all'intero sistema.
- *Breakout del container*. Un utente che ottiene l'accesso a un container non dovrebbe essere in grado di accedere ad altri container o all'host; dal momento che qualsiasi processo che esce dal container avrà gli stessi privilegi sull'host come nel container, è necessario preoccuparsi di potenziali attacchi che utilizzino un'escalation dei privilegi, in cui un utente ottiene privilegi elevati come quelli dell'utente *root*, spesso sfruttando un bug nel codice dell'applicazione. Dato che la tecnologia dei container è recente, è necessario gestire in maniera opportuna la sicurezza, in base al presupposto che i breakout sono improbabili, ma comunque possibili.
- *Namespace e container*. Non tutte le risorse alle quali un container ha accesso fanno parte del cosiddetto *namespace*. Per esempio, i processi hanno un namespace con un valore distinto dall'host: un processo di un container che abbia ID 123 non è lo stesso processo 123 presente su un altro container, o anche sull'host. Esistono però risorse che non fanno parte di alcun namespace e che quindi rappresentano possibili punti a vantaggio di un utente malintenzionato:
- *UID*. Abbiamo incontrato il termine UID parlando della gestione degli utenti all'interno di un container; gli utenti hanno lo stesso UID sia nel container sia nell'host, e ciò significa che se un container è in esecuzione come utente root e si verifica un breakout, l'hacker avrà i privilegi dell'utente root sull'host.
- *Il kernel (e relativi moduli)*. Se un container carica un modulo del kernel (che richiede privilegi extra), il modulo sarà disponibile su tutti i container e sull'host; ciò chiaramente include anche i moduli

di sicurezza di Linux che verranno discussi più avanti e che possono essere sfruttati per causare diversi problemi.

- *Dispositivi*. Tutte le unità, compresi i dischi, le schede audio e altro.

Attualmente Docker utilizza cinque namespace per gestire la visualizzazione dei processi del sistema: *Process*, *Mount*, *Network*, *Shared Memory* e *Hostname*.

Mentre questi offrono all'utente un certo livello di sicurezza, purtroppo non rappresentano un insieme abbastanza completo dei possibili *namespace*, come invece avviene nel caso di una *Kernel-based Virtual Machine*. In un ambiente di questo tipo i processi in una macchina virtuale non parlano in modo diretto con il kernel host, e dunque non hanno accesso ai file system del kernel, come nel caso di cartelle come `/sys` e `/sys/fs`, oppure `/proc/*`.

Come accennato prima, nel caso di una macchina virtuale, al fine di ottenere un'escalation dei privilegi, il processo attuato dall'utente deve riuscire ad accedere al kernel della VM, trovare una vulnerabilità in HyperVisor, superare i controlli SELinux (*sVirt*) - che sono molto stringenti su una VM - e infine attaccare il kernel dell'host.

Eseguendo invece un container, si è già arrivati al punto in cui si sta parlando con il kernel host, e dunque tutto il processo descritto prima non è necessario; infatti, riuscendo a comunicare con o ad attaccare uno di questi con un utente con privilegi abbastanza alti, è possibile ottenere il pieno controllo del sistema.

Come agire

Garantire un tipo di sicurezza impenetrabile per qualsiasi sistema IT e per i servizi aziendali è stata una delle esigenze principali e delle sfide principali per decenni. Una mente brillante può identificare e sfruttare tutti i tipi di falle e difetti di sicurezza introdotti con noncuranza e anche inconsapevolmente nel sistema. Esperti e ingegneri della sicurezza, d'altra parte, provano ogni tipo di tecnica per fermare l'azione degli hacker, anche se non è possibile al momento avere una vittoria schiacciante.

Le organizzazioni e i governi di tutto il mondo stanno pertanto investendo molto nelle attività di ricerca sulla sicurezza, al fine di ridurre tutti gli incidenti relativi alla sicurezza; esistono infatti numerosi fornitori di prodotti specifici per la sicurezza e fornitori di servizi di sicurezza che operano al fine di ridurre al minimo le conseguenze irreparabili rappresentate dalle minacce alla sicurezza e alle vulnerabilità dei sistemi IT. Precisamente, per qualsiasi tecnologia esistente ed emergente, la sicurezza è l'aspetto più cruciale e critico, e che non deve in alcun modo essere preso alla leggera.

Docker è una tecnologia di containerizzazione a maturità rapida nello spazio IT e, nel passato, l'aspetto della sicurezza ha ricevuto un'importanza primaria, considerando il fatto che l'adozione e l'adattamento dei container Docker sono costantemente in aumento.

Inoltre, un flusso di container specifici si sta spostando negli ambienti di produzione e, quindi, l'enigma della sicurezza acquisisce un significato speciale, soprattutto quando questi container si occupano della gestione di informazioni o servizi di pubblica utilità, come nel caso della pubblica amministrazione.

Il dibattito in materia di sicurezza che compie un paragone tra macchine virtuali e container è sempre molto attivo: ci sono molti aspetti a favore dell'uno e dell'altro caso, e come abbiamo visto non esiste un vincitore assoluto.

Nel caso del paradigma di virtualizzazione, gli *hypervisor* sono i controller centralizzati delle macchine virtuali. Qualsiasi tipo di accesso alle macchine virtuali deve passare attraverso questa soluzione, che pertanto si pone come un solido muro per qualsiasi tipo di attività per cui non si abbiano i privilegi sufficienti; pertanto in questo senso, la superficie di attacco di una macchina virtuale è più piccola rispetto a quella dei container.

Contrariamente al paradigma della virtualizzazione, i container sono posizionati direttamente in un livello superiore rispetto al kernel del sistema host. Questa architettura snellisce la soluzione complessiva e offre un'efficienza molto più elevata, perché elimina completamente lo strato di emulazione di un hypervisor. Tuttavia, a differenza del paradigma della macchina virtuale, quello del container non ha molti livelli, il che porta a un accesso più rapido e semplice al sistema host e ad altri container, se uno di essi viene compromesso.

Nei paragrafi seguenti, discuteremo la sicurezza così come è stata progettata nel sistema; esamineremo le soluzioni per migliorare sostanzialmente la sicurezza del container e le *best practice* e linee guida da adottare.

Attenzione: le specifiche presenti in questo paragrafo si basano sull'esperienza, ma questo non è un manuale di sicurezza, né la sicurezza può essere il suo tema predominante. Il tema sicurezza meriterebbe un libro a se stante.

Difesa in profondità

Vediamo dunque come agire all'atto pratico: per cominciare, diamo per certo che esistano vulnerabilità e costruiamo la cosiddetta *defence in depth*. Considerando l'analogia con un castello, che ha più livelli di difesa intorno a sé (le mura, il fossato, le torri di controllo e così via), è necessario adottare diverse tipologie di misure per contrastare vari tipi di attacco.

Anche le difese del sistema dovrebbero essere composte da più livelli. Per esempio, molto probabilmente i container dovrebbero essere eseguiti nelle macchine virtuali, in modo che se si verifica un breakout, un altro livello di difesa può impedire all'utente malintenzionato di raggiungere l'host o i container appartenenti ad altri utenti. In aggiunta a questo, dovrebbero essere messi in atto sistemi di monitoraggio di queste falle, che possano avvisare in tempo reale gli amministratori di sistema nel caso di comportamenti insoliti. Un altro tipo di strumento sono i *firewall*: questi dovrebbero limitare l'accesso di rete ai container, limitando la superficie di attacco esterna.

Least Privilege

Un altro principio importante al quale attenersi è quello di minimo privilegio; ogni processo o container deve essere eseguito con il minimo indispensabile dei diritti di accesso o delle risorse necessari per svolgere la sua funzione.

Il vantaggio principale di questo approccio è che se un container viene attaccato, l'hacker sarà fortemente limitato nella possibilità di accedere a ulteriori dati o risorse e quindi di causare danni irreparabili.

Buone pratiche in questo caso sono quelle riportate nel seguente elenco:

- accertarsi che i processi nei container *non vengano eseguiti con un'utenza root*, in modo tale che lo sfruttamento di una vulnerabilità presente in un processo non conceda l'accesso *root* all'autore della violazione;
- *eseguire i file system in sola lettura*, in modo che gli aggressori non possano sovrascrivere i dati o salvare script dannosi su file;
- *ridurre le chiamate del kernel che un container può effettuare*, per ridurre la potenziale superficie di attacco;
- *limitare le risorse che un container può utilizzare*, in modo da evitare attacchi DoS laddove un container o un'applicazione compromessi consumino le risorse fino ad arrestare il sistema host.

Immagini

Per usare le immagini in sicurezza, dobbiamo avere garanzie sulla loro provenienza: da dove sono state scaricate e chi le ha create. Dobbiamo essere sicuri che stiamo ottenendo esattamente la stessa immagine sottoposta a test dallo sviluppatore originale e che nessuno l'abbia manomessa durante l'archiviazione o il download. Se non possiamo verificarlo, l'immagine potrebbe essere corrotta o, peggio, sostituita con codice malevolo. Dati i problemi di sicurezza precedentemente discussi con Docker, questa è una delle maggiori preoccupazioni: dobbiamo supporre che l'immagine abbia pieno accesso all'host.

La provenienza è tutt'altro che un problema recente nel mondo dell'informatica: infatti, lo strumento principale per stabilire la provenienza del software o dei dati è l'hash sicuro, ossia qualcosa di simile a un'impronta digitale, ma creata per i dati: è una stringa relativamente piccola che è unica per i dati tale che qualsiasi modifica ai dati, comporti la variazione della stringa. Sono disponibili diversi algoritmi per il calcolo di hash sicuri, con vari gradi di complessità e garanzie dell'unicità del valore calcolato. Gli algoritmi più comuni sono SHA (presente in diverse varianti) e MD5 (che per diverse ragioni dovrebbe essere evitato). Se disponiamo di un hash sicuro per alcuni dati,

possiamo ricalcolare l'hash per i dati restanti e confrontarli. Se gli hash corrispondono, possiamo essere certi che i dati non siano stati danneggiati o manomessi.

Una domanda lecita potrebbe però essere la seguente: che cosa impedisce a un utente di modificare sia i dati sia l'hash? La miglior risposta è la firma crittografica e le coppie di chiavi pubbliche e private.

Attraverso la firma crittografica, possiamo verificare l'identità di chi pubblica del codice. Se uno sviluppatore firma, idealmente parlando, del codice sorgente con la sua chiave privata, qualsiasi destinatario di tale codice può verificare che provenga dal giusto mittente controllando la firma e utilizzando la chiave pubblica. Supponendo che il cliente abbia già ottenuto una copia della chiave dello sviluppatore e che la chiave di questo non sia stata compromessa, possiamo essere certi che il contenuto provenga da una fonte sicura e che non sia stato manomesso nel frattempo.

Isolamento dei container

Se disponiamo di un sistema host in cui operano diversi container per più utenti, dobbiamo assicurarci che ciascun utente sia collocato su un host Docker separato. Ciò è meno efficiente della condivisione degli host tra utenti e comporterà un numero maggiore di macchine, virtuali e non, rispetto al riutilizzo degli host, ma è importante per la sicurezza. Il motivo principale risiede nel fatto di evitare che l'accesso a un container apra una breccia per l'accesso a tutti gli altri. Nel caso della gestione isolata degli host Docker, se si dovesse verificare un accesso non autorizzato al sistema, l'utente si troverà comunque su una macchina separata dalle altre e non sarà in grado di accedere facilmente ai container appartenenti ad altri utenti.

Allo stesso modo, se disponiamo di container che elaborano o archiviano informazioni riservate, è bene tenerli su un host separato dai container che gestiscono informazioni meno sensibili e, in particolare, lontano dai container che eseguono applicazioni direttamente esposte agli utenti finali e connesse alla rete esterna; come abbiamo visto in alcuni esempi precedenti, nel caso di un'applicazione web, è bene separare lo strato visibile all'utente (*il frontend*) dalla gestione della logica applicativa e del database, così da limitare i danni.

L'isolamento e l'uso delle macchine virtuali possono anche fornire una protezione aggiuntiva contro gli attacchi DoS; gli utenti non saranno infatti in grado di monopolizzare la memoria dell'host e lasciare i container degli altri utenti senza risorse.

Aggiornamenti continui

La capacità di applicare rapidamente gli aggiornamenti a un sistema in esecuzione è fondamentale per mantenere la sua sicurezza, in particolare quando le vulnerabilità sono divulgate tramite strumenti di utility e framework comuni.

Identificare le immagini che necessitano di aggiornamento può richiedere un po' di lavoro, e il riavvio dei container a seguito dell'aggiornamento presuppone che sia in atto una sorta di supporto per gli aggiornamenti continui o che si desideri gestire i tempi di inattività.

Quando è il caso di correggere una vulnerabilità rilevata in un'immagine di terze parti, comprese le immagini ufficiali, questo dipende dal fatto che tale parte fornisca un aggiornamento in modo tempestivo. Per questo è necessario verificare se siano presenti aggiornamenti delle immagini in esecuzione, tramite i registri ufficiali o i siti di codice sorgente, ed effettuare la build del sistema, ripulendo la cache in modo che eventuali informazioni precedenti vengano rimosse.

Limitare l'accesso alla rete

Un container dovrebbe esporre solo le porte strettamente necessarie all'ambiente di produzione e tali porte dovrebbero essere accessibili solo ai container che ne hanno bisogno. Questo tipo di attività è semplice a dirsi, ma è un po' più complicata di quanto sembri, poiché per impostazione predefinita, i container possono comunicare tra loro indipendentemente dal fatto che le porte siano state esplicitamente pubblicate o esposte.

Docker offre molti tipi di isolamento, uno dei quali è l'isolamento della rete. Nel mondo Docker, la gestione della rete è estremamente importante, perché è il modo principale in cui i container comunicano con altri processi isolati senza scrivere su disco e senza accesso diretto agli eseguibili del sistema. La maggior parte delle configurazioni di rete è

costituita da opzioni che possono essere controllate tramite Docker o Docker Compose.

Immaginiamo uno scenario in cui abbiamo un modulo soggetto a possibili vulnerabilità, come un server che accetta richieste. È necessario creare dei criteri di accettazione che ne limitino l'accesso, come per esempio:

- il *container* dovrebbe accettare il traffico in entrata e in uscita da e verso una rete nota;
- il *container* dovrebbe bloccare il traffico in entrata e in uscita da e verso tutte le altre reti;
- l'applicazione all'interno del *container* deve essere eseguita come utente non privilegiato (vedi il paragrafo *Least Privilege*).

Queste sono regole abbastanza generiche, che però sono applicabili su piccola e grande scala: soprattutto nelle soluzioni che prevedono più container che devono cooperare tra loro, gestire in modo opportuno l'accesso ai diversi servizi può essere di grande aiuto per prevenire potenziali attacchi.

Hardening

Oltre a mantenere aggiornato il sistema operativo dell'host, è possibile prendere in considerazione l'idea di eseguire un kernel rinforzato, usando patch di sicurezza come quelle fornite da *grsecurity* e *PaX*. *PaX*, nello specifico, fornisce una protezione aggiuntiva contro gli utenti che intendono manipolare l'esecuzione di un dato programma modificando la memoria (come nel caso degli attacchi *buffer overflow*), contrassegnando il codice del programma come non scrivibile e i dati come non eseguibili.

grsecurity è progettato per funzionare insieme a *PaX* e aggiunge patch relative al controllo degli accessi in base al ruolo, al controllo e a varie altre funzioni.

Per abilitarne l'utilizzo, è necessario applicare le patch e compilare il kernel in maniera autonoma: non si tratta di un'attività particolarmente complessa, ed esistono diverse guide online per farlo, come quelle presenti su *WikiBooks*.

Tuttavia, è bene specificare che questi miglioramenti della sicurezza possono causare l'interruzione di alcune applicazioni; PaX, per esempio, entra in conflitto con qualsiasi programma che generi codice in fase di esecuzione. Inoltre, i controlli e le misure di sicurezza aggiuntivi comportano del sovraccarico, anche se questo è sicuramente un aspetto trascurabile rispetto alla gestione della sicurezza.

Gestione dell'audit

L'esecuzione di revisioni periodiche sui container e sulle immagini è un buon modo per garantire che il sistema rimanga "pulito" e aggiornato e per verificare che non si siano verificate violazioni della sicurezza. Un controllo in un sistema basato su container dovrebbe verificare che tutti quelli in esecuzione utilizzino immagini aggiornate e che tali immagini utilizzino un software sicuro. Qualsiasi differenza rispetto all'istanza di un container dall'immagine da cui è stata creata deve essere identificata e verificata. Inoltre, gli audit dovrebbero riguardare altre aree non specifiche dei sistemi basati su container, come il controllo dei log di accesso, le autorizzazioni dei file e l'integrità dei dati. Se gli audit possono essere ampiamente automatizzati, possono essere eseguiti per rilevare eventuali problemi il più rapidamente possibile.

Invece di dover accedere a ciascun container ed esaminarlo in maniera individuale, possiamo controllare l'immagine utilizzata per costruire il container e utilizzare Docker per verificare la presenza di eventuali modifiche non motivate rispetto all'immagine originale.

Il carico di lavoro necessario all'auditing può essere notevolmente ridotto eseguendo immagini minime che contengono solo i file e le librerie essenziali per l'applicazione.

Inoltre, è necessario controllare il sistema host come si farebbe con un normale computer host o VM e quindi accertarsi che il kernel sia aggiornato.

Diversi strumenti sono già disponibili per il controllo dei sistemi basati su container, come per esempio *Docker Bench for Security*, che verifica la conformità con molti dei suggerimenti presenti all'interno del documento *Docker Benchmark* del *Center for Internet Security* (CIS). Si tratta di uno script che riprende dozzine di *best practice* relative all'implementazione dei container Docker in produzione e le mette in

atto. I test sono tutti automatizzati; può essere facilmente installato tramite SSH sul sistema in uso ed eseguito seguendo le indicazioni presenti sul repository ufficiale: <https://github.com/docker/docker-bench-security>.

Che cosa abbiamo imparato

- Come ottimizzare il Dockerfile e quali regole applicare per renderlo efficiente al massimo.
- Quanto sia importante l'ordine delle istruzioni inserite nel Dockerfile.
- In che modo sfruttare l'istruzione `RUN` per evitare di creare livelli intermedi non necessari durante la build di un container.
- Che cosa vuol dire che un container è *effimero* e che cosa comporta.
- L'importanza del file `.dockerignore` e in che modo utilizzarlo per evitare di aggiungere al container file superflui o con dati sensibili.
- In che modo è possibile scegliere l'immagine di base di un'applicazione e quali sono i vantaggi e gli svantaggi principali delle immagini basate su sistemi operativi *Unix-based* più comuni.
- Come utilizzare strumenti come Kitematic o Docker Toolbox per sviluppare con Docker.
- Come sfruttare Docker Machine per gestire gli host, crearli e ispezionarli, al fine di avere a disposizione macchine virtuali facilmente configurabili.
- Che cosa sono le Docker API, come sfruttarle su un server remoto e le principali differenze rispetto ai comandi finora utilizzati.
- Che cosa sono i log, come recuperarli e come sfruttarli per gestire al meglio eventuali problematiche relative ai container.
- Il tema sicurezza in merito ai container e alle immagini Docker: quali sono le principali vulnerabilità e che tipo di azioni è possibile compiere per mettere in sicurezza una soluzione.

Strumenti

Finora abbiamo visto come creare container, definire immagini personalizzate, ma ignorando ciò che implica la cooperazione tra più elementi di questo tipo; abbiamo visto la gestione di una rete Docker e come i vari container possono comunicare tra loro, ma non come possono lavorare insieme e costituire un'unica soluzione. Nella realtà, infatti, le soluzioni software non sono fatte di un singolo “oggetto” che lavora ed esegue attività, ma spesso questi componenti sono disaccoppiati, in modo che ognuno di essi compia una serie di operazioni ben precise e che possa essere in collegamento costante con altri elementi, con i quali scambia (riceve o comunica) informazioni.

Per permettere a più container di lavorare insieme, e non solo di essere messi in comunicazione tra loro, è fondamentale introdurre un meccanismo di gestione dei container che lavorano con uno stesso obiettivo; immaginando di avere un'applicazione web su un primo container che recupera informazioni eseguendo chiamate a un database su un secondo container, secondo quanto visto finora, questi devono essere avviati manualmente e possono comunicare tramite la configurazione di una rete bridge; che cosa succede, però, se uno dei due, per esempio il database, per qualche motivo viene arrestato? L'altro container, che dipendente da questo, potrebbe presentare dei malfunzionamenti.

In questo senso, è necessario introdurre il concetto di *orchestrazione*: in Docker è un concetto molto diffuso, che permette la gestione di più container in simultanea, avendo a disposizione strumenti che vedremo a breve e che permettono di gestire l'avvio automatico dei container, di gestirne una rete di comunicazione interna e verso l'esterno, oltre che di definire le proprietà in termini di volumi o esecuzione.

L'orchestrazione dei container è il processo automatico di gestione o pianificazione del lavoro dei singoli container per applicazioni basate su microservizi (e non solo). Le piattaforme di orchestrazione dei container, ampiamente diffuse, si basano su soluzioni *open source* come *Kubernetes*, *Docker Swarm* o la versione commerciale di *Red Hat OpenShift*.

Tra i vantaggi nella gestione e l'automatizzazione delle attività presenti nell'orchestrazione di container, vi sono i seguenti.

- *Configurazione e pianificazione dei container*: questi vengono definiti all'interno di un unico file, che ne incapsula le proprietà e le informazioni di base per renderli operativi.
- *Disponibilità di container*: è possibile gestire il riavvio dei *container* in caso di errore, così da avere un'alta disponibilità dei servizi.
- *Ridimensionamento dei container*, per bilanciare equamente i carichi di lavoro delle applicazioni attraverso l'infrastruttura: questo garantisce il concetto di scalabilità di un'applicazione.
- *Bilanciamento del carico, instradamento del traffico e individuazione del servizio dei container*.
- *Monitoraggio dello stato dei container*: avendo una gestione unica, è possibile avere sempre a disposizione una panoramica dello stato di salute delle istanze e monitorarne il corretto funzionamento.
- *Protezione delle interazioni tra container*: è possibile definire delle reti diversificate a seconda dell'esposizione che si vuole garantire ai container.

Per vederne il funzionamento nel dettaglio, in questo capitolo esamineremo due possibili strumenti che permettono l'orchestrazione dei container; nel Capitolo 10 andremo a vedere un'altra alternativa che viene utilizzata su scala molto più grande e che permette non solo di orchestrare i container, ma anche di gestire i cluster in maniera più complessa (vedremo in seguito le differenze).

Docker Swarm

Docker Swarm è un gruppo di macchine fisiche o virtuali che eseguono l'applicazione Docker e che sono state configurate per unirsi a formare un *cluster*; una volta che un gruppo di macchine è stato raggruppato, è ancora possibile eseguire i comandi Docker ai quali siamo abituati, che però ora verranno eseguiti all'interno del cluster. Le attività del cluster sono controllate da *Swarm Manager* e le macchine che fanno parte del cluster vengono denominate *nodi*.

Docker Swarm è uno strumento di orchestrazione di container, il che significa che consente all'utente di gestire più container distribuiti su più macchine host. Uno dei principali vantaggi associati al funzionamento di uno Docker Swarm è l'elevato livello di disponibilità offerto per le applicazioni. In uno "sciame" in genere sono presenti diversi nodi di lavoro e almeno un nodo gestore, il quale è responsabile della gestione efficiente delle risorse dei nodi di lavoro e di garantire che il cluster funzioni in modo efficiente.

Nodi

Uno *sciame* è costituito da un gruppo di macchine, fisiche o virtuali, che operano in un *cluster*. Quando una macchina si unisce al cluster, diventa un nodo in quello sciame. Docker Swarm distingue tre diversi tipi di nodi, ognuno con un ruolo diverso all'interno dell'ecosistema.

- *Nodo Manager*: la funzione principale di questa tipologia di nodi è quella di assegnare attività ai nodi di lavoro presenti nello sciame. I nodi manager aiutano anche a svolgere alcune attività di gestione necessarie per far funzionare lo sciame. Viene consigliato di utilizzare un massimo di sette nodi manager per uno sciame.
- *Nodo Leader*: quando viene creato un cluster, viene utilizzato un algoritmo di consenso (*Raft*) per definire uno dei nodi come "leader". Il nodo leader prende tutte le decisioni di gestione e orchestrazione delle attività dello sciame; se il nodo leader diventa non disponibile a causa di un'interruzione o di un errore, può essere eletto un nuovo nodo leader utilizzando nuovamente l'algoritmo di consenso.

- *Nodo di lavoro*: in uno sciame con numerosi *host*, ciascun nodo di lavoro funziona ricevendo ed eseguendo le attività assegnategli dai nodi manager. Per impostazione predefinita, tutti i nodi manager sono anche nodi di lavoro e pertanto sono in grado di eseguire delle attività quando hanno le risorse disponibili per farlo.

Nel corso degli anni si è visto un numero crescente di sviluppatori adottare Docker e utilizzare Docker Swarm per produrre, aggiornare e gestire le applicazioni in modo più efficiente; questo tipo di tecnologia presenta diversi vantaggi, come quelli riportati di seguito.

- *Sfruttare la potenza dei container*. Gli sviluppatori utilizzano Docker Swarm perché sfrutta appieno i vantaggi progettuali offerti dai container; questi consentono agli sviluppatori di distribuire applicazioni o servizi in ambienti virtuali autonomi, un’attività che prima era eseguita tramite macchine virtuali. I container stanno dimostrando di costituire una versione più leggera delle macchine virtuali, poiché la loro architettura consente loro di utilizzare in modo più efficiente la potenza di calcolo.
- *Elevata disponibilità del servizio*. Uno dei principali vantaggi dell’utilizzo degli sciami è l’aumento della disponibilità delle applicazioni, attraverso la *ridondanza*. Per funzionare, uno sciame deve disporre di un manager in grado di assegnare le attività ai nodi di lavoro. Implementando più gestori, gli sviluppatori assicurano che il sistema possa continuare a funzionare anche se uno dei nodi manager fallisce.
- *Bilanciamento del carico automatizzato*. Docker Swarm pianifica le attività utilizzando una varietà di metodologie per garantire che siano disponibili risorse sufficienti per tutti i container. Attraverso un processo che può essere descritto come bilanciamento del carico automatizzato, il gestore assicura che i carichi di lavoro del singolo container siano assegnati per essere eseguiti sull’*host* più appropriato per maggiore efficienza. Gestisce anche il bilanciamento del carico e i DNS in modo molto simile a un host Docker locale; ogni sciame può esporre porte, proprio come i container Docker pubblicano porte. Come le porte del container, possono essere

definite automaticamente o manualmente. Lo sciame gestisce un DNS interno in modo molto simile a un host Docker, cosa che consente di rilevare servizi e lavoratori all'interno dello sciame.

Swarm ha un'architettura semplice: raggruppa insieme più host Docker e per gestirli utilizza le API standard. Questo tipo di lavoro è incredibilmente potente, perché aumenta l'astrazione di Docker e dei relativi container a livello di cluster, senza che sia necessario imparare a interagire con nuove API.

Docker Swarm, come molti altri strumenti Docker, segue il principio progettuale “le batterie sono incluse, ma possono essere rimosse”: ciò significa che viene fornito con strumenti per casi d'uso semplici e fornisce delle API per l'integrazione con strumenti più complessi. Swarm viene integrato con l'installazione Docker a partire dalla versione 1.12.

Algoritmo Raft

Raft è un algoritmo di consenso progettato per essere facile da capire. È equivalente a Paxos in termini di tolleranza agli errori e prestazioni; la differenza è che implementa un meccanismo di consenso basato sul leader. Il cluster ha un solo leader eletto, che è completamente responsabile della gestione della replica dei log sugli altri server del cluster; ciò significa che il leader può decidere la posizione dei nodi in ingresso e come gestire il flusso di dati tra questi senza consultare altri nodi. Un leader rimane tale fino a quando non va in errore o viene arrestato, nel qual caso viene eletto un nuovo leader.

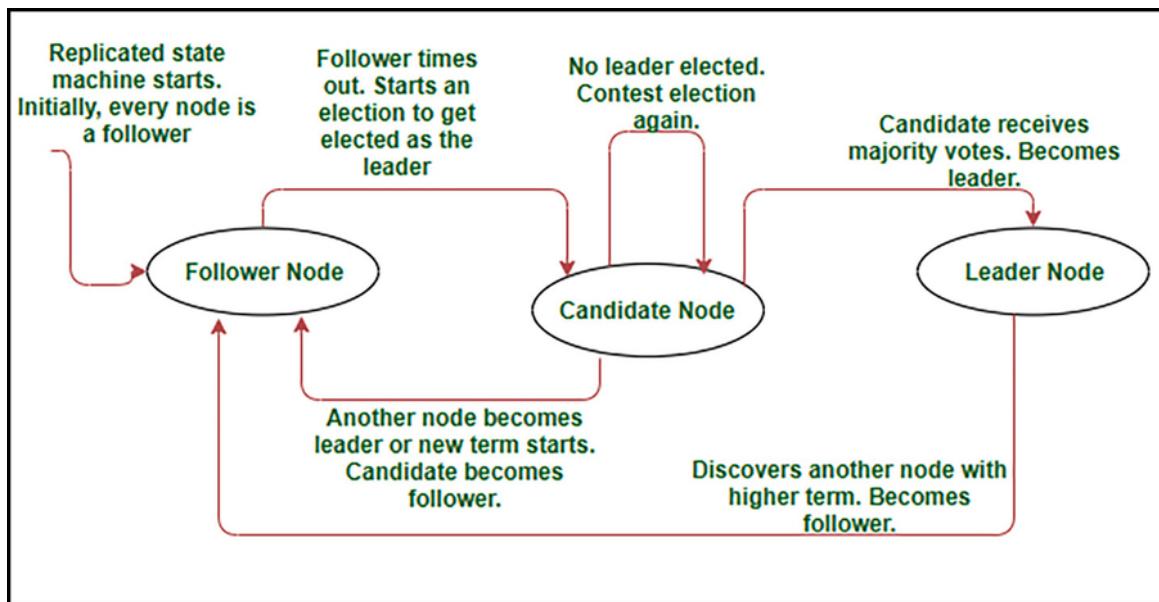


Figura 8.1 L'algoritmo Raft.

Il problema del consenso è scomposto in due sottoproblemi relativamente indipendenti, riportati di seguito.

Elezioni del leader

Quando il leader va in errore o quando si avvia l'algoritmo, è necessario scegliere un nuovo leader; in questo caso, viene inizializzato un nuovo *periodo*, ossia un lasso di tempo arbitrario entro il quale deve essere eletto un nuovo leader. Ogni periodo (*term* in inglese) inizia con l'elezione di un leader; se l'elezione viene completata con successo (ovvero il leader viene effettivamente eletto), il periodo continua con le normali operazioni orchestrate dal nuovo leader. Se l'elezione è un fallimento, inizia un nuovo mandato, con una nuova elezione.

L'elezione di un leader viene avviata da un server candidato. Un server diventa un candidato se non riceve alcuna comunicazione dal leader per un periodo chiamato *timeout* delle elezioni, e quindi si può presumere che non ci sia più alcun leader attivo. Inizia le elezioni aumentando il valore del periodo, votando come nuovo leader e inviando un messaggio a tutti gli altri server che richiedono il loro voto. Un server voterà una sola volta per trimestre, in base al principio "primo arrivato, primo a operare". Se un candidato riceve un messaggio da un altro server con un numero di termini almeno pari a quello attuale del candidato, l'elezione del candidato viene annullata e il candidato si trasforma in un seguace e riconosce il leader come legittimo. Se un candidato riceve la maggioranza dei voti, diventa il nuovo leader. Se non si verifica nessuno dei due casi, inizia un nuovo mandato e inizia una nuova elezione.

Raft utilizza un *timeout* per garantire che i problemi di voto vengano risolti rapidamente.

Replica dei log

Per motivi di semplicità, limiteremo l'ambito di applicazione a un client che effettua solo richieste di scrittura. Ogni richiesta fatta dal client è memorizzata nel log del leader; questo log viene quindi replicato su altri nodi, i *follower*. In genere, una voce di log contiene le tre informazioni seguenti:

- il comando specificato dal client;
- l'indice, su base 1, per identificare la posizione di ingresso nel log del nodo;
- il periodo, per accettare l'orario di arrivo del comando.

Il nodo *leader* attiva dei meccanismi chiamati *AppendEntries* su tutti i *follower* per sincronizzare i loro log con i suoi. Il leader continua a inviare questi segnali fino a quando tutti i *follower* replicano in modo sicuro la nuova voce nei loro log.

Quando la maggior parte dei server nel cluster copia correttamente le nuove voci nei propri log, viene effettuato il *commit*, ossia la conferma della chiusura dell'operazione. A questo punto, il leader esegue il *commit* della voce anche nel proprio log, per mostrare che è stata replicata con successo. Dopo il *commit* del log da replicare, il leader esegue l'operazione richiesta e risponde con il risultato al client. Queste richieste vengono eseguite nell'ordine in cui sono state ricevute.

Se due voci in due log diversi (*leader* e *follower*) hanno un indice e un periodo identici, significa che sono stati memorizzati come lo stesso comando e i relativi log risulteranno identici fino a quel punto.

Primi passi

Il modo più semplice per installare Swarm è utilizzare Docker stesso; vediamo dunque come utilizzare Docker Swarm per creare un piccolo sciame.

Il primo passo è creare un set di macchine Docker che fungeranno da nodi nel nostro Docker Swarm; in questo caso, creeremo uno swarm di sei macchine, in cui una fungerà da leader e le altre da nodi di lavoro.

Usiamo il comando standard per creare una Docker Machine chiamata `manager1` come mostrato di seguito.

Listato 8.1 Creazione di una Docker Machine (HyperV).

```
$ docker-machine create --driver hyperv manager1
```

Teniamo presente che questa operazione viene eseguita su una macchina con sistema operativo Windows, e che utilizza il gestore Hyper-V nativo, quindi stiamo utilizzando questo specifico comando. Avendo a disposizione VirtualBox, il comando sarebbe invece il seguente.

Listato 8.2 Creazione di una Docker Machine (VBox).

```
root@vbox:/home# docker-machine create --driver virtualbox manager1
```

Allo stesso modo, occorre creare i nodi di lavoro. Nel nostro caso, ne creeremo cinque, denominati `worker1`, `worker2` e così via.

Dopo la creazione, si consiglia di eseguire il comando `docker-machine ls` per verificare lo stato di tutte le macchine Docker.

Listato 8.3 Elenco delle Docker Machine presenti.

```
root@vbox:/home# docker-machine ls
NAME      DRIVER    URL                                STATE
manager1   hyperv   tcp://192.168.1.8:2376          Running
worker1    hyperv   tcp://192.168.1.9:2376          Running
worker2    hyperv   tcp://192.168.1.10:2376         Running
worker3    hyperv   tcp://192.168.1.11:2376         Running
worker4    hyperv   tcp://192.168.1.12:2376         Running
worker5    hyperv   tcp://192.168.1.13:2376         Running
```

Ricordiamo l'indirizzo IP del nodo `manager1`, perché tornerà utile a breve; per recuperarlo in un secondo momento, sarà sufficiente utilizzare il comando `docker-machine ip manager1`.

Dal momento che le macchine appena create sono vere e proprie istanze virtuali, è consigliabile utilizzare SSH come protocollo di accesso

alle macchine; questo è possibile utilizzando il comando `docker-machine ssh`.

Listato 8.4 Il comando docker-machine ssh.

Creazione di uno sciame

Ora che le nostre macchine sono installate, possiamo procedere con la configurazione dello sciame; la prima cosa da fare è inizializzarlo; lo faremo utilizzando SSH per entrare nella macchina `manager1`. Una volta avviato SSH, come visto nel comando precedente, dobbiamo eseguire questo comando all'interno del nodo, dove l'IP specificato è quello che corrisponde all'IP del nodo `manager1`, recuperato qualche passaggio fa.

Listato 8.5 Il comando docker swarm init.

```
docker@manager1:~$ docker swarm init --advertise-addr 192.168.1.8
```

Il risultato del comando precedente dovrebbe essere il seguente.

```
Swarm initialized: current node (...) is now a manager.  
To add a worker to this swarm, run the following command:  
  docker swarm join \  
    --token SWMTKN-1-5mgfyf6ehuc5pfbmar00njd3oxv8nmjhteejaald3yzbef7os11-  
ad7b1k8k3bl3aa3k3q13zivqd\  
    192.168.1.8:2377
```

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

Se l'output è simile a quanto riportato, vuol dire che lo sciame è stato creato e che il nodo `manager1` è stato inizializzato come `manager`. Possiamo anche notare che l'output riporta il comando `docker swarm join`, da utilizzare per aggiungere un altro nodo come nodo lavoratore.

A questo punto possiamo verificare lo stato dello sciame, lanciando il seguente comando.

Listato 8.6 Il comando docker node ls.

```
docker@manager1:~$ docker node ls
ID          HOSTNAME   STATUS    AVAILABILITY  MANAGER STATUS
ad7b1kbl3a...*  manager1  Ready     Active        Leader
```

Ciò dimostra che finora esiste un unico nodo, ossia `manager1`, e che ha il ruolo di leader. Vediamo come aggiungere un nodo di lavoro; per farlo, lanciamo il seguente comando, collegandoci in SSH al nodo di lavoro.

Listato 8.7 Il comando docker swarm join-token.

```
docker@worker1:~$ docker swarm join \
--token SWMTKN-1-5mgfyf6ehuc5pfbsmar00njd3oxv8nmjhteejaald3yzbef7os11-
ad7b1k8k3b13aa3k3q13zivqd \
192.168.1.8:2377

This node joined a swarm as a worker.
```

La stessa operazione andrà effettuata per tutti gli altri nodi; alla fine, andando a visualizzare l'elenco dei nodi presenti tramite il comando `docker node ls`, avremo un risultato simile al seguente.

Listato 8.8 Il comando docker node ls.

```
docker@manager1:~$ docker node ls
ID          HOSTNAME   STATUS    AVAILABILITY  MANAGER STATUS
...          worker4    Ready     Active
...          worker5    Ready     Active
...          worker3    Ready     Active
...*         manager1   Ready     Active        Leader
...          worker2    Ready     Active
...
...          worker1    Ready     Active
```

Per ottenere più informazioni di dettaglio sullo sciame appena creato, possiamo anche eseguire il comando `docker info`; l'output restituito riporta le informazioni relative al numero di nodi di lavoro presenti, al numero di manager, all'ID del cluster e all'indirizzo assegnato al nodo da cui si esegue il comando. Infatti, se eseguiamo questo comando tramite la sessione SSH del nodo `manager1`, abbiamo questo risultato.

Listato 8.9 Il comando docker info.

```
docker@manager1:~$ docker info
...
Swarm: active
  NodeID: ...
  Is Manager: true
  ClusterID: ...
```

*****ebook converter DEMO

Watermarks*****

```

Managers: 1
Nodes: 6
Orchestration:
  Task History Retention Limit: 5
Raft:
  Snapshot Interval: 10000
  Heartbeat Tick: 1
  Election Tick: 3
Dispatcher:
  Heartbeat Period: 5 seconds
CA Configuration:
  Expiry Duration: 3 months

Node Address: 192.168.1.8

```

Possiamo vedere che il campo `isManager` è `true`; se avessimo eseguito lo stesso comando in uno dei nodi di lavoro, il valore sarebbe ovviamente `false`.

Avvio del servizio

Quello che abbiamo appena fatto è stato definire sei nodi all'interno di uno sciame e definire il manager; questi nodi sono attualmente vuoti, perché non contengono alcun servizio. Tutto ciò che faremo è dire al nodo `manager1` di eseguire i container e il nodo si occuperà di pianificarne l'avvio, inviare i comandi ai nodi e distribuirli.

Per avviare un servizio, è necessario disporre di quanto segue.

- *L'immagine Docker da eseguire.* Nel nostro caso, eseguiremo quella di `Nginx`.
- *Il numero di porta da esporre,* essendo un web server.
- *Il numero di container (o istanze) da avviare,* specificato tramite il parametro `replicas`.

In questo esempio, andremo a lanciare cinque repliche del container Nginx. Per farlo, è necessario avviare nuovamente una sessione SSH per il nodo `manager1` e lanciare il seguente comando.

Listato 8.10 Il comando `docker service create`.

```
docker@manager1:~$ docker service create --replicas 5 -p 80:80 --name webserver
nginx
```

Quello che abbiamo appena fatto è stato creare un servizio di cui vengono generate cinque repliche (escludendo il nodo `manager1`), il cui

nome è `webserver` e lo facciamo a partire dall'immagine di base `nginx`.

Ispezione del servizio

Per verificare che le repliche siano in esecuzione, possiamo eseguire il comando `docker service ls`: se l'output fosse simile a quello riportato di seguito, vuol dire che ancora non sono state avviate tutte le repliche, e che l'avvio necessita di un po' di tempo in più.

Listato 8.11 Il comando docker service ls.

```
docker@manager1:~$ docker service ls
ID      NAME      REPLICAS     IMAGE      COMMAND
...
...      web       0/5          nginx
```

Possiamo vedere lo stato del dettaglio del servizio eseguendo il comando `docker service ps`, che ci mostra come questo viene gestito dai diversi nodi.

Listato 8.12 Il comando docker service ps.

```
docker@manager1:~$ docker service ps webserver
ID      NAME      IMAGE      NODE      DESIRED STATE      CURRENT STATE
7i*    webserver.1  nginx     worker3   Running      Preparing 3
minutes ago
22*    webserver.2  nginx     manager1  Running      Running 21
seconds ago
ab*    webserver.3  nginx     worker2   Running      Running 4 minutes
ago
bd*    webserver.4  nginx     worker5   Running      Running 43
seconds ago
cb*    webserver.5  nginx     worker4   Running      Running 5 minutes
ago
```

All'appello manca un nodo, il nodo `worker1`: questo vuol dire che la replica ancora non è stata avviata, e possiamo verificarlo eseguendo il comando `docker service ls` e controllando che il numero di repliche avviate sia 5 su 5. Una volta raggiunta questa fase, eseguendo il comando `docker ps` potremo vedere che il daemon Nginx è stato avviato e che il servizio è disponibile. Possiamo dunque visualizzare la pagina principale di Nginx utilizzando nel browser l'indirizzo IP di uno qualunque dei nodi, e andremo a visualizzare una pagina come quella rappresentata nella Figura 8.2.



Figura 8.2 Pagina predefinita di Nginx.

Scalare il servizio

Dopo aver distribuito un servizio in uno sciame, siamo pronti per utilizzare l’interfaccia a riga di comando Docker per ridimensionare il numero di container del servizio. I container in esecuzione in un servizio sono chiamati “attività”.

Avviamo quindi una sessione SSH nella macchina in cui abbiamo il nodo `manager1` e lanciamo il comando seguente per modificare lo stato del servizio in esecuzione nello sciame con il numero di attività che vogliamo impostare.

Listato 8.13 Il comando docker service scale.

```
docker@manager1:~$ docker service scale webserver=10
webserver scaled to 10
```

In questo modo, il nostro servizio avrà a disposizione non più cinque repliche, ma dieci: possiamo verificarlo eseguendo il comando `docker service ls` e vedendo che il numero è aumentato e che il nodo manager li sta avviando.

Listato 8.14 Il comando docker service ls.

```
docker@manager1:~$ docker service ls
ID          NAME          REPLICAS      IMAGE          COMMAND
...          webserver     5/10          nginx
```

Così come per i container, abbiamo a disposizione anche per i servizi il comando `docker node inspect`, che ci permette di ispezionare le informazioni relative a un nodo (passando il parametro `--pretty`, questo verrà presentato in un formato più leggibile).

Listato 8.15 Il comando `docker node inspect`.

```
docker@manager1:~$ docker node inspect --pretty worker1
ID: ...
Hostname: worker1
Joined at: 2020-03-18 12:33:21.5478505 +0000 utc
Status:
  State: Ready
  Availability: Active
Platform:
  Operating System: linux
  Architecture: x86_64
Resources:
  CPUs: 1
  Memory: 987.2 MiB
Plugins:
  Network: bridge, host, null, overlay
  Volume: local
Engine Version: 1.12.1
Engine Labels:
```

Gestione del servizio

Per rimuovere un servizio, è sufficiente utilizzare il comando `docker service rm`: questo rimuoverà tutti i nodi associati e anche lo sciame, ragion per cui ci vorrà qualche momento prima che l'operazione sia completata.

Listato 8.16 Il comando `docker service rm`.

```
docker@manager1:~$ docker service rm webserver
webserver
```

Per verificare che la rimozione sia andata a buon fine, possiamo eseguire il comando `docker service inspect` sul servizio oppure `docker ps`, per verificare che non siano più presenti i container a esso associati.

Listato 8.17 Il comando `docker service inspect`.

```
docker@manager1:~$ docker service inspect webserver
[]
Error: no such service: webserver
```

Infine, vediamo come aggiornare un’immagine di un servizio: questo può tornare utile nel momento in cui l’immagine utilizzata per la costruzione di uno o più nodi sia stata modificata e il servizio debba prenderne le modifiche.

Per fare ciò, possiamo eseguire il comando `docker service update`, passando come parametro l’immagine a partire dalla quale effettuare l’aggiornamento, come nel seguente esempio.

Listato 8.18 Il comando `docker service update`.

```
docker@manager1:~$ docker service update --image nginx:1.17.10 webserver  
webserver
```

Così facendo, il nodo `manager1` effettua l’aggiornamento ai nodi di lavoro in base alla politica di `UpdateConfig`: vengono dunque automaticamente applicati gli aggiornamenti come segue:

- viene arrestata l’attività;
- viene pianificato l’aggiornamento dell’attività interrotta;
- viene avviato il container per l’attività aggiornata;
- se questa va a buon fine, si attende un certo periodo di tempo prima di ripartire con l’aggiornamento dell’attività successiva;
- se l’aggiornamento di un’attività fallisce, viene sospeso l’aggiornamento.

Per specificare il tempo da attendere tra un aggiornamento e l’altro, è necessario specificare il parametro `--update-delay` durante la creazione del servizio, come nell’esempio sottostante.

Listato 8.19 Il comando `docker service create`: opzione `-update-delay`.

```
docker@manager1:~$ docker service create \  
--replicas 10 \  
--name nginx \  
--update-delay 20s \  
nginx
```

Docker Compose

Ora familiarizziamo con *Docker Compose*, con il quale definiamo un set di container che hanno in comune l’avvio e le proprietà di runtime,

*****ebook converter DEMO

Watermarks*****

tutte definite in un *file YAML*. Docker Compose chiama ciascuno di questi container “servizi” e li definisce come elementi che interagiscono con altri container e che hanno specifiche proprietà di runtime.

Installazione

Vediamo come installare Docker Compose e poi come utilizzarlo per creare un semplice stack di applicazioni con diversi container. Iniziamo installando Docker Compose. Attualmente è disponibile per Linux, Windows e macOS. Può essere installato direttamente come codice binario, tramite Docker per Mac o Docker per Windows o tramite un pacchetto Python Pip.

Per installare Docker Compose su Linux possiamo prendere il codice binario da GitHub ed eseguirlo; è possibile lanciare il seguente comando tramite il terminale, sostituendo eventualmente il numero di versione con quella in uso.

Listato 8.20 Installare Docker Compose su Linux.

```
root@vbox:/home# sudo curl -L  
"https://github.com/docker/compose/releases/download/1.25.5/docker-compose-$(uname  
-s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

Questo comando andrà a scaricare un pacchetto che può essere eseguito aggiungendo i relativi permessi di esecuzione.

Listato 8.21 Modifica dei permessi di esecuzione.

```
root@vbox:/home# sudo chmod +x /usr/local/bin/docker-compose
```

Come Docker, anche Docker Compose è attualmente supportato solo su installazioni Linux a 64 bit; se invece utilizziamo un sistema operativo Windows o macOS, Docker Compose viene fornito all’interno di Docker per Windows o Docker per Mac, rispettivamente.

In tutti i casi presentati, per sottoporre a test la corretta installazione di Docker Compose, è sufficiente digitare nel terminale il seguente comando, e verificare che l’output sia simile al seguente.

Listato 8.22 Verifica di installazione di Docker Compose.

```
root@vbox:/home# docker-compose --version  
docker-compose version 1.25.5, build 1110ad01
```

docker-compose.yml

La potenza di Docker Compose si basa tutta sul file di configurazione in formato YAML `docker-compose.yml`; questo file ha normalmente l'aspetto seguente.

Listato 8.23 Esempio di file `docker-compose.yml`.

```
version: "3"
services:
  nome-servizio1:
    build: "path-Dockerfile"
  nome-servizio2:
    image: [nome-immagine]
    container_name: [nome-container]
    restart: unless-stopped
    environment:
      - ""
    volumes:
      - [sorgente]:[destinazione]
    networks:
      - [nome-rete]
```

Nella prima riga specificiamo la versione utilizzata; la versione 3 è attualmente la più recente. Sono presenti varie versioni e il controllo delle versioni è essenziale per assicurarsi che le modifiche non rendano inutilizzabili le applicazioni o possano provocare malfunzionamenti; per verificare le versioni disponibili, è sufficiente controllare sul sito

<https://docs.docker.com/compose/compose-file/>.

Se non viene specificata la versione nel file, verrà considerata la versione 1; la notazione utilizzata è a due cifre; l'ultima versione è la 3.8, ma è sufficiente specificare la versione principale (ovvero la versione 3); è chiaro che per alcune nuove funzionalità potrebbe essere necessario aggiungere il numero esatto di versione a due cifre: 3.8.

Successivamente, andremo a elencare tutti i servizi che Docker Compose dovrà avviare e rendere disponibili; ogni applicazione in esecuzione avrà il proprio nome e sarà necessario fornire le informazioni per la build (proprietà `build`), come per esempio un Dockerfile, oppure un'immagine di base da recuperare tramite il registro Docker (proprietà `image`).

Nella sezione `services` stiamo definendo due servizi, ai quali devono essere assegnati dei nomi che verranno poi utilizzati come alias in fase di

esecuzione dei container, ma anche per eventuali dipendenze tra gli stessi.

I servizi corrispondono ai componenti dell'applicazione; a mano a mano che vengono separate le attività nell'architettura, sarà possibile creare immagini Docker separate per gestire operazioni separate. Questo viene fatto nell'ottica di lavorare con architetture a microservizi, che abbiamo visto garantire diverse proprietà nel caso di soluzioni che richiedono scalabilità.

NOTA

Un file YAML utilizza gli spazi come rientro del paragrafo, ed è possibile utilizzare 2 o 4 spazi per il rientro, ma *non* le tabulazioni. Perché YAML vieta l'uso delle tabulazioni? Perché sono trattate in modo diverso dai vari editor e strumenti e siccome lo spazio di indentazione è così importante per la corretta interpretazione di YAML, questo problema è stato risolto definendo regole che impongano uno standard univoco. In effetti, Guido van Rossum di Python ha riconosciuto che consentire l'uso delle tabulazioni nella scrittura di codice Python spesso è stato causa di problemi per molte persone e che se dovesse progettare Python oggi, le vieterebbe.

Un servizio, secondo Docker Compose, dovrebbe occuparsi di un'unica operazione, e per questa ragione è possibile specificare esattamente un'immagine per servizio. Vediamo anche che per ogni servizio sono definite diverse proprietà: vengono impostate le variabili d'ambiente (proprietà `environments`), le porte che devono essere esposte (proprietà `ports`) e i volumi esterni che devono essere montati (proprietà `volumes`). Tutte queste caratteristiche potrebbero essere definite tramite la riga di comando di Docker, ma in questo caso abbiamo l'opportunità di creare un file più pulito e più facile da leggere, che può essere eseguito in una sola volta.

Esempio: applicazione Flask con MongoDB

Lo sviluppo di applicazioni web può diventare complesso e richiedere molto tempo per la creazione e la manutenzione di diverse tecnologie. Considerare opzioni di peso più leggero progettate per ridurre la complessità e il tempo di produzione per l'applicazione può risultare ottimale per creare una soluzione più flessibile e scalabile. Come microframework web basato su Python, *Flask* offre agli sviluppatori un modo per far crescere le proprie applicazioni attraverso le diverse estensioni che possono essere integrate nei progetti. Per ottimizzare la

scalabilità dello stack tecnologico di uno sviluppatore, *MongoDB* è un database NoSQL progettato per adattarsi e funzionare con frequenti modifiche. Gli sviluppatori possono utilizzare Docker per semplificare il processo di imballaggio e distribuzione delle loro applicazioni; con Docker Compose ha ulteriormente semplificato l’ambiente di sviluppo, consentendo di definire la propria infrastruttura, inclusi i servizi delle applicazioni, i volumi e la gestione della rete, il tutto in un unico file. L’uso di Docker Compose, come abbiamo visto in diverse occasioni, offre facilità d’uso sull’esecuzione di più comandi in un container; consente di definire tutti i servizi in un unico file e con un singolo comando permette di creare e avviare tutti i servizi previsti dalla configurazione.

In questo mini-progetto verrà creato, configurato ed eseguito un esempio di applicazione web da eseguire con Flask, Nginx e MongoDB, sfruttando tutti container Docker e vedendo passo dopo passo l’utilizzo delle proprietà messe a disposizione da Docker Compose. Verrà definita l’intera configurazione dello stack in un file `docker-compose.yml`, insieme ai file di configurazione per Python, MongoDB e Nginx. Flask richiede che un server web serva le richieste HTTP, quindi per usare l’applicazione utilizzeremo anche Gunicorn, che è un server HTTP WSGI Python. Nginx funge da server proxy inverso, che inoltra le richieste a Gunicorn per l’elaborazione.

La creazione di applicazioni su Docker consente di costruire facilmente un’infrastruttura in base alle impostazioni di configurazione specificate tramite Docker Compose. L’infrastruttura può essere definita in un singolo file e creata con un singolo comando. In questo passaggio, imposteremo il file `docker-compose.yml` per eseguire l’applicazione Flask.

All’interno del file `docker-compose.yml` è possibile definire l’infrastruttura dell’applicazione come singoli servizi; questi possono essere collegati tra loro e ognuno può avere un volume a esso collegato per l’archiviazione persistente. I volumi sono memorizzati in una parte del file system host gestito da Docker (all’interno della cartella `/var/lib/docker/volumes/` su sistemi Linux).

Per iniziare, creiamo una directory per l’applicazione nella *home directory* del sistema host, definita `myapp`; al suo interno, creiamo un file `docker-compose.yml`, il quale inizia con un numero che ne identifica la versione; in questo caso, viene fatto riferimento alla versione `3`, che

corrisponde alla versione *1.13.0+* del Docker Engine. Andiamo dunque a definire i servizi che costituiscono la nostra architettura, e iniziamo definendo il servizio relativo all'applicazione Flask.

Listato 8.24 docker-compose.yml.

```
version: '3'
services:
  flask:
    build:
      context: app
      dockerfile: Dockerfile
    container_name: flask
    image: digitalocean.com/flask-python:3.6
    restart: unless-stopped
    environment:
      APP_ENV: "prod"
      APP_DEBUG: "False"
      APP_PORT: 5000
      MONGODB_DATABASE: flaskdb
      MONGODB_USERNAME: flaskuser
      MONGODB_PASSWORD: your_mongodb_password
      MONGODB_HOSTNAME: mongodb
    volumes:
      - appdata:/var/www
    depends_on:
      - mongodb
    networks:
      - frontend
      - backend
...
...
```

La proprietà `build` definisce il contesto della stessa; in questo caso, la cartella `app` che conterrà il Dockerfile.

Utilizziamo la proprietà `container_name` per definire un nome per ciascun container, mentre la proprietà `image` specifica il nome dell'immagine e come verrà contrassegnata. La proprietà `restart` definisce il modo in cui il container deve essere riavviato, e nel nostro caso questo avviene a meno che il container non venga arrestato; ciò significa che i container verranno arrestati solo quando Docker Engine viene arrestato o riavviato o quando si arrestano esplicitamente i container. Il vantaggio dell'utilizzo di questa tipologia di proprietà è che i container si avvieranno automaticamente una volta riavviato il Docker Engine o se si dovesse verificare un errore.

La proprietà `environment` contiene le variabili d'ambiente che vengono passate al container; è necessario fornire una password sicura per la variabile d'ambiente `MONGODB_PASSWORD`. La proprietà `volumes` definisce i

volumi che il servizio sta utilizzando; nel caso dell'esempio il volume `appdata` è montato all'interno del container nella directory `/var/www`.

La proprietà `depends_on` definisce il servizio dal quale Flask dipende per funzionare correttamente; ovviamente l'applicazione dipenderà da *MongoDB*, poiché il servizio funge da database per il sistema; in questo modo, si garantisce che il servizio dell'applicazione web sia in esecuzione solo se il servizio MongoDB è in esecuzione.

La proprietà `networks` definisce due reti, `frontend` e `backend`: quelle alle quali avrà accesso il servizio Flask.

Con la definizione di quest'ultimo servizio, siamo pronti per aggiungere la configurazione MongoDB al file. In questo esempio, verrà utilizzata l'immagine MongoDB della versione 4.0.8 ufficiale.

Listato 8.25 docker-compose.yml.

```
...
mongodb:
    image: mongo:4.0.8
    container_name: mongodb
    restart: unless-stopped
    command: mongod --auth
    environment:
        MONGO_INITDB_ROOT_USERNAME: mongodbuser
        MONGO_INITDB_ROOT_PASSWORD: your_mongodb_root_password
        MONGO_INITDB_DATABASE: flaskdb
        MONGODB_DATA_DIR: /data/db
        MONDODB_LOG_DIR: /dev/null
    volumes:
        - mongodata:/data/db
    networks:
        - backend
...

```

In questo caso, come proprietà `container_name` per questo servizio abbiamo impostato `mongodb`, con una politica di riavvio a meno che non il container non venga arrestato (così come in precedenza). Utilizziamo la proprietà `command` per definire il comando che verrà eseguito all'avvio del container; il comando `mongod --auth` disabilita l'accesso alla shell MongoDB senza credenziali, il che protegge il database, richiedendo l'autenticazione.

Le variabili d'ambiente `MONGO_INITDB_ROOT_USERNAME` e `MONGO_INITDB_ROOT_PASSWORD` creano un utente *root* con le credenziali specificate, quindi dobbiamo assicurarci di sostituire i valori inseriti come segnaposto nel caso di esempio con il reale nome utente e con la relativa password.

Per impostazione predefinita MongoDB memorizza i suoi dati nella cartella `/data/db`, quindi i dati nella cartella verranno scritti nel volume denominato `mongodbdata`, che ne garantisce la persistenza, così da non perdere le informazioni in caso di riavvio. Il servizio mongoDB non espone alcuna porta, quindi il servizio sarà accessibile solo attraverso la rete di backend.

Successivamente, definiremo il server per l'applicazione; aggiungiamo quindi il seguente codice al file `docker-compose.yml` per configurare Nginx.

Listato 8.26 docker-compose.yml.

```
...
webserver:
    build:
        context: nginx
        dockerfile: Dockerfile
    image: digitalocean.com/webserver:latest
    container_name: webserver
    restart: unless-stopped
    environment:
        APP_ENV: "prod"
        APP_NAME: "webserver"
        APP_DEBUG: "false"
        SERVICE_NAME: "webserver"
    ports:
        - "80:80"
        - "443:443"
    volumes:
        - nginxdata:/var/log/nginx
    depends_on:
        - flask
    networks:
        - frontend
...
...
```

Qui definiamo sempre il contesto della build, che è la cartella `nginx` contenente il Dockerfile; con la proprietà `image`, specifichiamo l'immagine utilizzata per contrassegnare ed eseguire il container, ossia l'immagine `nginx`. La proprietà `ports` configura il servizio Nginx in modo che sia accessibile pubblicamente tramite le porte 80 e 443 e il volume montato sarà `nginxdata`, all'interno del container nella directory `/var/log/nginx`.

Nella prima fase del progetto abbiamo configurato il servizio Flask come dipendente dal database; in questo caso, impostiamo un legame di dipendenza con l'applicazione web. Infine, la proprietà `networks` definisce in che modo verrà gestita la configurazione della rete che darà accesso all'applicazione.

Creeremo una rete bridge, per consentire ai container di comunicare tra loro, e questo può essere fatto aggiungendo le seguenti righe alla fine del file.

Listato 8.27 docker-compose.yml.

```
...  
  networks:  
    frontend:  
      driver: bridge  
    backend:  
      driver: bridge  
...  
...
```

Abbiamo definito due reti, `frontend` e `backend`, per i servizi ai quali i servizi dovranno connettersi. I servizi di front-end, come Nginx, si collegheranno alla rete `frontend`, poiché devono essere accessibili pubblicamente. I servizi di back-end, come MongoDB, si collegheranno alla rete `backend`, per impedire l'accesso non autorizzato al servizio.

Come ultimo passo, definiamo i volumi che permetteranno di rendere persistenti il database, l'applicazione e i file di configurazione; dal momento che l'applicazione utilizzerà il database e diversi file, è indispensabile mantenere le modifiche apportate. I volumi sono gestiti da Docker e disponibili nel file system.

Aggiungiamo queste ultime righe di codice al file `docker-compose.yml` per configurare i volumi.

Listato 8.28 docker-compose.yml.

```
...  
  volumes:  
    mongodata:  
      driver: local  
    appdata:  
      driver: local  
    nginxdata:  
      driver: local  
...
```

La sezione `volumes` dichiara i volumi che l'applicazione utilizzerà per conservare i dati. Qui sono stati definiti i volumi `mongodata`, `appdata` e `nginxdata` per conservare i dati del database MongoDB, i dati dell'applicazione Flask e i log del server Nginx. Tutti questi volumi utilizzano un driver locale per archiviare i dati localmente (vedi il Capitolo 7). I volumi vengono utilizzati per conservare questi dati in

modo tale che dati come quelli dei database MongoDB e i log del server web Nginx non vadano perduti dopo il riavvio dei container.

Il file `docker-compose.yml` dovrebbe dunque avere, complessivamente, il seguente aspetto.

Listato 8.29 docker-compose.yml.

```
...
version: '3'
services:
  flask:
    build:
      context: app
      dockerfile: Dockerfile
      container_name: flask
    image: digitalocean.com/flask-python:3.6
    restart: unless-stopped
    environment:
      APP_ENV: "prod"
      APP_DEBUG: "False"
      APP_PORT: 5000
      MONGODB_DATABASE: flaskdb
      MONGODB_USERNAME: flaskuser
      MONGODB_PASSWORD: your_mongodb_password
      MONGODB_HOSTNAME: mongodb
    volumes:
      - appdata:/var/www
    depends_on:
      - mongodb
  networks:
    - frontend
    - backend
  mongodb:
    image: mongo:4.0.8
    container_name: mongodb
    restart: unless-stopped
    command: mongod --auth
    environment:
      MONGO_INITDB_ROOT_USERNAME: mongodbuser
      MONGO_INITDB_ROOT_PASSWORD: your_mongodb_root_password
      MONGO_INITDB_DATABASE: flaskdb
      MONGODB_DATA_DIR: /data/db
      MONDODB_LOG_DIR: /dev/null
    volumes:
      - mongodbdata:/data/db
    networks:
      - backend
  webserver:
    build:
      context: nginx
      dockerfile: Dockerfile
    image: digitalocean.com/webserver:latest
    container_name: webserver
    restart: unless-stopped
    environment:
      APP_ENV: "prod"
      APP_NAME: "webserver"
      APP_DEBUG: "false"
```

```

        SERVICE_NAME: "webserver"
ports:
  - "80:80"
  - "443:443"
volumes:
  - nginxdata:/var/log/nginx
depends_on:
  - flask
networks:
  - frontend
networks:
  frontend:
    driver: bridge
  backend:
    driver: bridge
volumes:
  mongodata:
    driver: local
  appdata:
    driver: local
  nginxdata:
    driver: local

```

Con Docker, è possibile creare container per eseguire le applicazioni tramite il Dockerfile.

In questo passaggio, scriveremo i Dockerfile per i servizi Flask e per il web server Nginx. Per iniziare, creiamo una directory `app` per l'applicazione, al cui interno definiremo il seguente Dockerfile.

Listato 8.30 myapp/app/Dockerfile.

```

FROM python:3.6.8-alpine3.9
LABEL MAINTAINER="Serena Sensini mail@mail.com"
ENV GROUP_ID=1000 \
    USER_ID=1000
WORKDIR /var/www
...
```

In questo Dockerfile stiamo creando un'immagine a partire dall'immagine base denominata `python:3.6.8-alpine3.9`, che si basa su Alpine 3.9 con Python 3.6.8 preinstallato.

La direttiva `ENV` definisce le variabili d'ambiente per il nostro gruppo e ID utente; Linux Standard Base (LSB) specifica che UID e GID con valori compresi tra 0 e 99 sono allocati staticamente dal sistema. I valori nell'intervallo 100 e 999 dovrebbero essere allocati dinamicamente per utenti e gruppi di sistema, mentre i valori tra 1000 e 59999 dovrebbero essere allocati dinamicamente per gli account utente. Tenendo presente questo, possiamo assegnare in sicurezza un UID e un GID di `1000`, aggiornando `GROUP_ID` e `USER_ID` per soddisfare qualunque tipo di requisito (passaggio non obbligatorio, ma a volte utile).

*****ebook converter DEMO

Watermarks*****

La direttiva `WORKDIR` definisce la directory di lavoro per il container, che in questo caso sarà `/var/www`. Il campo `LABEL MAINTAINER` andrà modificato con il nome e l'indirizzo email dello sviluppatore, anche se non è obbligatorio specificarlo.

Aggiungiamo il seguente blocco di codice per copiare l'applicazione Flask nel container e per installare le dipendenze necessarie.

Listato 8.30 myapp/app/Dockerfile.

```
...
ADD ./requirements.txt /var/www/requirements.txt
RUN pip install -r requirements.txt
ADD . /var/www/
RUN pip install gunicorn
```

Il codice riportato utilizzerà l'istruzione `ADD` per copiare i file dalla directory dell'app locale nella directory `/var/www` direttamente nel container. Successivamente, il Dockerfile utilizzerà la direttiva `RUN` per installare Gunicorn e i pacchetti specificati nel file `requirements.txt`, che vedremo più avanti.

Il seguente blocco di codice aggiunge invece un nuovo utente e ne definisce il gruppo, per poi inizializzare l'applicazione.

Listato 8.31 myapp/app/Dockerfile.

```
...
RUN addgroup -g $GROUP_ID www
RUN adduser -D -u $USER_ID -G www www -s /bin/sh
USER www
EXPOSE 5000

CMD [ "gunicorn", "-w", "4", "--bind", "0.0.0.0:5000", "wsgi"]
```

Per impostazione predefinita, i container Docker vengono eseguiti con i permessi dell'utente `root`; questo utente sappiamo che ha accesso a tutto il sistema, quindi le implicazioni di una violazione della sicurezza possono essere disastrose. Per mitigare questo rischio per la sicurezza, le istruzioni riportate creeranno un nuovo utente, col relativo gruppo, che avrà accesso solo alla directory `/var/www`.

Questo codice utilizzerà prima il comando `addgroup` per creare un nuovo gruppo chiamato `www`. Il parametro `-g` imposterà l'ID del gruppo sulla variabile `ENV GROUP_ID` impostando il valore a `1000`, come specificato in precedenza nel Dockerfile.

La seconda riga crea un utente `www` con un ID utente pari a `1000`, come definito dalla variabile `ENV`; l'opzione `-s` crea la directory dell'utente (se non esiste) e imposta la shell di accesso predefinita su `/bin/sh`. Il parametro `-G` viene utilizzato per impostare il gruppo di accesso iniziale dell'utente su `www`, creato nel comando precedente.

L'istruzione `USER` definisce che i programmi eseguiti nel container useranno come impostazione predefinita l'utente `www`; Gunicorn sarà in ascolto sulla porta `5000`, quindi dovremo rendere accessibile questa porta tramite la direttiva `EXPOSE`.

Infine, la riga `CMD ["gunicorn", "-w", "4", "--bind", "0.0.0.0:5000", "wsgi"]` esegue il comando per avviare il server Gunicorn con quattro operatori in ascolto su porta `5000`; il numero dovrebbe generalmente essere compreso tra `2` e `4` per core nel server, secondo quanto riportato all'interno della documentazione di Gunicorn.

Il Dockerfile, una volta completato sarà simile al seguente.

Listato 8.32 myapp/app/Dockerfile.

```
FROM python:3.6.8-alpine3.9
LABEL MAINTAINER="Serena Sensini mail@mail.com"
ENV GROUP_ID=1000 \
    USER_ID=1000
WORKDIR /var/www
ADD ./requirements.txt /var/www/requirements.txt
RUN pip install -r requirements.txt
ADD . /var/www/
RUN pip install gunicorn
RUN addgroup -g $GROUP_ID www
RUN adduser -D -u $USER_ID -G www www -s /bin/sh
USER www
EXPOSE 5000

CMD [ "gunicorn", "-w", "4", "--bind", "0.0.0.0:5000", "wsgi"]
```

Creiamo ora un'altra cartella, denominata `nginx`, dove configureremo il Dockerfile relativo.

Listato 8.33 myapp/nginx/Dockerfile.

```
FROM alpine:latest
LABEL MAINTAINER="Serena Sensini mail@mail.it"
RUN apk --update add nginx && \
    ln -sf /dev/stdout /var/log/nginx/access.log && \
    ln -sf /dev/stderr /var/log/nginx/error.log && \
    mkdir /etc/nginx/sites-enabled/ && \
    mkdir -p /run/nginx && \
    rm -rf /etc/nginx/conf.d/default.conf && \
    rm -rf /var/cache/apk/*
COPY conf.d/app.conf /etc/nginx/conf.d/app.conf
EXPOSE 80 443
```

```
CMD ["nginx", "-g", "daemon off;"]
```

Questo Dockerfile utilizza come immagine di base `alpine`, che abbiamo utilizzato spesso perché è una piccola distribuzione Linux con una superficie di attacco minima che ne garantisce la sicurezza.

Nella direttiva `RUN` viene installato `Nginx` e vengono creati dei collegamenti simbolici per poter accedere ai log del server e a quelli di errore tramite le cartelle `/dev/stderr` e `/dev/stdout`.

Al termine, vengono eseguiti i comandi per rimuovere `default.conf` e tutti i file presenti nella cartella `/var/cache/apk/`, per ridurre le dimensioni dell'immagine finale.

La direttiva `COPY` copia all'interno del container la configurazione del server web `app.conf`. La direttiva `EXPOSE` garantisce che i container siano in ascolto sulle porte 80 e 443, poiché l'applicazione verrà eseguita solo su queste porte.

Infine, la direttiva `CMD` definisce il comando per avviare il server Nginx. Ora che Dockerfile è pronto, possiamo configurare il proxy inverso Nginx per instradare il traffico all'applicazione Flask.

Nel seguente passaggio, configureremo Nginx come proxy inverso per inoltrare richieste a Gunicorn sulla porta 5000. Un *server proxy inverso* viene utilizzato per indirizzare le richieste client al server back-end appropriato, e questo fornisce un ulteriore livello di astrazione e di controllo per garantire un flusso regolare del traffico di rete tra client e server.

Creiamo ora una sottocartella della cartella `nginx` denominata `conf.d`, all'interno della quale predisporremo la configurazione del server nel file `app.conf`.

Listato 8.34 myapp/nginx/conf.d/app.conf.

```
upstream app_server {
    server flask:5000;
}
server {
    listen 80;
    server_name _;
    error_log /var/log/nginx/error.log;
    access_log /var/log/nginx/access.log;
    client_max_body_size 64M;
    location / {
        try_files $uri @proxy_to_app;
    }
    location @proxy_to_app {
        gzip_static on;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

```

        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header Host $http_host;
        proxy_buffering off;
        proxy_redirect off;
        proxy_pass http://app_server;
    }
}

```

Ciò definirà innanzitutto il server cosiddetto di *upstream*, che viene comunemente utilizzato per specificare un server web o per il routing di un'applicazione o per il bilanciamento del carico.

Il server upstream `app_server` definisce l'indirizzo del server con la direttiva `server`, identificata dal nome del container Flask sulla porta 5000; vuol dire che si aspetterà di ricevere delle richieste per l'applicazione Flask sulla porta specificata.

La configurazione per il server web Nginx è definita nel blocco `server`: la direttiva `listen` definisce il numero di porta su cui il server ascolterà le richieste in arrivo. Le direttive `error_log` e `access_log` definiscono i file per la scrittura dei log raccolti. La direttiva `proxy_pass` viene utilizzata per impostare il server *upstream* per l'inoltro delle richieste all'indirizzo `http://app_server`.

Ultimata la configurazione del server Nginx, è possibile passare alla creazione dell'API in Flask. Torniamo nella cartella `app` e definiamo un file `requirements.txt`, dove specificheremo tutte le librerie che devono essere installate affinché l'applicazione funzioni correttamente, ossia `Flask`, `PyMongo` per la connessione al database Mongo e `requests` per eseguire chiamate HTTP.

Listato 8.35 myapp/app/requirements.txt.

```

Flask==1.0.2
Flask-PyMongo==2.2.0

requests==2.20.1

```

All'interno della stessa cartella, creiamo un file `app.py` nel quale inseriremo la logica dell'applicazione; come prima cosa, importiamo le librerie precedentemente specificate per l'installazione.

Listato 8.36 myapp/app/app.py.

```

import os
from flask import Flask, request, jsonify
from flask_pymongo import PyMongo

...

```

La libreria `os` viene utilizzata per importare le variabili d’ambiente; dalla libreria `flask` importiamo invece gli oggetti `Flask`, `request` e `jsonify` per, rispettivamente, creare un’istanza dell’applicazione, gestire le richieste e inviare le risposte JSON. Dalla libreria `flask_pymongo` importeremo infine l’oggetto `PyMongo`, per interagire con il database MongoDB.

Successivamente, aggiungiamo il codice necessario per creare un collegamento al database.

Listato 8.37 myapp/app/app.py.

```
...
application = Flask(__name__)
application.config["MONGO_URI"] = 'mongodb://'+ os.environ['MONGODB_USERNAME'] +
':'+ os.environ['MONGODB_PASSWORD'] + '@' + os.environ['MONGODB_HOSTNAME'] +
':27017/' + os.environ['MONGODB_DATABASE']
mongo = PyMongo(application)
db = mongo.db
...
```

L’applicazione Flask carica l’oggetto dell’applicazione nella variabile `application`. Successivamente, viene creata la stringa di connessione `MongoDB` tramite le variabili d’ambiente utilizzando `os.environ`, che ne permette il recupero tramite la definizione fatta prima all’interno del file `docker-compose.yml`. L’oggetto `application` viene passato al metodo `PyMongo()`, il quale restituirà un oggetto contenuto all’interno della variabile `mongo`, che a sua volta contiene l’oggetto `db` a partire da `mongo.db`.

Vediamo ora la configurazione per un breve messaggio di benvenuto che comparirà quando l’utente si collegherà all’indirizzo esposto dal server.

Listato 8.38 myapp/app/app.py.

```
...
@application.route('/')
def index():
    return jsonify(
        status=True,
        message='Benvenuto in questa applicazione realizzata con Flask e MongoDB!'
    )
...
```

L’annotazione `@application.route` definisce l’indirizzo relativo dell’API alla quale il server sarà in ascolto e potrà rispondere, tramite una richiesta HTTP in GET, con il messaggio che abbiamo definito.

Definiamo ora anche un’altra rotta alla quale il nostro server risponderà con una lista delle attività da terminare, recuperate direttamente dal

database MongoDB.

Listato 8.39 myapp/app/app.py.

```
...
@application.route('/todo')
def todo():
    _todos = db.todo.find()
    item = {}
    data = []
    for todo in _todos:
        item = {
            'id': str(todo['_id']),
            'todo': todo['todo']
        }
        data.append(item)
    return jsonify(
        status=True,
        data=data
    )
...
```

L'annotazione `@application.route('/todo')` definisce il percorso `/todo` dell'API, che restituisce le cose da fare contenute nel database. Il metodo `db.todo.find()` effettuerà la ricerca al suo interno e restituirà i risultati recuperati; nel ciclo `for` costruiremo gli oggetti JSON che verranno restituiti alla fine della richiesta.

Al fine di permettere all'utente di inserire le attività da svolgere, creiamo un altro punto di accesso, tramite il metodo `POST`.

Listato 8.40 myapp/app/app.py.

```
...
@application.route('/todo', methods=['POST'])
def createTodo():
    data = request.get_json(force=True)
    item = {
        'todo': data['todo']
    }
    db.todo.insert_one(item)
    return jsonify(
        status=True,
        message='Attività salvata con successo!'
    ), 201
...
```

Il metodo `request.get_json(force=True)` ottiene il JSON allegato alla `request` e i dati vengono utilizzati per creare il JSON che verrà salvato nel database tra le attività; la funzione `db.todo.insert_one(item)` viene utilizzata per inserire un elemento nel database. Dopo aver salvato le attività nel database, viene restituita una risposta JSON con un codice di stato `201` (è

possibile visualizzare il codice della risposta HTTP tramite la gestione della rete di un qualsiasi browser).

Infine, aggiungiamo il codice per eseguire l'applicazione.

Listato 8.41 myapp/app/app.py.

```
...  
if __name__ == "__main__":  
    ENVIRONMENT_DEBUG = os.environ.get("APP_DEBUG", True)  
    ENVIRONMENT_PORT = os.environ.get("APP_PORT", 5000)  
  
    application.run(host='0.0.0.0', port=ENVIRONMENT_PORT, debug=ENVIRONMENT_DEBUG)
```

In quest'ultimo passaggio, impostiamo le diverse variabili d'ambiente dell'applicazione tramite quelle definite all'interno del file `docker-compose.yml`, che vengono recuperate per mezzo del metodo `os.environ.get()`.

A questo punto il file `app.py` dovrebbe avere l'aspetto mostrato di seguito.

Listato 8.42 myapp/app/app.py.

```
import os  
from flask import Flask, request, jsonify  
from flask_pymongo import PyMongo  
application = Flask(__name__)  
application.config["MONGO_URI"] = 'mongodb://{}:{}@{}:{}/{}'.format(  
    os.environ['MONGODB_USERNAME'],  
    os.environ['MONGODB_PASSWORD'],  
    os.environ['MONGODB_HOSTNAME'],  
    os.environ['MONGODB_DATABASE'])  
mongo = PyMongo(application)  
db = mongo.db  
@application.route('/')  
def index():  
    return jsonify(  
        status=True,  
        message='Benvenuto in questa applicazione realizzata con Flask e MongoDB!')  
@application.route('/todo')  
def todo():  
    todos = db.todo.find()  
    item = {}  
    data = []  
    for todo in todos:  
        item = {  
            'id': str(todo['_id']),  
            'todo': todo['todo']}  
        data.append(item)  
    return jsonify(  
        status=True,  
        data=data)  
@application.route('/todo', methods=['POST'])  
def createTodo():  
    data = request.get_json(force=True)  
    item = {  
        'todo': data['todo']}
```

```

        db.todo.insert_one(item)
        return jsonify(
            status=True,
            message='Attività salvata con successo!'
        ), 201
    if __name__ == "__main__":
        ENVIRONMENT_DEBUG = os.environ.get("APP_DEBUG", True)
        ENVIRONMENT_PORT = os.environ.get("APP_PORT", 5000)

        application.run(host='0.0.0.0', port=ENVIRONMENT_PORT, debug=ENVIRONMENT_DEBUG)

```

L'ultimo passaggio da effettuare consiste nel creare il file `wsgi.py`, il quale crea un oggetto in modo che il server possa utilizzarlo; ogni volta che arriva una richiesta, il server utilizza questo oggetto per lanciare i gestori di richieste dell'applicazione dopo aver analizzato l'URL.

Inseriamo quindi il seguente codice nel file `wsgi.py`; questo file importa l'oggetto `application` dal file `app.py` creato in precedenza e crea un oggetto per il server Gunicorn.

Listato 8.43 myapp/app/wsgi.py.

```

from app import application
if __name__ == "__main__":
    application.run()

```

Ora che abbiamo definito tutti i servizi nel file `docker-compose.yml` e le loro configurazioni, possiamo avviare i container.

Poiché i servizi sono definiti in un singolo file, è necessario emettere un singolo comando per avviare i container, creare i volumi e configurare le reti. Questo comando crea anche l'immagine per l'applicazione Flask e il server web Nginx; si tratta del comando `docker-compose up`; quando si esegue il comando per la prima volta, verranno scaricate tutte le immagini Docker necessarie, il che può richiedere tempo. Una volta che le immagini sono state scaricate e memorizzate nel computer, Docker configurerà e avvierà i diversi container. Può essere aggiunta l'opzione `-d` per “daemonizzare” il processo, il che gli consente di essere eseguito in background.

Se eseguiamo il comando `docker ps` per elencare i container in esecuzione al termine del processo di compilazione, dovremmo avere un risultato simile al seguente.

COMMAND	CREATED	PORTS	N
AMES			
"nginx -g ..." 2 weeks ago >443/tcp	2 weeks ago	0.0.0.0:80->80/tcp, 0.0.0.0:443-	webserver
"gunicorn -w ..." ago 5000/tcp	2 weeks ago	flask	

*****ebook converter DEMO

Watermarks*****

Per impostazione predefinita, MongoDB consente agli utenti di accedere senza credenziali e concede privilegi illimitati. In questo passaggio, configureremo il database MongoDB creando un utente dedicato per accedervi.

Per fare ciò, avremo bisogno del nome utente e della password di root impostati nelle variabili d'ambiente del file `docker-compose.yml` come `MONGO_INITDB_ROOT_USERNAME` e `MONGO_INITDB_ROOT_PASSWORD` per il servizio analogo. In generale, è meglio evitare di utilizzare l'account amministrativo di `root` quando si interagisce con il database; invece, creeremo un utente di database dedicato per l'applicazione Flask, nonché un nuovo database al quale sarà consentita l'accesso.

Per creare un nuovo utente, dobbiamo prima avviare una shell interattiva sul container `mongodb`, tramite il seguente comando.

Listato 8.44 Avvio del terminale del container con alias `mongodb`.

```
root@vbox:/home# docker exec -it mongodb bash
```

Una volta all'interno del container, accediamo all'account amministrativo principale di MongoDB tramite il seguente comando.

Listato 8.45 Accesso al database MongoDB.

```
root@378bf621:/# mongo -u mongodbuser -p
```

Verrà richiesta la password impostata come valore per la variabile `MONGO_INITDB_ROOT_PASSWORD` nel file `docker-compose.yml`. La password può essere cambiata impostando un nuovo valore per `MONGO_INITDB_ROOT_PASSWORD` nel servizio `mongodb`, nel qual caso sarà necessario rieseguire il comando `docker-compose up`.

Eseguiamo il comando per mostrare i database presenti all'interno dell'istanza MongoDB, come di seguito riportato.

Listato 8.46 Elenco dei database.

```
mongodb> show dbs;
admin      0.000GB
config     0.000GB
local      0.000GB
```

`admin` è un database speciale, che concede autorizzazioni amministrative agli utenti; se un utente ha accesso in lettura al database di amministrazione, avrà le autorizzazioni di lettura e scrittura per tutti gli altri database. Poiché l'output elenca il database `admin`, l'utente ha accesso

a questo database e può quindi leggere e scrivere su tutti gli altri database.

La creazione della prima attività nell'applicazione creerà automaticamente il database MongoDB; infatti, è possibile passare anche a un database che non esiste, usando il comando `use database`; in questo caso, il database viene creato quando viene salvato un documento in una *collection*. Pertanto, il database non verrà creato qui; ciò accadrà quando l'utente inserirà la prima attività nella sua lista di cose da fare, sfruttando le API dell'applicazione.

Eseguiamo il comando `use` per passare al database `flaskdb`, dove salveremo queste attività, e creiamo quindi un nuovo utente che avrà l'accesso a questo database.

Listato 8.47 Creazione database flaskdb.

```
mongodb> use flaskdb;
mongodb> db.createUser({user: 'user', pwd: 'password', roles: [{role: 'readWrite', db: 'flaskdb'}]})

mongodb> exit;
```

Questo comando crea un utente chiamato `user`, con accesso in lettura e scrittura al database `flaskdb`. L'utente e la password qui sono i valori definiti nel file `docker-compose.yml` nella sezione delle variabili d'ambiente per il servizio di Flask, quindi utilizzeremo i valori lì inseriti.

Per verificare che la creazione sia andata a buon fine, proviamo a effettuare l'autenticazione.

Listato 8.48 Autenticazione dell'utente appena creato.

```
root@378bf621:/# mongo -u user -p password --authenticationDatabase flaskdb
```

Se l'accesso va a buon fine, vuol dire che l'utente è stato creato correttamente.

A questo punto, non ci resta che accedere all'applicazione Flask e sottoporre a test le API per l'inserimento e la lettura delle attività da svolgere, e possiamo farlo tramite un qualsiasi programma per chiamate REST, come Postman o tramite il comando `curl`, come nel seguente esempio.

Listato 8.49 Il comando curl.

```
root@vbox:/home# curl -i http://localhost
```

Il risultato atteso dovrebbe essere il seguente.

```
{"message":"Benvenuto in questa applicazione realizzata con Flask e MongoDBWelcome  
to the Dockerized Flask MongoDB app!","status":true}
```

Per sottoporre a test tutto, creiamo una nuova attività usando le API di Flask e effettuando una richiesta `POST` al path `/todo`.

Listato 8.50 Il comando curl.

```
root@vbox:/home# curl -i -H "Content-Type: application/json" -X POST -d '{"todo":  
"Studiare Docker Compose"}' http://localhost/todo
```

Se il messaggio ricevuto è il seguente, la creazione è andata a buon fine.

```
{"message": " = 'Attività salvata con successo! ", "status":true}
```

Con questo esercizio, abbiamo “dockerizzato” una semplice API in Flask che sfrutta un database MongoDB con Nginx come proxy inverso distribuito sul server. Per un ambiente di produzione è possibile utilizzare `sudo systemctl enable docker` per assicurarsi che il servizio Docker si avvii automaticamente in fase di esecuzione dei container.

Che cosa abbiamo imparato

- Quali sono gli strumenti che abbiamo a disposizione per gestire più di un container, le sue proprietà e la sua esecuzione.
- Che cosa vuol dire “orchestrare” dei container.
- Che cos’è Docker Swarm e che cosa rappresenta uno sciame.
- Che cosa sono i nodi e che tipologie esistono.
- Come creare uno sciame e associargli dei nodi.
- Come avviare un servizio e che cosa sono le repliche.
- Come ispezionare un servizio e come variarne la scala.
- Come gestire il ciclo di vita di un servizio.
- Che cos’è Docker Compose e come installarlo.
- In cosa consiste la potenza del file `docker-compose.yml` e quali attributi è possibile utilizzare per definire più servizi che cooperino tra loro.
- Come creare un’applicazione Flask (Python) con un database MongoDB sfruttando Docker Compose.

Esempi pratici

In questo capitolo vedremo una serie di esempi di pratici di come utilizzare Docker con diverse tecnologie; alcuni passaggi tra un esempio e l'altro potrebbero ripetersi, ma l'obiettivo è quello di avere dei progetti minimi ma completi da cui trarre spunto per svolgere altre attività. Alcuni degli esempi presenteranno anche parti di codice in vari linguaggi di programmazione; questi verranno spiegati nel dettaglio, e alla fine di ogni paragrafo ritroveremo il codice completo.

Lo scopo di questi paragrafi è quello di vedere come utilizzare in pratica Docker, Docker Machine e Docker Compose, così da coprire i vari aspetti finora solo accennati.

Applicazione Laravel, Nginx e MySQL

Quando si utilizza uno stack di *applicazioni LEMP*, per esempio, con PHP, Nginx, MySQL e il framework Laravel, Docker può semplificare notevolmente il processo di installazione.

Docker Compose ha ulteriormente semplificato il processo di sviluppo, consentendo agli sviluppatori di definire la propria infrastruttura, inclusi i servizi applicativi, le reti e i volumi, in un unico file. Offre, infatti, un'alternativa efficiente all'esecuzione di più comandi Docker di creazione ed esecuzione dei container.

In questo progetto, creeremo un'applicazione web usando il framework Laravel, con Nginx come server web e MySQL come database, tutto all'interno di container Docker. Definiremo l'intera configurazione dello

stack tramite il file `docker-compose.yml`, insieme ai file di configurazione per PHP, MySQL e Nginx.

Step 1 - Installazione e configurazione di Laravel

Come primo passo, dobbiamo scaricare l'ultima versione di Laravel e installare le dipendenze del progetto, incluso Composer, il gestore di pacchetti a livello di applicazione per PHP. Installeremo queste dipendenze tramite Docker, per evitare di dover installare Composer a livello globale.

Innanzitutto, controlliamo di essere in una cartella nota e cloniamo l'ultima versione di Laravel a partire dal suo repository Github in una nuova directory: `laravel-app`.

Listato 9.1 Clonazione del repo Laravel.

```
$ git clone https://github.com/laravel/laravel.git laravel-app
```

Dopo che la clonazione sarà terminata, ci spostiamo all'interno della cartella `laravel-app` (tramite il comando `cd`) e utilizziamo Composer per predisporre il progetto Laravel, lanciando il comando presentato nel Listato 9.2; l'uso dei parametri `-v` e `--rm` crea un container effimero, che verrà associato alla directory corrente prima di essere rimosso. Questo andrà a copiare il contenuto della directory in cui abbiamo clonato il progetto Laravel (il comando `pwd` restituisce infatti la posizione corrente) nella cartella `app` del container.

Listato 9.2 Configurazione del progetto tramite Composer.

```
$ docker run --rm -v $(pwd):/app composer install
```

Come passaggio finale, impostiamo le autorizzazioni per la cartella del progetto, in modo che sia di proprietà di un utente che non abbia permessi da amministratore; questo si rivelerà importante quando scriveremo il Dockerfile per l'immagine dell'applicazione nello Step 4, poiché consentirà di modificare il codice ed eseguire i processi nel proprio container come utenti non *root*.

Step 2 - Configurazione di Docker Compose

La creazione di applicazioni con Docker Compose semplifica il processo di configurazione e controllo delle versioni dell'infrastruttura. Per configurare la nostra applicazione Laravel, scriveremo un file `docker-compose.yml` che definisce il server web, il database e il servizio relativo all'applicazione.

Apriamo dunque il file `docker-compose.yml` tramite un qualunque editor di testi e definiamo i tre servizi, chiamati rispettivamente `application`, `webserver` e `db`.

Listato 9.3 Il file `./laravel-app/docker-compose.yml`.

```
version: '3'
services:
  application:
    build:
      context: .
      dockerfile: Dockerfile
    image: php
    container_name: app
    restart: unless-stopped
    tty: true
    environment:
      SERVICE_NAME: app
      SERVICE_TAGS: dev
    working_dir: /var/www
    volumes:
      - ./:/var/www
      - ./php/local.ini:/usr/local/etc/php/conf.d/local.ini
    networks:
      - app-network
  webserver:
    image: nginx:alpine
    container_name: webserver
    restart: unless-stopped
    tty: true
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./:/var/www
      - ./nginx/conf.d/:/etc/nginx/conf.d/
    networks:
      - app-network
  db:
    image: mysql:5.7.22
    container_name: db
    restart: unless-stopped
    tty: true
    ports:
      - "3306:3306"
```

```

environment:
  MYSQL_DATABASE: laravel
  MYSQL_ROOT_PASSWORD: your_mysql_root_password
  SERVICE_TAGS: dev
  SERVICE_NAME: mysql
volumes:
  - dbdata:/var/lib/mysql/
  - ./mysql/my.cnf:/etc/mysql/my.cnf
networks:
  - app-network
networks:
  app-network:
    driver: bridge
volumes:
  dbdata:
    driver: local

```

Dopo aver aggiunto il seguente codice al file `docker-compose.yml`, assicuriamoci di sostituire il valore della password per `MYSQL_ROOT_PASSWORD`, definita come variabile d'ambiente nel servizio `db`, con una password complessa a scelta.

Ecco i servizi qui definiti.

- `application`: la definizione del servizio conterrà l'applicazione Laravel ed eseguirà l'immagine Docker `php`, che verrà definita nello Step 4. Impostiamo inoltre come cartella di lavoro `/var/www`.
- `webserver`: in questo caso estraiamo l'immagine `nginx:alpine` tramite Docker Hub ed esponiamo le porte 80 e 443.
- `db`: estraiamo l'immagine `mysql:5.7.22` tramite Docker Hub e definiamo alcune variabili d'ambiente necessarie alla configurazione del database, tra cui un la creazione di un database chiamato `laravel` per l'applicazione e la *password di root* per il database. Viene anche mappata la porta 3306 del container con la stessa porta dell'host.

Ogni proprietà `container_name` definisce un nome per il container che verrà utilizzato come alias e che corrisponde al nome del servizio.

Per facilitare la comunicazione tra container, i servizi sono collegati a una rete bridge chiamata `app-network`. Una rete bridge, come abbiamo visto in precedenza, utilizza un software di *bridging* che consente ai container collegati alla stessa rete di comunicare tra loro. Il *driver bridge* configura automaticamente le regole nel sistema host in modo che i container su reti bridge diverse non possano comunicare direttamente tra loro. Ciò crea un maggior livello di sicurezza per le applicazioni, garantendo che

solo i servizi correlati possano comunicare tra loro. Significa anche che è possibile definire più reti e servizi che si collegano a funzioni correlate: per esempio i servizi di applicazioni front-end possono utilizzare una rete front-end, mentre i servizi back-end possono utilizzare una rete back-end.

Docker offre funzionalità molto potenti e comode per la persistenza dei dati; nella nostra applicazione, utilizzeremo volumi e *bind mount* per memorizzare in modo permanente le informazioni del database e dei file dell'applicazione e di configurazione. I volumi offrono molta flessibilità per i backup e una persistenza che non dipende dal ciclo di vita di un container, mentre i *bind mount* facilitano le modifiche al codice durante lo sviluppo, apportando immediatamente modifiche ai file host e di conseguenza alle directory presenti nei container. La nostra configurazione utilizzerà entrambi.

Nel file `docker-compose.yml`, abbiamo infatti definito un volume chiamato `dbdata` nella definizione del servizio `db` per rendere persistente il database MySQL; il volume denominato `dbdata` salverà il contenuto della cartella `/var/lib/mysql` presente all'interno del container e ciò consente di arrestare e riavviare il servizio `db` senza perdere dati.

Nella parte inferiore del file, abbiamo aggiunto la definizione per il volume `dbdata`; con questa configurazione, saremo in grado di utilizzare questo volume tra i diversi servizi.

Successivamente, aggiungiamo un *bind mount* al servizio `db` per i file di configurazione di MySQL che creeremo a breve; questo associa il file di configurazione `my.cnf` presente nella cartella `laravel-app` creata nella prima parte di questo paragrafo con la cartella l'omonimo file nella cartella `/etc/mysql` del container.

Successivamente, abbiamo inserito due *bind mount* al servizio webserver: uno per il codice dell'applicazione e un altro per la configurazione di Nginx, che verrà creata nello Step 5. Il primo associa il codice dell'applicazione nella directory `laravel-app` alla directory `/var/www` all'interno del container. Anche il file di configurazione `/laravel-app/nginx/conf.d` verrà montato nella cartella `/etc/nginx/conf.d/` del container, permettendoci di aggiungere o modificare il contenuto della directory di configurazione secondo le necessità dell'applicazione.

Infine, configuriamo dei *bind mount* per il servizio `application` che si occuperanno del codice applicativo e dei file di configurazione; il servizio andrà infatti a montare la cartella `laravel-app`, che contiene il

codice dell'applicazione, nella cartella `/var/www` del container e ciò accelererà il processo di sviluppo, poiché eventuali modifiche apportate alla directory dell'applicazione locale verranno immediatamente riportate all'interno del container. Stiamo inoltre creando un vincolo con il file di configurazione PHP presente in locale nella cartella `laravel-app/php/local.ini` con quello presente nella directory `/usr/local/etc/php/conf.d/local.ini` all'interno del container. Vedremo il contenuto di questo file di configurazione nello Step 4.

Step 3 - Configurazione di Dockerfile

Docker consente di specificare l'ambiente all'interno dei singoli container tramite Dockerfile; questo ci consente di creare immagini personalizzate che possiamo utilizzare per installare il software richiesto dall'applicazione e configurare le impostazioni in base alle nostre esigenze.

Il nostro Dockerfile si troverà nella nostra directory principale `laravel-app`. Questo file imposterà l'immagine di base e specificherà i comandi e le istruzioni necessari per creare l'immagine dell'applicazione Laravel. Aggiungiamo quindi al file il seguente codice.

Listato 9.4 Il file `./laravel-app/Dockerfile`.

```
FROM php:7.2-fpm
COPY composer.lock composer.json /var/www/
WORKDIR /var/www
RUN apt-get update && apt-get install -y \
    build-essential \
    libpng-dev \
    libjpeg62-turbo-dev \
    libfreetype6-dev \
    locales \
    zip \
    jpegoptim optipng pngquant gifsicle \
    vim \
    unzip \
    git \
    curl
RUN apt-get clean && rm -rf /var/lib/apt/lists/*
RUN docker-php-ext-install pdo_mysql mbstring zip exif pcntl
RUN docker-php-ext-configure gd --with-gd --with-freetype-dir=/usr/include/ --with-jpeg-dir=/usr/include/ --with-png-dir=/usr/include/
RUN docker-php-ext-install gd
RUN curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin --filename=composer
RUN groupadd -g 1000 www
RUN useradd -u 1000 -ms /bin/bash -g www www
COPY . /var/www
```

*****ebook converter DEMO

Watermarks*****

```
COPY --chown=www:www . /var/www
USER www
EXPOSE 9000

CMD ["php-fpm"]
```

Innanzitutto, il Dockerfile utilizza l'immagine di base `php:7.2-fpm`; questa è un'immagine basata su Debian sulla quale è installata l'implementazione PHP FastCGI PHP-FPM. Il file installa anche i pacchetti necessari al funzionamento di Laravel: `mcrypt`, `pdo_mysql`, `mbstring` e `imagick` con compositore.

La direttiva `RUN` specifica i comandi per aggiornare, installare e configurare le impostazioni all'interno del container, inclusa la creazione di un utente e un gruppo dedicati chiamati entrambi `www`. L'istruzione `WORKDIR` specifica la directory `/var/www` come cartella di lavoro principale per l'applicazione.

La creazione di un utente e un gruppo dedicati, con autorizzazioni limitate, attenua le vulnerabilità di cui abbiamo parlato nel Capitolo 7 durante l'esecuzione di container Docker, che vengono eseguiti per impostazione predefinita come `root`. Invece di eseguire questo container come `root`, abbiamo creato l'utente `www`, che ha accesso in lettura e scrittura alla cartella `/var/www` e grazie all'istruzione `COPY` e all'opzione `-chown`, abbiamo impostato le autorizzazioni della cartella dell'applicazione.

Infine, il comando `EXPOSE` espone una porta nel container, la porta 9000, per il server `php-fpm`. La direttiva `CMD` specifica il comando che deve essere eseguito una volta creato il container, ossia `php-fpm`, che avvierà il server.

Ora possiamo passare occuparci di configurare PHP.

Step 4 - Configurazione di PHP

Ora che abbiamo definito l'infrastruttura tramite il file `docker-compose.yml`, possiamo dedicarci alla configurazione del servizio PHP in modo che funga da processore per le richieste in arrivo da Nginx.

Per configurare PHP, creeremo il file `local.ini` nella cartella `laravel-app/php`; questo file l'abbiamo specificato quando abbiamo definito il servizio `application` negli Step precedenti. La creazione di questo file ci consentirà di sovrascrivere il file `php.ini` predefinito di PHP, che legge all'avvio, dove sono contenute le informazioni circa il funzionamento

dell'applicazione o eventuali regole da imporre alle richieste in ingresso, come in questo caso: definiremo infatti una dimensione massima permessa per l'upload di file, così come una dimensione massima per le richieste di tipo `POST`, entrambe a 40MB.

Listato 9.5 Il file `./laravel-app/php/local.ini`.

```
upload_max_filesize=40M  
post_max_size=40M
```

Step 5 - Configurazione di Nginx

Ora che il servizio PHP configurato, è possibile modificare il servizio Nginx per utilizzare PHP-FPM come server FastCGI per esporre del contenuto dinamico.

Per configurare Nginx, creeremo un file `app.conf` dove specificheremo la configurazione del servizio e lo inseriremo nella cartella `laravel-app/nginx/conf.d/`; il file conterrà le seguenti specifiche.

Listato 9.6 Il file `./laravel-app/nginx/conf.d/app.conf`.

```
server {  
    listen 80;  
    index index.php index.html;  
    error_log /var/log/nginx/error.log;  
    access_log /var/log/nginx/access.log;  
    root /var/www/public;  
    location ~ \.php$ {  
        try_files $uri =404;  
        fastcgi_split_path_info ^(.+\.php)(/.+)$;  
        fastcgi_pass app:9000;  
        fastcgi_index index.php;  
        include fastcgi_params;  
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;  
        fastcgi_param PATH_INFO $fastcgi_path_info;  
    }  
    location / {  
        try_files $uri $uri/ /index.php?$query_string;  
        gzip_static on;  
    }  
}
```

Il blocco `server` definisce la configurazione per il server web Nginx con le seguenti direttive.

- `listen`: questo blocco definisce la porta su cui il server ascolterà le richieste in arrivo.

- `error_log` e `access_log`: queste direttive definiscono i file dove verranno raccolti i log del server.
- `root`: questa proprietà imposta il percorso della cartella principale, specificando il percorso completo di qualsiasi file richiesto sul file system.

Nel blocco `location` la direttiva `fastcgi_pass` specifica che il servizio `app` è in ascolto su una socket che segue il protocollo TCP sulla porta 9000; in questo modo il server PHP-FPM è in ascolto sulla rete anziché su una socket Unix.

Grazie al *bind mount* creato allo Step 2, tutte le modifiche apportate all'interno della cartella `nginx/conf.d/` verranno riportate direttamente nel container del server web.

Step 6 - Configurazione di MySQL

Per configurare MySQL, creeremo il file `my.cnf` nella cartella `laravel-app/mysql`. Questo file è stato associato sempre a `/etc/mysql/my.cnf` all'interno del container allo Step 2; andrà a sovrascrivere le impostazioni `my.cnf` con le definizioni che vedremo a breve.

Nel file, aggiungiamo il seguente codice, per abilitare il registro delle query e impostare il percorso dei file log.

Listato 9.7 Il file `./laravel-app/mysql/my.cnf`.

```
[mysqld]
general_log = 1
general_log_file = /var/lib/mysql/general.log
```

Questo file abilita i log, impostando `general_log` a `1` per abilitare la registrazione dei log generali; l'impostazione `general_log_file` specifica dove verranno archiviati i log.

Step 7 - Avvio dei container

Ora che abbiamo definito tutti i servizi e che abbiamo creato i file di configurazione per questi servizi, possiamo avviare i container. Come passaggio finale, tuttavia, creeremo una copia del file `.env.example` che

Laravel include per impostazione predefinita e sostituiremo il file con `.env`, dove definiremo tutte le variabili d'ambiente necessarie per il corretto funzionamento del database, come quelle di seguito, di cui modificheremo i valori secondo quelli impostati.

Listato 9.8 Il file `/var/www/.env`.

```
DB_CONNECTION=mysql  
DB_HOST=db  
DB_PORT=3306  
DB_DATABASE=laravel  
DB_USERNAME=user  
  
DB_PASSWORD=your_password
```

A questo punto siamo pronti per avviare i container tramite Docker Compose: utilizzeremo il comando `docker-compose up -d`.

La prima volta che viene eseguito `docker-compose up` su una specifica applicazione, verranno scaricate tutte le immagini Docker necessarie, il che potrebbe richiedere del tempo. Una volta che le immagini sono state scaricate e memorizzate in locale, Compose creerà i container. Il parametro `-d` serve per eseguire i container in background.

Per controllare che l'avvio sia stato eseguito, possiamo eseguire il comando `docker ps` per elencare tutti i container in esecuzione. Tramite browser, potremo vedere la pagina principale di Laravel inserendo l'indirizzo del server nella barra di ricerca.



Figura 9.1 Pagina principale di Laravel.

Applicazione Express.js con MongoDB

Prendiamo ora un esempio reale, così da poterne analizzare in dettaglio le proprietà: vediamo come creare una semplice applicazione Express.js che espone tramite un sito web, integrato con Bootstrap, informazioni recuperate da un database MongoDB. Per prima cosa, creiamo un file `package.json` dove specificare tutte le librerie da installare.

Listato 9.9 Il file `./package.json`.

```
{  
  "name": "nodejs-example",  
  "version": "1.0.0",  
  "description": "nodejs image demo",  
  "author": "Serena Sensini <mail@mail.com>",  
  "license": "MIT",  
  "main": "app.js",  
  "scripts": {  
    "start": "node app.js",  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "keywords": [  
    "nodejs",  
    "bootstrap",  
    "express"  
  ],  
  "dependencies": {  
    "express": "^4.16.4"  
  },  
  "devDependencies": {  
    "nodemon": "^1.18.10"  
  }  
}
```

Dopo aver installato queste dipendenze, lanciando il comando `npm i` all'interno della cartella in cui si trova il file, creiamo un file `app.js` dove esporremo un servizio REST che ci restituisca le informazioni di cui abbiamo bisogno tramite semplici chiamate `GET`.

Listato 9.10 Il file `./app.js`.

```
var express = require("express");  
var app = express();  
var router = express.Router();  
var path = __dirname + '/views/';  
const PORT = 8080;  
const HOST = '0.0.0.0';  
router.use(function (req, res, next) {  
  console.log(req.method);  
  next();  
});  
router.get("/", function (req, res) {  
  res.sendFile(path + "index.html");  
});  
router.get("/fruits", function (req, res) {  
  res.sendFile(path + "fruits.html");  
});
```

*****ebook converter DEMO

Watermarks*****

```

app.use(express.static(path));
app.use("/", router);
app.listen(8080, function () {
  console.log('Sono in ascolto sulla porta 8080!')
})

```

Un primo router renderà disponibile, tramite la chiamata `GET`, la pagina `index.html`, dove potremo visualizzare la pagina principale del sito web statico; la chiamata `/fruits` permetterà invece di visualizzare la pagina relativa, dove sarà presente un elenco di frutti e le relative informazioni.

Predisponiamo dunque le due pagine HTML, creandole all'interno della cartella `views`.

Listato 9.11 || file ./views/index.html.

```

<!DOCTYPE html>
<html lang="en">
<head>
  <title>About fruits</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet"
  href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-MCw98/     SFnGE8fJT3GXwEOngsV7Zt27NXFaoApmYm8liuXoPkFOJwJ8ERdknLPMO" crossorigin="anonymous">
    <link href=".//css/styles.css" rel="stylesheet">
    <link href="https://fonts.googleapis.com/css?family=Merriweather:400,700"           rel="stylesheet" type="text/css">
</head>
<body>
  <nav class="navbar navbar-dark navbar-static-top navbar-expand-md">
    <div class="container">
      <button type="button" class="navbar-toggler collapsed" data-toggle="collapse" data-target="#bs-example-navbar-collapse-1" aria-expanded="false"> <span class="sr-only">Toggle navigation</span>
      </button> <a class="navbar-brand" href="#">Everything fruits</a>
      <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
        <ul class="nav navbar-nav mr-auto">
          <li class="active nav-item"><a href="/" class="nav-link">Home</a>
            </li>
          <li class="nav-item"><a href="/fruits" class="nav-link">fruits</a>
            </li>
        </ul>
      </div>
    </div>
  </nav>
  <div class="jumbotron">
    <div class="container">
      <h1>Want to Learn About fruits?</h1>
      <p>Are you ready to learn about fruits?</p>
      <br>
      <p><a class="btn btn-primary btn-lg" href="/fruits" role="button">Get Fruit Info</a>
      </p>
    </div>
  </div>

```

```

        </div>
    </div>
<div class="container">
    <div class="row">
        <div class="col-lg-6">
            <h3>Not all fruits are alike</h3>
            <p>Though some are dangerous, fruits generally do not attack
humans.                                              Out of the 500 species known to researchers, only 30
have been known to attack humans.
            </p>
        </div>
        <div class="col-lg-6">
            <h3>fruits are ancient</h3>
            <p>There is evidence to suggest that fruits lived up to 400
million years ago.
            </p>
        </div>
    </div>
</body>
</html>

```

Listato 9.12 || file ./views/fruits.html.

```

<!DOCTYPE html>
<html lang="en">
    <head>
        <title>About Fruits</title>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/
bootstrap.min.
css">
        <link href=".//css/styles.css" rel="stylesheet">
        <link href='https://fonts.googleapis.com/css?
family=Merriweather:400,700'           rel='stylesheet' type='text/css'>
        <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js">           <
/script>
        <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js">
        </script>
    </head>
    <nav class="navbar navbar-inverse navbar-static-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle collapsed" data-
toggle="collapse"          data-target="#bs-example-navbar-collapse-1"
aria-expanded="false">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a class="navbar-brand" href="#">Everything fruits</a>
            </div>
            <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
                <ul class="nav navbar-nav mr-auto">
                    <li><a href="/">Home</a></li>
                    <li class="active"><a href="/fruits">Fruits</a></li>
                </ul>
            </div>
        </div>
    </nav>

```

```

        </div>
    </div>
</nav>
<div class="jumbotron text-center">
    <h1>Fruits Info</h1>
</div>
<div class="container">
    <div class="row">
        <div class="col-md-4">
            <p>
                <div class="caption">Some fruits are known to be dangerous to
humans. The yellow star fruit, for example, is not considered a
threat to humans.</div>
                
            </p>
        </div>
        <div class="col-md-4">
            <p>
                <div class="caption">Other fruits are really tasty and available all
over
                    <p>
                        <form action="/fruits/addfruit" method="post">
                            <div class="caption">Enter Your Favorite Fruit</div><br/>
                            <input type="text" placeholder="Fruit Name" name="name"
<%=fruits[i].name; %><br/>
                            <input type="text" placeholder="Colour" name="colour"
<%=fruits[i].colour; %><br/>
                            <button type="submit">Submit</button>
                        </form>
                    </p>
                </div>
            </div>
        </div>
    </div>
</body>

</html>

```

Eseguendo l'applicazione, dovremmo visualizzare all'interno del browser all'indirizzo `http://localhost:8080` una pagina simile a quella rappresentata nella Figura 9.2.

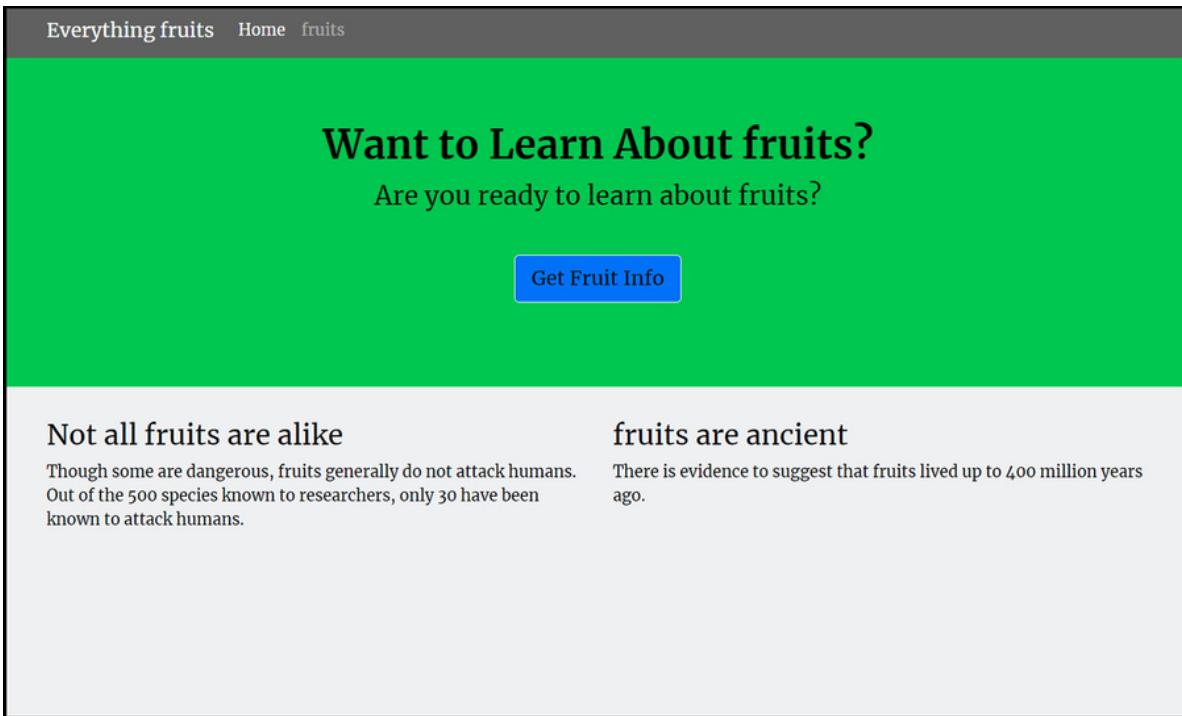


Figura 9.2 Schermata della pagina principale.

A questo punto, colleghiamo un database all'applicazione, in modo da poter recuperare le informazioni in maniera dinamica, e non statica come abbiamo fatto finora; definiamo dunque un file `db.js` dove importeremo le dipendenze necessarie per collegarci al database MongoDB e per creare una connessione.

Listato 9.13 Il file `./db.js`.

```
const mongoose = require('mongoose');
const {
  MONGO_USERNAME,
  MONGO_PASSWORD,
  MONGO_HOSTNAME,
  MONGO_PORT,
  MONGO_DB
} = process.env;
const options = {
  useNewUrlParser: true,
  reconnectTries: Number.MAX_VALUE,
  reconnectInterval: 500,
  connectTimeoutMS: 10000,
};
const url =
`mongodb://${MONGO_USERNAME}:${MONGO_PASSWORD}@${MONGO_HOSTNAME}: ${MONGO_PORT}
/${MONGO_DB}?authSource=admin`;
mongoose.connect(url, options).then( function() {
  console.log('MongoDB is connected');
})
```

```

    .catch( function(err) {
  console.log(err);
}) ;

```

Il prossimo passo sarà rendere più robusto il nostro metodo di connessione al database, aggiungendo del codice che gestisca i casi in cui la nostra applicazione non riesca a connettersi al database. L'introduzione di questo livello di resilienza al codice dell'applicazione è una pratica consigliata quando si lavora con i container utilizzando Docker Compose.

Vediamo il codice appena aggiunto: abbiamo importato la libreria `mongoose` per gestire il metodo di connessione; aggiungiamo alcune opzioni al metodo `connect()`, come i parametri per i tentativi di riconnessione. Possiamo farlo creando una costante `options` che include le informazioni pertinenti.

L'opzione `reconnectTries` dice a Mongoose di continuare a provare a connettersi indefinitamente, mentre `reconnectInterval` definisce il periodo tra i tentativi di connessione in millisecondi. `connectTimeoutMS` definisce 10 secondi come valore per il periodo di tempo che il driver attenderà prima di dichiarare il fallimento del tentativo di connessione.

Invece di codificare all'interno di questo file le informazioni sensibili, come le credenziali di accesso al database, possiamo utilizzare l'oggetto `process.env` per acquisire i valori per queste costanti. Per fare questo, creiamo un file `.env` dove inseriremo queste informazioni utilizzando le chiavi riportate nel file precedente.

Listato 9.14 Il file `./.env`.

```

MONGO_USERNAME=utente
MONGO_PASSWORD=password
MONGO_PORT=27017
MONGO_DB=fruits

```

Poiché il nostro file `.env` contiene informazioni riservate, dovremo assicurarci che venga specificato nel file `.dockerignore`, in modo che queste non vengano copiate all'interno dei container. Oltre al file `.env`, inseriamo anche i file relativi alle dipendenze di Node, così da non rendere più pesante la build dell'immagine.

Listato 9.15 Il file `./.dockerignore`.

```

node_modules
npm-debug.log

.env

```

Ora dobbiamo dedicare le nostre attenzioni a creare dei router Express che ci permettano di creare un oggetto e restituire l'elenco di quelli presenti nel database; a tal proposito, creiamo due file, `index.js` e `fruits.js`, dove inseriamo il codice per la creazione di un'applicazione Express e la gestione delle *route* che permettono di collegarsi alle diverse pagine.

Listato 9.16 || file `./index.js`.

```
const express = require('express');
const router = express.Router();
const path = require('path');
router.use(function (req, res, next) {
  next();
});
router.get('/', function(req, res) {
  res.sendFile(path.resolve('views/index.html'));
});
module.exports = router;
```

Listato 9.17 || file `./fruits.js`.

```
const express = require('express');
const router = express.Router();
const fruit = require('../controllers/fruits');
router.get('/', function(req, res) {
  fruit.index(req, res);
});
router.post('/addfruit', function(req, res) {
  fruit.create(req, res);
});
router.get('/getfruit', function(req, res) {
  fruit.list(req, res);
});
module.exports = router;
```

Senza addentrarci troppo nel codice *Node.js*, che certamente non rientra negli scopi del libro, vediamo brevemente che cosa eseguono i due file: nel file `index.js` viene predisposto Express, che caricherà la pagina principale dell'applicazione web, ossia `index.html`; nel file `fruits.js` vengono invece gestite le chiamate per recuperare l'elenco degli oggetti e anche l'inserimento, qualora l'utente volesse inserirne uno. In questo caso, viene chiamato il controller di riferimento (riportato nel riepilogo del codice alla fine del paragrafo), che si occuperà di gestire la chiamata al database.

Con il codice dei file `db.js`, `index.js`, `fruits.js` e `.dockerignore`, siamo pronti per configurare il file `docker-compose.yml` con le definizioni del servizio. Un servizio, in Compose, non è nient'altro che un container in esecuzione e la definizione di un servizio - che includeremo nel file

`docker-compose.yml` - conterrà le informazioni su come verrà eseguita ciascuna immagine del container. Lo strumento Compose consente infatti di definire più servizi per creare applicazioni multi-container.

Prima di definire i nostri servizi, tuttavia, aggiungeremo al nostro progetto lo strumento `wait-for`, per garantire che la nostra applicazione tenti di connettersi al nostro database solo una volta completate le attività di avvio del database. Questo script utilizza `netcat` per eseguire il *polling* e verificare se il servizio del database accetta o meno connessioni ed è quindi disponibile. Il suo utilizzo consente di controllare i tentativi dell'applicazione di connettersi al database, verificando se il database è pronto o meno ad accettare connessioni.

Sebbene Compose consenta di specificare le dipendenze tra i servizi usando l'opzione `depends_on`, questo parametro stabilisce l'ordine con cui avviare dei container, ma non verifica la sua disponibilità. L'uso di `depends_on` non è ottimale in questo caso, poiché vogliamo che la nostra applicazione si connetta solo quando le attività di avvio del database, inclusa l'aggiunta di un utente e una password al database di autenticazione dell'amministratore, sono complete. Per ulteriori informazioni sull'utilizzo di `wait-for` e altri strumenti per controllare l'ordine di avvio, puoi consultare le raccomandazioni presenti nella documentazione di Docker Compose.

Creiamo dunque un file `wait.sh` (disponibile nella versione originale sul repository <https://github.com/eficode/wait-for>) e inseriamo il seguente codice.

Listato 9.18 Il file `./wait.sh`.

```
#!/bin/sh
TIMEOUT=15
QUIET=0
echoerr() {
    if [ "$QUIET" -ne 1 ]; then printf "%s\n" "$*" 1>&2; fi
}
usage() {
    exitcode="$1"
    cat << USAGE >&2
Usage:
    $cmdname host:port [-t timeout] [-- command args]
    -q | --quiet                         Non stampare alcun messaggio
    -t TIMEOUT | --timeout=timeout        Secondi, zero se non esiste
    -- COMMAND ARGS                       Eseguire dopo il test
USAGE
    exit "$exitcode"
}
wait_for() {
    for i in `seq $TIMEOUT` ; do
```

```

nc -z "$HOST" "$PORT" > /dev/null 2>&1
result=$?
if [ $result -eq 0 ] ; then
    if [ $# -gt 0 ] ; then
        exec "$@"
    fi
    exit 0
fi
sleep 1
done
echo "Time out raggiunto" >&2
exit 1
}
while [ $# -gt 0 ]
do
    case "$1" in
        *:* )
            HOST=$(printf "%s\n" "$1" | cut -d : -f 1)
            PORT=$(printf "%s\n" "$1" | cut -d : -f 2)
            shift 1
            ;;
        -q | --quiet)
            QUIET=1
            shift 1
            ;;
        -t)
            TIMEOUT="$2"
            if [ "$TIMEOUT" = "" ]; then break; fi
            shift 2
            ;;
        --timeout=*)
            TIMEOUT="${1#*=}"
            shift 1
            ;;
        --)
            shift
            break
            ;;
        --help)
            usage 0
            ;;
        *)
            echoerr "Argomento non trovato: $1"
            usage 1
            ;;
    esac
done
if [ "$HOST" = "" -o "$PORT" = "" ]; then
    echoerr "Errore: fornire un host e una porta"
    usage 2
fi
wait_for "$@"

```

In questo script gestiamo delle funzioni che vanno a eseguire delle chiamate di test al database, per verificare se è in esecuzione su un container e disponibile per le funzioni previste dall'applicazione; può dunque essere utile per verificare se l'host e la porta passati come parametri sono disponibili, modificando i permessi di esecuzione ed

eseguendo il comando tramite la riga di comando, come nell'esempio riportato.

Listato 9.19 Esecuzione dello script wait.sh.

```
$ chmod +x wait.sh
$ ./wait.sh www.google.com:80 -- echo "google is up"
Connection to www.google.com port 80 [tcp/http] succeeded!
google is up
```

A questo punto, procediamo con la definizione del file `docker-compose.yml` e dei relativi servizi.

Listato 9.20 Il file `./docker-compose.yml`.

```
version: '3'
services:
  nodejs:
    build:
      context: .
      dockerfile: Dockerfile
    image: nodejs
    container_name: nodejs
    restart: unless-stopped
    env_file: .env
    environment:
      - MONGO_USERNAME=$MONGO_USERNAME
      - MONGO_PASSWORD=$MONGO_PASSWORD
      - MONGO_HOSTNAME=db
      - MONGO_PORT=$MONGO_PORT
      - MONGO_DB=$MONGO_DB
    ports:
      - "80:8080"
    volumes:
      - .:/home/node/app
      - node_modules:/home/node/app/node_modules
    networks:
      - mynetwork
    command: ./wait.sh db:27017 -- /home/node/app/node_modules/.bin/nodemon app.js
...
...
```

Questo esempio presenta un file `docker-compose.yml` abbastanza complesso, perché contiene molti attributi mai visti finora; vediamo dunque la definizione del servizio `nodejs`, la quale include le seguenti configurazioni.

- `build`: definisce le opzioni di configurazione, inclusi il contesto e il Dockerfile, che verranno applicati quando Compose creerà l'immagine dell'applicazione.
 - `context`: definisce il contesto di compilazione per la creazione dell'immagine, in questo caso la directory del progetto corrente.

- `dockerfile`: specifica il Dockerfile da utilizzare nella directory del progetto corrente.
 - `image, container_name`: applicano degli alias all'immagine e al container.
 - `restart`: definisce il criterio di riavvio. L'impostazione predefinita è `no`, ma abbiamo impostato il riavvio del container se questo viene arrestato.
 - `env_file`: chiede di aggiungere delle variabili d'ambiente a partire da un file chiamato `.env`, situato nel contesto di compilazione.
 - `environment`: l'utilizzo di questa opzione consente di aggiungere le impostazioni di connessione Mongo definite nel file `.env`.
 - `ports`: mappa la porta 80 dell'host alla porta 8080 del container.
 - `volumes`: qui includiamo due tipi di supporti.
 - Il primo è un *bind mount*, che monta il codice dell'applicazione sull'host nella cartella `/home/node/app` sul container. Ciò faciliterà uno sviluppo rapido, poiché eventuali modifiche apportate al codice originale verranno rese disponibili immediatamente nel container.
 - Il secondo è un volume chiamato `node_modules`; quando Docker esegue le istruzioni di installazione di `npm` elencate nell'applicazione Dockerfile, sul container viene creata una nuova cartella `node_modules`, che include i pacchetti richiesti per eseguire l'applicazione. In questo modo, il volume avrà a disposizione in maniera permanente il contenuto della directory `/home/node/app/node_modules`.
 - `networks`: specifica che il servizio applicativo farà parte della rete `app-network`.
 - `command`: consente di impostare il comando che deve essere eseguito quando Compose esegue l'immagine. Ciò sovrascriverà le istruzioni `CMD` impostate nel Dockerfile dell'applicazione. Qui, stiamo eseguendo l'applicazione utilizzando lo script `wait`, per eseguire il polling del servizio `db` sulla porta 27017, con lo scopo di verificare che il servizio sia disponibile. Una volta completato il test, lo script eseguirà il comando che abbiamo impostato,
- `/home/node/app/node_modules/.bin/nodemon app.js`, per avviare l'applicazione.

Quindi creiamo il servizio `db`, aggiungendo il seguente codice in seguito alla definizione del servizio `nodejs`.

Listato 9.21 Il file ./docker-compose.yml.

```
...
db:
  image: mongo:4.1.8-xenial
  container_name: db
  restart: unless-stopped
  env_file: .env
  environment:
    - MONGO_INITDB_ROOT_USERNAME=$MONGO_USERNAME
    - MONGO_INITDB_ROOT_PASSWORD=$MONGO_PASSWORD
  volumes:
    - dbdata:/data/db
  networks:
    - mynetwork
...
...
```

Alcune delle impostazioni che abbiamo definito per il servizio nodejs sono simili, altre possono invece meritare dei chiarimenti.

- `image`: per creare questo servizio, Compose estraе l'immagine `Mongo 4.1.8-xenial` dal Docker Hub. Stiamo specificando una versione particolare di questa immagine per evitare possibili conflitti futuri quando l'immagine Mongo verrà aggiornata, e questo per le ragioni spiegate nel Capitolo 7 dedicato alle *best practice*.
- `environments`.
 - `MONGO_INITDB_ROOT_USERNAME`, `MONGO_INITDB_ROOT_PASSWORD`: l'immagine `mongo` rende disponibili queste variabili d'ambiente, in modo da poter modificare l'inizializzazione dell'istanza del database. Queste due variabili creano un utente *root* nel database di autenticazione dell'amministratore e assicurano che l'autenticazione sia abilitata all'avvio del container. Abbiamo impostato i loro valori sfruttando quelli presenti nel file `.env` utilizzato in precedenza.
- `volumes`.
 - `dbdata:/data/db`: il volume denominato `dbdata` conserverà i dati memorizzati nella directory dei dati predefinita di Mongo, ossia `/data/db`. Ciò assicurerà di non perdere dati nei casi in cui il container venga interrotto o rimosso.

Come passaggio finale, aggiungiamo il volume e le definizioni di rete nella parte inferiore del file.

Listato 9.22 Il file ./docker-compose.yml.

```

...
networks:
  mynetwork:
    driver: bridge
volumes:
  dbdata:

  node_modules:

```

La rete `mynetwork` definita dall'utente consente la comunicazione tra i container, poiché si trovano sullo stesso host; ciò semplifica il traffico e la comunicazione all'interno dell'applicazione, in quanto apre tutte le porte tra i container sulla stessa rete sfruttando la rete bridge, senza esporre porte al mondo esterno. Pertanto, i nostri container `db` e `nodejs` possono comunicare tra loro e dobbiamo solo esporre la porta 80 per l'accesso front-end all'applicazione.

Per quanto riguarda invece i volumi, quando vengono creati, i contenuti sono memorizzati in una parte del file system host, ossia la cartella `/var/lib/docker/volumes/`, sempre gestita da Docker.

Il file `docker-compose.yml`, a questo punto, sarà simile al seguente.

Listato 9.23 Il file `./docker-compose.yml`.

```

version: '3'
services:
  nodejs:
    build:
      context: .
      dockerfile: Dockerfile
    image: nodejs
    container_name: nodejs
    restart: unless-stopped
    env_file: .env
    environment:
      - MONGO_USERNAME=$MONGO_USERNAME
      - MONGO_PASSWORD=$MONGO_PASSWORD
      - MONGO_HOSTNAME=db
      - MONGO_PORT=$MONGO_PORT
      - MONGO_DB=$MONGO_DB
    ports:
      - "80:8080"
    volumes:
      - .:/home/node/app
      - node_modules:/home/node/app/node_modules
    networks:
      - mynetwork
    command: ./wait.sh db:27017 -- /home/node/app/node_modules/.bin/nodemon
app.js
db:
  image: mongo:4.1.8-xenial
  container_name: db
  restart: unless-stopped
  env_file: .env
  environment:

```

```

    - MONGO_INITDB_ROOT_USERNAME=$MONGO_USERNAME
    - MONGO_INITDB_ROOT_PASSWORD=$MONGO_PASSWORD
  volumes:
    - dbdata:/data/db
  networks:
    - mynetwork
  networks:
    mynetwork:
      driver: bridge
  volumes:
    dbdata:

  node_modules:

```

Con il file `docker-compose.yml` in atto, è possibile creare nuovi servizi con il comando `docker-compose up`. Inoltre è possibile verificare che i dati vengano memorizzati, arrestando e rimuovendo i container con il comando `docker-compose down`.

Per sottoporre a test l'applicazione, eseguiamo il comando `docker-compose up` con l'opzione `-d`, che avvierà i container in background; una volta che i servizi saranno avviati, aprendo il browser sarà possibile accedere all'applicazione web e utilizzarne le funzionalità.

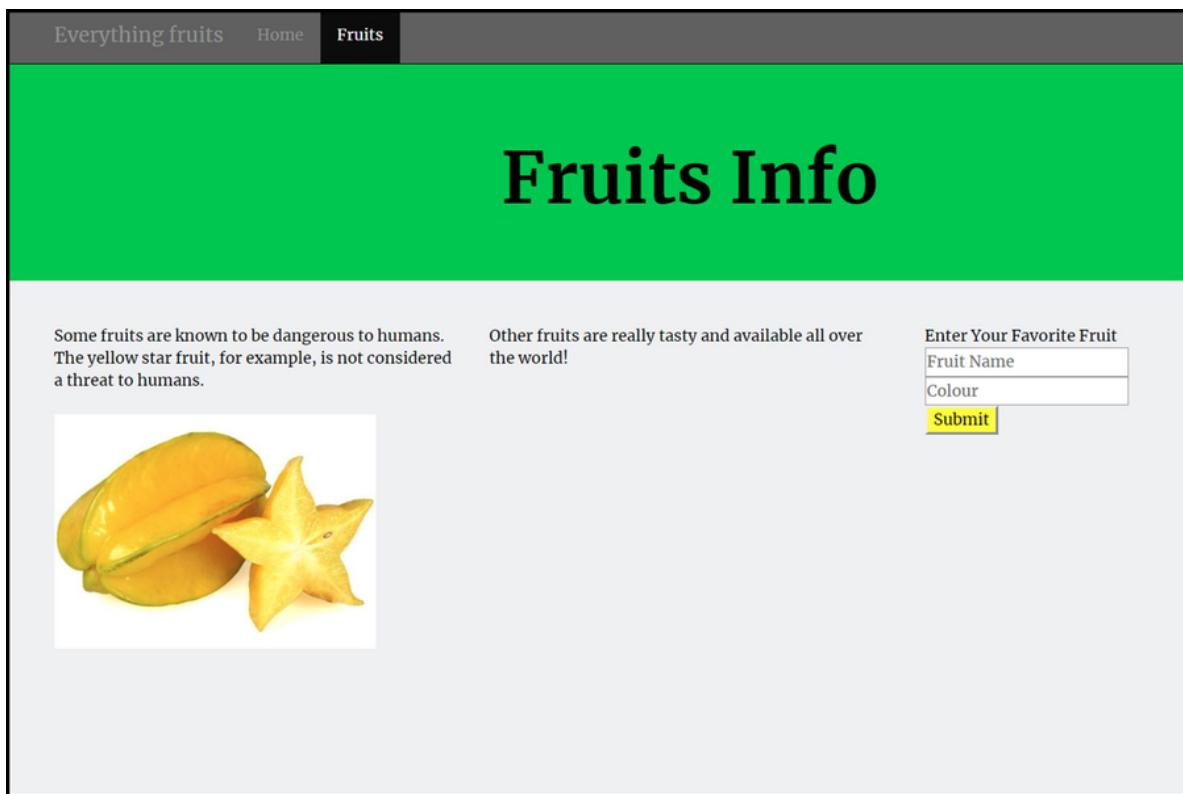


Figura 9.3 Schermata della pagina fruits.html.

Applicazione Flask con SQLite

Un caso d'uso molto comune della libreria Flask per Python è quello di creare un'applicazione web, piuttosto che delle API REST; vediamo dunque come creare un'applicazione web molto semplice che faccia anche uso di un database SQLite, sfruttando però le potenzialità di Docker.

È considerata una buona pratica per un container avere una sola responsabilità e un solo processo da gestire, quindi per la nostra app avremo bisogno di almeno *due container*: uno per l'esecuzione dell'app stessa e uno per l'esecuzione del database. Come coordiniamo questi container? Qui entrerà in gioco Docker Compose.

Il progetto conterrà due cartelle, `/app` e `/db`, che conterranno rispettivamente tre file, ossia il file `app.py`, dove creeremo l'applicazione che ci collegherà al database e ci esporrà delle informazioni tramite un endpoint, il file `requirements.txt`, dove definiremo le librerie che vanno installate all'avvio del container, e il file `dump.sql`, ovvero lo script necessario a inizializzare il database al primo avvio. Naturalmente, nella cartella dell'applicazione andrà il Dockerfile, in cui configureremo il sistema: come immagine di base, utilizzeremo quella Python nella versione 3.6. Esporteremo la porta 5000, che è quella di default utilizzata in ascolto da Flask; dopo aver installato le dipendenze, eseguiremo l'app tramite Python.

Impostiamo il seguente Dockerfile.

Listato 9.24 Il file /app/Dockerfile.

```
# immagine di base
FROM python:3.6
# esponiamo la porta 5000
EXPOSE 5000
# cartella di lavoro
WORKDIR /app
# copia del file dove sono definite le librerie da definire nella cartella di lavoro
COPY requirements.txt /app
# installazione delle librerie tramite pip
RUN pip install -r requirements.txt
# copia del file che contiene l'applicazione nella cartella di lavoro
COPY app.py /app
# esecuzione dell'applicazione

CMD python app.py
```

Abbiamo bisogno che le dipendenze `Flask` e `mysql-connector` siano installate con l'immagine, quindi dobbiamo creare il file `requirements.txt` e inserire le dipendenze.

Listato 9.25 Il file `/app/requirements.txt`.

```
Flask
mysql-connector
```

Ora possiamo creare un'immagine Docker per la nostra applicazione, ma non possiamo ancora utilizzarla, poiché dipende da MySQL, il quale risiederà in un altro container.

Useremo Docker Compose per facilitare la gestione dei due container indipendenti in un'unica applicazione. A tal proposito, è necessario impostare il file `docker-compose.yml` nella cartella principale del progetto, dove dobbiamo definire i servizi che costituiscono la nostra soluzione, ossia l'applicazione e il database.

Listato 9.26 Il file `docker-compose.yml`.

```
version: "2"
services:
  app:
    build: ./app
    links:
      - db
    ports:
      - "5000:5000"
    db:
      image: mysql:5.7
      ports:
        - "32000:3306"
      environment:
        MYSQL_ROOT_PASSWORD: root
      volumes:
        - ./db:/docker-entrypoint-initdb.d/:ro
```

In questo caso, stiamo utilizzando due servizi: uno è il container che espone le API REST e uno contiene il database. Vediamo nel dettaglio i vari attributi.

- `build`: specifica la directory contenente le istruzioni per la creazione di questo servizio.
- `links`: collega questo servizio a un altro container. Questo ci consentirà anche di utilizzare il nome del servizio invece di dover

trovare l'IP del container del database, e inoltre esplicita la dipendenza che determinerà l'ordine di avvio del container.

- `ports`: *mapping* delle porte `<host>: <container>`; si noti che il mapping è solo *dall'host al container*, quindi il container contenente le API REST utilizzerà comunque la porta 3306 per connettersi al database.
- `image`: usiamo un'immagine esistente come base. È importante specificare la versione: se il client MySQL installato non ha la stessa versione, potrebbero verificarsi problemi.
- `environment`: aggiunge le variabili d'ambiente. La variabile specificata è richiesta per questa immagine e, come suggerisce il nome, configura la password per l'utente root di MySQL in questo container.
- `volumes`: poiché vogliamo che il container venga inizializzato con questo schema, colleghiamo la directory contenente il nostro script `dump.sql` al punto di ingresso per questo container, che secondo le specifiche dell'immagine esegue tutti gli script `.sql` contenuti nella directory specificata.

Ora siamo pronti per avviare l'applicazione. Ma prima di farlo, diamo un'occhiata al codice che permette alle API di collegarsi al database.

Listato 9.27 || file /app/app.py

```
from typing import List, Dict
from flask import Flask
import mysql.connector
import json
app = Flask(__name__)
def getPets() -> List[Dict]:
    config = {
        'user': 'root',
        'password': 'root',
        'host': 'db',
        'port': '3306',
        'database': 'zoo'
    }
    connection = mysql.connector.connect(**config)
    cursor = connection.cursor()
    cursor.execute('SELECT * FROM pets')
    results = [{name: pet} for (name, pet) in cursor]
    cursor.close()
    connection.close()
    return results
@app.route('/')
def index() -> str:
    return json.dumps({'my-pets': getPets()})
if __name__ == '__main__':
```

```
app.run(host='0.0.0.0')
```

Nella prima parte del file, importiamo le librerie necessarie al corretto funzionamento delle API; dopo aver creato un'app Flask (riga 6), definiamo una funzione `getPets()` che restituisce un dizionario contenente tutti gli animali presenti nel database specificato (righe 9-24). Il sistema sarà in ascolto sull'indirizzo principale (riga 27) e restituirà un JSON con le informazioni richieste (riga 29).

In questo momento ci stiamo connettendo come utenti *root* con la password configurata nel file `docker-compose.yml`; è bene osservare che definiamo esplicitamente l'host (`localhost` per impostazione predefinita) poiché il servizio SQL si trova effettivamente in un container diverso da quello che esegue questo frammento di codice. Possiamo (e dovremmo) usare il nome `db` come valore del parametro `host` poiché questo è il nome del servizio che abbiamo definito e collegato in precedenza, e la porta esposta è la 3306, e non la 32000, poiché questo codice non è in esecuzione sull'host.

Per eseguire l'applicazione appena descritta, impiegheremo il seguente comando dal terminale.

Listato 9.28 Avvio di docker-compose.

```
root@vbox:/home/flask# docker-compose up
```

È possibile vedere l'immagine in fase di creazione, i pacchetti installati tramite il file `requirements.txt` e l'avvio dei container. Se tutto è andato per il verso giusto, vedremo la seguente riga.

```
app_1  | * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Possiamo verificare che tutto stia funzionando come previsto inserendo questo URL in un browser o usando il comando `curl` e ricevendo la seguente risposta.

```
{"my-pets": [{"Birillo": "cane"}, {"Graffio": "gatto"}]}
```

Naturalmente questo esempio può essere utilizzato senza particolari configurazioni su ogni sistema, dal momento che contiene tutto l'essenziale affinché l'applicazione sia funzionante e consistente.

Installazione di Drupal con Docker Compose

Drupal è un sistema di gestione dei contenuti (ovvero un *CMS*) scritto in PHP e distribuito sotto la GNU General Public License. Sempre più persone e organizzazioni in tutto il mondo usano Drupal per creare siti governativi, blog personali o siti aziendali; ciò che rende Drupal unico rispetto ad altri framework CMS è la community di sviluppatori in forte crescita e anche un insieme di funzionalità che includono processi sicuri, prestazioni affidabili, modularità e flessibilità.

Drupal richiede l'installazione dello stack LAMP (*Linux, Apache, MySQL e PHP*) o LEMP (*Linux, Nginx, MySQL e PHP*); l'installazione dei singoli componenti è un'operazione che richiede del tempo. In questo paragrafo utilizzeremo le immagini Docker per l'installazione dei singoli componenti all'interno dei container Docker. Utilizzando Docker Compose, possiamo definire e gestire più container separati per mettere in piedi il database, l'applicazione e gestire la comunicazione tra di loro.

In questo esempio, installeremo Drupal usando Docker Compose, in modo da poter trarre vantaggio dalla *containerizzazione* e da poter distribuire il nostro sito web Drupal su qualunque server. Gestiremo un container per il database MySQL, un server web Nginx e infine Drupal; garantiremo inoltre l'utilizzo di un processo di sicurezza ottenendo certificati TLS/SSL tramite l'uso di *Let's Encrypt* per il dominio che vogliamo associare al nostro sito. Infine, imposteremo un *cron job* per rinnovare i nostri certificati, in modo che il nostro dominio rimanga sicuro.

Prima di avviare qualsiasi container, è necessario definire la configurazione per il nostro server web Nginx; il file di configurazione includerà alcuni blocchi `location` specifici di Drupal, insieme a un blocco `location` per indirizzare le richieste di verifica di Encrypt al client Certbot per il rinnovo automatico dei certificati.

Dopo aver creato la cartella principale del progetto, al suo interno creiamo una directory `nginx-conf`, dove inseriremo il file di configurazione `nginx.conf` necessario per la predisposizione del server; in questo file aggiungeremo un blocco `server` con le direttive che specificano il nome del server e la radice del documento, oltre che i blocchi `location` per

indirizzare la richiesta del client Certbot per certificati, per l'elaborazione PHP e per le richieste di risorse statiche.

Aggiungiamo il seguente codice nel file appena creato; al posto di *mio_dominio*, sarà necessario inserire il nostro dominio. Per creare uno gratuito, ci sono moltissimi siti; qualora volessimo sottoporre a test la soluzione in locale, possiamo utilizzare i puntamenti a `localhost`.

Listato 9.29 Il file `./nginx-conf/nginx.conf`.

```
server {
    listen 80;
    listen [::]:80;
    server_name mio_dominio www.mio_dominio;
    index index.php index.html index.htm;
    root /var/www/html;
    location ~ /.well-known/acme-challenge {
        allow all;
        root /var/www/html;
    }
    location / {
        try_files $uri $uri/ /index.php$is_args$args;
    }
    rewrite ^/core/authorize.php/core/authorize.php(.*)$ /core/authorize.php$1;
    location ~ \.php$ {
        try_files $uri =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass drupal:9000;
        fastcgi_index index.php;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param PATH_INFO $fastcgi_path_info;
    }
    location ~ /\.ht {
        deny all;
    }
    location = /favicon.ico {
        log_not_found off; access_log off;
    }
    location = /robots.txt {
        log_not_found off; access_log off; allow all;
    }
    location ~* \.(css|gif|ico|jpeg|jpg|js|png)$ {
        expires max;
        log_not_found off;
    }
}
```

Il blocco `server` include le seguenti informazioni, per quanto riguarda le direttive.

- `listen`: dice a Nginx di stare in ascolto sulla porta 80, il che permetterà l'utilizzo del plugin `webroot` di Certbot per effettuare le richieste di certificati. Si noti che non stiamo ancora includendo la

porta 443, che di solito viene utilizzata in presenza del protocollo SSL: aggiorneremo la configurazione una volta ottenuti con successo i certificati.

- `server_name`: definisce il nome del server.
- `index`: definisce i file che verranno utilizzati come indici durante l'elaborazione delle richieste al server. Abbiamo modificato l'ordine di priorità predefinito, spostando `index.php` davanti a `index.html`, in modo che Nginx dia la priorità ai file chiamati `index.php`.
- `root`: definisce la directory *root* per le richieste al nostro server. La cartella specificata viene “montata” al momento della compilazione dalle istruzioni nel Dockerfile Drupal, che vedremo in seguito. Queste istruzioni assicurano che anche i file della versione Drupal siano montati su questo stesso volume.
- `rewrite`: se l'espressione regolare corrisponde a un URI di richiesta, l'URI viene modificato come specificato nella stringa di sostituzione.

Per quanto riguarda invece la gestione degli attributi `location` abbiamo le seguenti annotazioni.

- *Righe 11-14*: questo blocco gestirà le richieste alla directory `.well-known`, dove *Certbot* inserirà un file temporaneo per convalidare la risoluzione del DNS per il dominio sul nostro server. Con questa configurazione, saremo in grado di utilizzare il plugin `webroot` di Certbot per ottenere certificati.
- *Righe 16-18*: in questo blocco utilizzeremo una direttiva `try_files` per verificare la presenza di file che corrispondono alle singole richieste URI; questo perché, invece di restituire uno stato 404 come predefinito, passeremo il controllo al file `index.php` di Drupal con gli argomenti della `request`.
- *Righe 22-30*: questo blocco gestirà l'elaborazione PHP e inoltrerà queste richieste al container Drupal. Dal momento che l'immagine Drupal è basata sull'immagine `php:fpm`, in questo blocco includeremo anche le opzioni di configurazione specifiche per il protocollo FastCGI. Nginx richiede un processore PHP indipendente per le richieste PHP: nel nostro caso, queste richieste saranno gestite dal

processore `php-fpm` incluso nell'immagine. Inoltre, questo blocco include direttive, variabili e opzioni specifiche di FastCGI, che faranno da proxy per le richieste all'applicazione Drupal in esecuzione nel nostro container Drupal, il quale imposterà l'indice preferito per l'URI della richiesta e analizzerà le richieste dell'URI.

- *Righe 32-34:* questo blocco gestirà i file `.htaccess`; la direttiva `deny_all` assicura che i file `.htaccess` non saranno mai resi visibili agli utenti.
- *Righe 36-42:* questi blocchi si assicurano che le richieste per il caricamento della `favicon.ico` e per il file `robots.txt` non vengano registrate tra i log del server.
- *Righe 42-45:* questo blocco disattiva la registrazione delle richieste di risorse statiche e garantisce che queste risorse siano altamente memorizzabili nella cache, poiché sono generalmente costose da servire.

È bene chiarire alcuni aspetti: come sempre, molte delle operazioni qui specificate rientrano tra le *best practice* da attuare quando si progetta una soluzione web, tuttavia non sono sempre generalizzabili; le ultime due citate sono sicuramente regole che è bene applicare nella maggior parte dei casi relativi ad applicazioni *web-based*, così come quella specificata nelle righe 16-18. Le altre opzioni sono state adottate per questo singolo caso e vanno applicate a seconda del tipo di soluzione progettata.

Ora che la configurazione di Nginx terminata, possiamo passare alla creazione delle variabili d'ambiente da passare ai container dell'applicazione e del database in fase di esecuzione.

La nostra applicazione Drupal necessita di un database per il salvataggio delle informazioni relative al sito; il container Drupal dovrà accedere ad alcune variabili d'ambiente in fase di esecuzione per poter utilizzare le informazioni contenute nel container del database (*MySQL*). Queste variabili contengono informazioni sensibili, come le credenziali del database, quindi non possiamo esporle direttamente nel file `docker-compose.yml`; è sempre meglio impostare i valori sensibili in un file `.env` e limitarne la circolazione. Ciò impedirà a questi valori di essere copiati nei repository dei nostri progetti e di essere esposti pubblicamente.

Nella cartella principale del progetto creiamo dunque un file `.env` e aggiungiamogli le seguenti variabili d'ambiente.

Listato 9.30 Il file `./.env`.

```
MYSQL_ROOT_PASSWORD=root_password
MYSQL_DATABASE=drupal
MYSQL_USER=drupal_database_user
MYSQL_PASSWORD=drupal_database_password
```

Quello che abbiamo fatto è stato aggiungere la password per l'account amministrativo di MySQL, così come il nostro nome utente e password per il database dell'applicazione, che ovviamente vanno sostituiti con i valori reali delle credenziali.

Il file `.env` contiene informazioni riservate, pertanto si consiglia sempre di includerle nei file `.gitignore` e `.dockerignore` di un progetto, in modo che non vengano aggiunti nei nostri repository Git e nelle immagini Docker.

A questo punto è ora di procedere con la configurazione del file `docker-compose.yml`: questo ci permetterà di creare diversi container, ciascuno con il proprio compito, per creare una soluzione che sia la più modulare e flessibile possibile; nella cartella principale del progetto creiamo dunque questo file e aggiungiamo il seguente codice per la creazione del servizio MySQL.

Listato 9.31 Il file `./docker-compose.yml`.

```
version: "3"
services:
  mysql:
    image: mysql:8.0
    container_name: mysql
    command: --default-authentication-plugin=mysql_native_password
    restart: unless-stopped
    env_file: .env
    volumes:
      - db-data:/var/lib/mysql
    networks:
      - internal
...

```

Come sempre, esaminiamo le righe del codice, per comprendere appieno le impostazioni definite.

- `image`: specifica l'immagine che verrà utilizzata per la creazione del container. Si consiglia sempre di utilizzare l'immagine con il tag di una versione specifica, escluso il tag più recente per evitare conflitti futuri.
- `container_name`: per definire il nome del container che potremo usare come alias.

- `command`: viene utilizzato per sovrascrivere il comando predefinito (ossia l’istruzione `CMD`) nell’immagine. MySQL supporta diversi plugin di autenticazione, ma `mysql_native_password` rappresenta il metodo tradizionale per l’autenticazione. Dato che PHP, e quindi Drupal, non supportano la nuova autenticazione MySQL, dobbiamo impostare questo parametro come meccanismo di autenticazione predefinito, così che sia possibile usarlo nell’installazione di Drupal.
- `restart`: viene utilizzato per definire la politica di riavvio del container; questo fa sì che il container non venga riavviato, a meno che non venga arrestato manualmente.
- `env_file`: questo recupera le variabili d’ambiente a partire un file; nel nostro caso, leggerà le variabili d’ambiente dal file `.env` definito nel passaggio precedente.
- `volumes`: monta i volumi a disposizione dei servizi; in questo caso, stiamo montando un volume denominato `db-data` nella directory `/var/lib/mysql` del container, dove MySQL scriverà automaticamente i dati.
- `networks`: definisce la rete `internal` alla quale aderirà il nostro servizio applicativo.

Passiamo ora al servizio Drupal.

Listato 9.32 Il file `./docker-compose.yml`.

```
...
drupal:
    image: drupal:8.7.8-fpm-alpine
    container_name: drupal
    depends_on:
        - mysql
    restart: unless-stopped
    networks:
        - internal
        - external
    volumes:
        - drupal-data:/var/www/html
...

```

Nel definire questo servizio, stiamo creando il container con il nome `drupal` per Drupal e stiamo definendo una politica di riavvio, come abbiamo fatto con il servizio per MySQL. Oltre a queste opzioni, ne stiamo aggiungendo anche alcune specifiche.

- `image`: qui stiamo usando l'immagine Drupal `8.7.8-fpm-alpine`; questa immagine ha il processore `php-fpm` che il nostro server web Nginx richiede per gestire l'elaborazione PHP. Inoltre, stiamo usando un'immagine basata su Alpine, che ridurrà la dimensione dell'immagine complessiva.
- `depends_on`: viene utilizzato per esprimere la dipendenza tra i servizi. La definizione del servizio `mysql` come dipendenza dal nostro container `drupal` assicurerà che quest'ultimo verrà creato solo dopo che il servizio `mysql` è stato avviato e consentirà alla nostra applicazione di essere eseguita senza problemi.
- `networks`: qui specifichiamo che il servizio verrà esposto tramite la rete `internal` e anche `external`; ciò garantirà che il servizio `mysql` sia accessibile solo dal container Drupal attraverso la rete interna, mantenendo tale container non accessibile ad altri container attraverso la rete esterna.
- `volumes`: stiamo montando un volume denominato `drupal-data` nella directory `/var/www/html` del container creato a partire dall'immagine Drupal. L'uso di un volume con nome in questo modo ci consentirà di condividere il nostro codice dell'applicazione con altri container.

Definiamo infine il servizio per l'avvio del server web Nginx.

Listato 9.33 Il file `./docker-compose.yml`.

```
...
webserver:
    image: nginx:1.17.4-alpine
    container_name: webserver
    depends_on:
        - drupal
    restart: unless-stopped
    ports:
        - 80:80
    volumes:
        - drupal-data:/var/www/html
        - ./nginx-conf:/etc/nginx/conf.d
        - certbot-etc:/etc/letsencrypt
    networks:
        - external
...

```

Ancora una volta, stiamo assegnando un nome al container (in questo caso, `webserver`) e lo stiamo rendendo dipendente dal container Drupal nell'ordine di avvio.

Nella definizione di questo servizio includiamo anche le seguenti opzioni.

- `ports`: questa sezione indica che si espone la porta 80 per abilitare le opzioni di configurazione che abbiamo definito nel nostro file `nginx.conf` all'inizio del paragrafo.
- `volumes`: stiamo definendo sia il volume denominato sia il percorso host.
 - `drupal-data:/var/www/html`: questo monterà il codice dell'applicazione Drupal nella directory specificata a destra, che abbiamo in precedenza impostato come cartella principale nel server Nginx.
 - `./nginx-conf:/etc/nginx/conf.d`: questa parte definisce in che modo si monterà la directory di configurazione Nginx sull'host nella relativa directory sul container, assicurando che eventuali modifiche apportate ai file sull'host si riflettano anche all'interno del container.
 - `certbot-etc:/etc/letsencrypt`: in questo volume verranno montati i certificati e le chiavi di *Let's Encrypt* pertinenti al nostro dominio nella directory appropriata sul *container*.
- `networks`: qui viene fatto riferimento alla sola rete esterna (definita `external`) per consentire a questo container di comunicare con il container Drupal e non con il container MySQL.

Infine, dobbiamo aggiungere un ultimo servizio per la gestione di Certbot. È importante sostituire `utente@tuo_dominio` e `tuo_dominio` con la giusta e-mail e il giusto nome di dominio.

Listato 9.34 Il file `./docker-compose.yml`.

```
...
certbot:
  depends_on:
    - webserver
  image: certbot/certbot
  container_name: certbot
  volumes:
    - certbot-etc:/etc/letsencrypt
    - drupal-data:/var/www/html
  command: certonly --webroot --webroot-path=/var/www/html --email
  utente@mio_dominio --agree-tos --no-eff-email --staging -d mio_dominio -d
  www.mio_dominio
...

```

Questa definizione specifica a Docker Compose di estrarre l'immagine `certbot/certbot` dal Docker Hub; utilizza inoltre i volumi con specificati per condividere le risorse con il container Nginx, inclusi i certificati di dominio e la chiave presente nella cartella `certbot-etc`, oltre al codice dell'applicazione presente in `drupal-data`.

Abbiamo anche utilizzato `depends_on`, per assicurarci che il container `certbot` venga avviato solo dopo l'esecuzione del servizio relativo a Nginx, così come abbiamo fatto per Drupal.

Non abbiamo specificato alcuna rete, perché questo container non comunicherà con nessun servizio sulla rete; aggiunge solo i certificati e la chiave del dominio, che abbiamo montato utilizzando i volumi con nome.

Abbiamo anche incluso l'opzione `command`, che specifica un'istruzione da eseguire con il comando `certbot` predefinito del container: dal momento che il client Certbot supporta plugin per ottenere e installare certificati, utilizziamo il plugin `webroot` per ottenere un certificato includendo `certonly` e `-webroot`.

Non rimane che definire la rete della soluzione e i volumi pocanzi specificati.

Listato 9.35 Il file `./docker-compose.yml`.

```
...
networks:
  external:
    driver: bridge
  internal:
    driver: bridge
volumes:
  drupal-data:
  db-data:
  certbot-etc:
```

La direttiva `networks` ci consente di specificare le reti da creare, le quali permettono le comunicazioni tra i servizi su tutte le porte, poiché si trovano sullo stesso host su cui viene eseguito il daemon Docker.

Abbiamo definito due reti, `internal` ed `external`, per proteggere la comunicazione tra i servizi `webserver`, `drupal` e `mysql`.

La proprietà relativa ai volumi viene utilizzata per definire i volumi denominati `drupal-data`, `db-data` e `certbot-etc`. Quando Docker crea i volumi, i contenuti del volume vengono archiviati in una directory sul file `system host`, ossia nella cartella `/var/lib/docker/volumes` (nel caso di un host Linux). Il contenuto di ciascun volume viene quindi montato a partire da questa

directory e reso disponibile a qualsiasi container che utilizza il volume. In questo modo, è possibile condividere codice e dati tra container, come nel caso del servizio `certbot` che accede al volume `drupal-data`.

Il file `docker-compose.yml`, finito, sarà simile al seguente.

Listato 9.36 Il file `./docker-compose.yml`.

```
version: "3"
services:
  mysql:
    image: mysql:8.0
    container_name: mysql
    command: --default-authentication-plugin=mysql_native_password
    restart: unless-stopped
    env_file: .env
    volumes:
      - db-data:/var/lib/mysql
    networks:
      - internal
  drupal:
    image: drupal:8.7.8-fpm-alpine
    container_name: drupal
    depends_on:
      - mysql
    restart: unless-stopped
    networks:
      - internal
      - external
    volumes:
      - drupal-data:/var/www/html
  webserver:
    image: nginx:1.17.4-alpine
    container_name: webserver
    depends_on:
      - drupal
    restart: unless-stopped
    ports:
      - 80:80
    volumes:
      - drupal-data:/var/www/html
      - ./nginx-conf:/etc/nginx/conf.d
      - certbot-etc:/etc/letsencrypt
    networks:
      - external
  certbot:
    depends_on:
      - webserver
    image: certbot/certbot
    container_name: certbot
    volumes:
      - certbot-etc:/etc/letsencrypt
      - drupal-data:/var/www/html
    command: certonly --webroot --webroot-path=/var/www/html --email
    utente@mio_dominio --agree-tos --no-eff-email --staging -d mio_dominio -d
    www.mio_dominio
    networks:
      external:
        driver: bridge
      internal:
```

```

        driver: bridge
volumes:
  drupal-data:
  db-data:
  certbot-etc:

```

Abbiamo finito con la definizione dei nostri servizi. Quindi avviamo il container e proviamo le richieste di certificati; possiamo farlo usando il comando `docker-compose up`, che creerà ed eseguirà i container che abbiamo definito poco fa nell'ordine specificato. Se le richieste del nostro dominio hanno esito positivo, vedremo nel terminale la buona riuscita dell'avvio dei servizi e i certificati che saranno presenti nella cartella `/etc/letsencrypt/live` sul container del server.

Se vedessimo qualcosa di diverso da `up` nella colonna `state` per i servizi `mysql`, `drupal` o `webserver` oppure uno stato di uscita diverso da `0` per il container `certbot`, dobbiamo controllare i log del servizio usando il comando `docker-compose logs`.

Per verificare la presenza dei certificati, eseguiamo il comando `docker exec` per il container `webserver` e verifichiamo che l'output sia simile al seguente.

Listato 9.37 Il comando docker exec.

```

root@vbox:/home# docker-compose exec webserver ls -la /etc/letsencrypt/live
total 16
drwx----- 3 root      root          4096 Sep  5 10:12 .
drwxr-xr-x  9 root      root          4096 Sep  5 10:12..
-rw-r--r--  1 root      root          740 Sep  5 10:12 README
drwxr-xr-x  2 root      root          4096 Sep  5 10:12 mio_dominio

```

Ora che tutto funziona correttamente, possiamo modificare la definizione del servizio `certbot` e rimuovere l'opzione `--staging`; apriamo dunque il file `docker-compose.yml` e sostituiamo, nel punto in cui avviene la definizione del servizio `certbot` il parametro `--staging` con il flag `--force-renewal`, che dirà a Certbot di richiedere un nuovo certificato con lo stesso dominio del certificato esistente. La definizione di `certbot` aggiornata sarà simile alla seguente.

Listato 9.38 Il file `./docker-compose.yml`.

```

...
certbot:
  depends_on:
    - webserver
  image: certbot/certbot
  container_name: certbot
  volumes:

```

*****ebook converter DEMO

Watermarks*****

```

- certbot-etc:/etc/letsencrypt
- drupal-data:/var/www/html

command: certonly --webroot --webroot-path=/var/www/html --email
utente@mio_dominio --agree-tos --no-eff-email --force-renewal -d mio_dominio -d
www.mio_dominio

```

Dobbiamo eseguire nuovamente `docker-compose up` per ricreare il container `certbot`, e specificheremo anche l'opzione `--no-deps` per saltare l'avvio del servizio server web, poiché è già in esecuzione.

Listato 9.39 Il comando `docker-compose up`.

```
root@vbox:/home# docker-compose up --force-recreate --no-deps certbot
```

Ora che abbiamo generato correttamente i nostri certificati, possiamo aggiornare la configurazione Nginx per includere SSL; dopo aver installato i certificati in Nginx, dovremo reindirizzare tutte le richieste HTTP su HTTPS. Inoltre dovremo specificare il certificato SSL e le posizioni delle chiavi, oltre che aggiungere i parametri e gli *header* di sicurezza nelle *request*.

Per farlo, è necessario arrestare il servizio `webserver` e modificare il file di configurazione specificando le informazioni necessarie all'utilizzo del protocollo HTTPS.

Listato 9.40 Il file `./nginx-conf/nginx.conf`.

```

server {
    listen 80;
    listen [::]:80;
    server_name mio_dominio www.mio_dominio;
    location ~ /.well-known/acme-challenge {
        allow all;
        root /var/www/html;
    }
    location / {
        rewrite ^ https://$host$request_uri? permanent;
    }
}
server {
    listen 443 ssl;
    listen [::]:443 ssl;
    server_name mio_dominio www.mio_dominio;
    index index.php index.html index.htm;
    root /var/www/html;
    server_tokens off;
    ssl_certificate /etc/letsencrypt/live/mio_dominio/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/mio_dominio/privkey.pem;
    add_header X-Frame-Options "SAMEORIGIN" always;
    add_header X-XSS-Protection "1; mode=block" always;
    add_header X-Content-Type-Options "nosniff" always;
    add_header Referrer-Policy "no-referrer-when-downgrade" always;
    add_header Content-Security-Policy "default-src * data: 'unsafe-eval' 'unsafe-inline'" always;
    location /

```

*****ebook converter DEMO

Watermarks*****

```

        try_files $uri $uri/ /index.php$is_args$args;
    }
    rewrite ^/core/authorize.php/core/authorize.php(.*)$ /core/authorize.php$1;
    location ~ \.php$ {
        try_files $uri =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass drupal:9000;
        fastcgi_index index.php;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param PATH_INFO $fastcgi_path_info;
    }
    location ~ /\.ht {
        deny all;
    }
    location = /favicon.ico {
        log_not_found off; access_log off;
    }
    location = /robots.txt {
        log_not_found off; access_log off; allow all;
    }
    location ~* \.(css|gif|ico|jpeg|jpg|js|png)$ {
        expires max;
        log_not_found off;
    }
}

```

La direttiva che fa riferimento al server `http` (porta 80) rimane sostanzialmente invariata; nel blocco relativo al server HTTPS (porta 443) viene abilitato il protocollo SSL e HTTP2.

Queste istruzioni abilitano SSL, poiché abbiamo incluso il nostro certificato SSL e le chiavi negli header tramite le istruzioni `ssl_certificate` e `add_header`. Prima di avviare nuovamente il container del server Nginx, dovremo aggiungere il mapping della porta 443 alla definizione del servizio, per rendere effettive le modifiche fatte.

Modifichiamo dunque il file `docker-compose.yml` come segue.

Listato 9.41 Il file `./docker-compose.yml`.

```

...
webserver:
  image: nginx:1.17.4-alpine
  container_name: webserver
  depends_on:
    - drupal
  restart: unless-stopped
  ports:
    - 80:80
    - 443:443
  volumes:
    - drupal-data:/var/www/html
    - ./nginx-conf:/etc/nginx/conf.d
    - certbot-etc:/etc/letsencrypt
  networks:
    - external

```

*****ebook converter DEMO

Watermarks*****

...
A questo punto, non resta che riavviare il servizio `webserver` e verificare tramite `docker-compose ps` che tutti i container siano attivi; per completare l'installazione di Drupal, possiamo utilizzare l'interfaccia grafica fornita che possiamo raggiungere tramite https://mio_dominio.

I certificati di *Let'sEncrypt* sono validi solo per un periodo, quindi è importante impostare un processo di rinnovo automatico per garantire che non scadano. Un modo per farlo è quello di creare un *cron job*; in questo caso, creeremo uno script che verrà eseguito periodicamente per rinnovare i certificati e ricaricare la configurazione Nginx.

Creiamo quindi il file `ssl_renew.sh` per rinnovare i certificati e aggiungiamo il seguente codice, andando però a sostituire nella terza riga la cartella *utente* con il nome utente presente nel sistema host.

Listato 9.42 Il file `ssl_renew.sh`

```
#!/bin/bash
cd /home/utente/drupal/
/usr/local/bin/docker-compose -f docker-compose.yml run certbot renew --dry-run &&
\
/usr/local/bin/docker-compose -f docker-compose.yml kill -s SIGHUP webserver
```

Questo script entra nella directory del progetto `drupal` ed esegue i comandi specificati.

- `docker-compose run`: avvierà un container `certbot` e sovrascriverà il comando fornito nella nostra definizione del servizio `certbot`. Invece di utilizzare `certonly`, qui utilizziamo `renew`, che rinnoverà i certificati che stanno per scadere.
- `docker-compose kill`: invierà un segnale `SIGHUP` al container del server Nginx, per ricaricare la relativa configurazione.

Dopo aver assegnato i permessi di esecuzione allo script, lo inseriamo nella crontab, modificando il file tramite il comando seguente.

Listato 9.43 Il comando `crontab`.

```
root@vbox:/home# sudo crontab -e
```

Alla fine del file, aggiungiamo la seguente riga, sostituendo a *utente* il nostro nome utente.

Listato 9.44 Il comando crontab.

```
*/10 * * * * /home/utente/drupal/ssl_renew.sh >> /var/log/cron.log 2>&1
```

Questo imposterà il *cron job* ogni dieci minuti, in modo da poter verificare se la richiesta di rinnovo dei certificati ha funzionato come previsto. Abbiamo anche creato un *file log*, denominato `cron.log`, per registrare eventuali output prodotti dall'esecuzione dello script.

Quest'ultimo passaggio non è obbligatorio; è tuttavia consigliato applicare un procedimento simile per evitare di dover rinnovare manualmente i certificati prima della scadenza, o anche di farli scadere e avere problemi con l'applicazione.

Gestione di un sito Wordpress

WordPress è uno dei software di gestione dei contenuti (*CMS*) più popolari, grazie alla molitudine di funzionalità che offre e alla sua facilità d'uso; tuttavia, la creazione di un ambiente host può richiedere molto tempo, soprattutto se è necessario farlo spesso. Semplificare il processo di installazione con pochi comandi riduce notevolmente il tempo e gli sforzi richiesti, ed è qui che entra in gioco Docker.

L'installazione di WordPress con Docker è un gioco da ragazzi: con pochi e semplici passi, saremo in grado di mettere in piedi un sito di base, avendo a disposizione una soluzione a container.

Prima di installare WordPress con Docker, dovremo avere un luogo in cui archiviare i dati. MariaDB è un sistema di gestione di database relazionali e un valido sostituto per MySQL. È ufficialmente disponibile per Docker e presente sul Docker Hub con un'immagine costantemente aggiornata.

Iniziamo quindi creando una nuova cartella dove salveremo i file per WordPress e MariaDB, denominata `wordpress`.

Il download e l'installazione di un container MariaDB possono essere eseguiti con un solo comando; vediamo quindi quali parametri sostituire al comando che eseguiremo e i loro significati.

Le variabili d'ambiente `MariaDB` sono precedute dall'opzione `-e` nel comando Docker e sono le seguenti.

- `MYSQL_ROOT_PASSWORD`: password con cui accedere al database.
- `MYSQL_DATABASE`: creazione di un nuovo database.

Inoltre, specificheremo diversi parametri, come il nome del container, che potremo usare successivamente come alias, e un volume, che verrà utilizzato per mantenere le informazioni relative al database.

Lanciamo dunque il comando seguente, sostituendo i parametri in modo opportuno.

Listato 9.45 Avvio del container mariadb.

```
root@vbox:/home# docker run -e MYSQL_ROOT_PASSWORD=<password> -e MYSQL_DATABASE=<nome-db> --name wordpressdb -v "$PWD/database":/var/lib/mysql -d mariadb:latest
```

Se Docker è riuscito a creare il container, possiamo eseguire il comando `docker ps` per confermare che il container MariaDB sia in esecuzione.

Così come MariaDB, anche WordPress è ufficialmente disponibile su Docker Hub, e quindi possiamo estrarre la relativa immagine usando il comando `docker pull`. Così come abbiamo fatto prima, eseguiremo un unico comando, impostando tutti i parametri necessari all'avvio di Wordpress, come la variabile d'ambiente per l'accesso al database, la creazione di un link tra i due container (questa operazione può essere fatta in modo analogo creando due servizi tramite Docker Compose), specificando l'esposizione della porta alla quale l'utente potrà accedere e definendo un volume che renda disponibile al di fuori del container i file di Wordpress.

Listato 9.46 Avvio del container wordpress.

```
root@vbox:/home# docker run -e WORDPRESS_DB_PASSWORD=<password> --name wordpress -link wordpressdb:mysql -p 80:80 -v "$PWD/html":/var/www/html -d wordpress
```

A questo punto possiamo collegarci al dominio tramite l'indirizzo IP del server per sottoporre a test l'installazione, e se tutto è stato eseguito correttamente, verremo reindirizzati alla pagina iniziale di installazione di WordPress su `http://<dominio>wp-admin/install.php`.

Così come per un'installazione in locale senza l'aiuto di Docker, sarà sufficiente seguire l'installazione guidata e completare la configurazione di Wordpress direttamente tramite browser.

Creazione di un'applicazione MEAN

MEAN è un tipo di stack di sviluppo web molto popolare, composto da MongoDB, Express, Angular e Node.js. MEAN ha guadagnato popolarità perché consente agli sviluppatori di programmare in JavaScript sia sul client sia sul server; consente infatti una perfetta armonia di sviluppo utilizzando oggetti JSON, perfetti per MongoDB. Express e Node.js facilitano invece la creazione di query JSON e Angular consente al client di inviare e ricevere senza problemi documenti JSON.

MEAN viene generalmente utilizzato per creare applicazioni web basate su browser, poiché Angular (lato client) ed Express (lato server) sono entrambi framework per app web; un altro caso d'uso tipico dello stack MEAN è lo sviluppo di server API RESTful. La creazione di servizi API RESTful è diventata un'attività di sviluppo sempre più importante e comune, poiché le applicazioni hanno sempre più bisogno di supportare opportunamente un'ampia varietà di dispositivi per utenti finali, come smartphone e tablet. In questo progetto vedremo come utilizzare lo stack MEAN per creare rapidamente un server che funga da API RESTful.

Angular, come framework lato client, non è un componente necessario per la creazione di un server API. Includiamo Angular in questo progetto per dimostrare come creare rapidamente un'applicazione web che viene eseguita sopra al server API.

Quella che svilupperemo in questo progetto è un'applicazione di base per la gestione dei contatti, che supporta le operazioni *CRUD* (inserisci, leggi, modifica, cancella) standard. Innanzitutto, creeremo un server API RESTful che fungerà da interfaccia per l'interrogazione e la memorizzazione dei dati in un database MongoDB e quindi utilizzeremo il server API per creare un'applicazione web basata su Angular che fornisce un'interfaccia per gli utenti finali.

Per prima cosa, definiamo il database nel quale verranno salvate le informazioni relative alla gestione di alcuni libri; creiamo dunque tre cartelle dove gestiremo in maniera separata il front-end, il back-end e il database, che chiameremo rispettivamente `frontend`, `backend` e `db`.

Nella cartella del database, creiamo un file Dockerfile e aggiungiamo il contenuto seguente.

Listato 9.47 || file ./db/Dockerfile.

```
# immagine di base
FROM mysql:5
# aggiungo variabili d'ambiente
# nome database
ENV MYSQL_DATABASE mean
# password del database
ENV MYSQL_ROOT_PASSWORD=password
# inizializzazione db

COPY ./sql-scripts/ /docker-entrypoint-initdb.d/
```

Questo creerà un server MySQL con la password di root; inoltre, è possibile creare una directory chiamata `sql-script` per creare al momento dell'esecuzione le tabelle iniziali con dei valori precompilati, di cui riportiamo di seguito gli script.

Listato 9.48 || file ./db/create.sql.

```
CREATE TABLE books(title varchar(100), description text);
```

Listato 9.49 || file .db/insert.sql.

```
INSERT INTO books (title, description) VALUES ('The Chronicles of Narnia', 'The Chronicles of Narnia is a series of fantasy novels by British author C. S. Lewis. Written by Lewis, illustrated by Pauline Baynes, and originally published in London between 1950...');
```

A questo punto definiremo la parte relativa al back-end e faremo affidamento su Express come framework per comunicare con il driver di MySQL e per eseguire query sul database. Definiamo dunque il file `package.json`, nel quale descriviamo tutte le dipendenze che ci servono per lavorare. Poi definiremo due file, `app.js` e `api.js`: il primo conterrà la logica dell'applicazione Express, mentre la seconda conterrà le funzionalità delle API.

Listato 9.50 || file ./backend/package.json

```
{
  "name": "backend",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "body-parser": "~1.18.3",
    "cors": "^2.8.5",
    "express": "~4.16.4",
    "mysql": "^2.16.0"
  }
}
```

Listato 9.51 || file ./backend/app.js.

```
const express = require('express');
const path = require('path');
const http = require('http');
const bodyParser = require('body-parser');
const api = require('.api');
const app = express();
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use('/', api);
const port = process.env.PORT || '3000';
app.set('port', port);
const server = http.createServer(app);
server.listen(port, () => console.log(`backend running on port:${port}`));
```

Listato 9.52 || file ./backend/api.js.

```
const express = require('express');
const router = express.Router();
const mysql = require('mysql');
const cors = require('cors');
var con = mysql.createConnection({
  host: "database",
  user: "root",
  port: '3306',
  password: "password",
  database: "mean",
  charset : 'utf8'
});
var corsOptions = {
  origin: '*',
  optionsSuccessStatus: 200
}
con.connect(function(err) {
  if(err) console.log(err);
});
router.get('/books', cors(corsOptions), (req, res) => {
  con.query("SELECT * FROM books", function (err, result, fields) {
    if (err) res.send(err);
    res.send(result);
    console.log(result);
  });
});
module.exports = router;
```

Ciò che è importante notare, qui, è che abbiamo chiamato il nostro host del database `database`. Questo collegamento verrà infatti utilizzato più avanti nel file `docker-compose.yml`, in cui collegheremo il servizio `database` al servizio `back-end`, in modo che il back-end sia in grado di interagire come descritto.

Ora che abbiamo creato il nostro back-end, semplifichiamo la dockerizzazione creando innanzitutto un Dockerfile per questo servizio.

Listato 9.53 Il file ./backend/Dockerfile.

```
# immagine di base
FROM node
# creazione della cartella di lavoro
RUN mkdir -p /usr/src/app
# set della cartella di lavoro
WORKDIR /usr/src/app
# copia del file delle dipendenze
COPY package.json /usr/src/app
# installazione delle dipendenze
RUN npm install
# copia del codice
COPY . /usr/src/app
# esposizione della porta
EXPOSE 3000
# avvio dell'app

CMD ["npm", "start"]
```

In seguito, quando creeremo la nostra immagine `docker back-end`, per non includere nell'immagine tutta la cartella `node_modules`, creeremo un file `.dockerignore` che elenca questa directory tra quelle da escludere.

Listato 9.54 Il file ./backend/.dockerignore.

```
node_modules/
```

Creiamo ora il progetto Angular: all'interno della cartella principale, eseguiamo il comando `ng g frontend`, il quale creerà una cartella `frontend` con la struttura di base del progetto.

Concentriamoci semplicemente su come interrogare il nostro `back-end`: per fare ciò, creeremo un servizio `BooksService` che disporrà di un metodo per interrogare l'API `/books` e restituire un elenco di libri presenti nel database.

Listato 9.55 Il file ./frontend/src/app/services/books.service.ts.

```
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: 'root'
})
export class BooksService {
  constructor(private http: HttpClient) { }
  getBooks (): Observable<Book[]> {
    return this.http.get<Book[]>("http://localhost:3000/books");
  }
}
```

Questo metodo potrà essere richiamato all'interno di qualsiasi componente che importi il servizio; in questo modo, potremo utilizzare le

API in maniera estremamente semplice, anche tramite l'applicazione web.

Il Dockerfile per il servizio `frontend` somiglia molto a quello relativo al `backend` poiché entrambi usano un'immagine di base `node`; l'unica differenza è che esponiamo il nostro `frontend` sulla porta 4200. Non dimentichiamo inoltre di creare il file `.dockerignore`, che includa la cartella `node_modules` per i motivi prima descritti.

Listato 9.56 Il file `./frontend/Dockerfile`.

```
# immagine di base
FROM node
# creazione della cartella di lavoro
RUN mkdir -p /usr/src/app
# set della cartella di lavoro
WORKDIR /usr/src/app
# copia del file delle dipendenze
COPY package.json /usr/src/app
# installazione delle dipendenze
RUN npm install
# copia del codice
COPY . /usr/src/app
# esposizione della porta
EXPOSE 4200
# avvio dell'app
CMD ["npm", "start"]
```

Ora che abbiamo tutti i servizi definiti con i rispettivi Dockerfile, possiamo configurare il file `docker-compose.yml` che porremo nella cartella principale del progetto; creeremo tre servizi distinti nel nostro stack *MEAN*, e ognuno di loro ha un proprio Dockerfile. Tramite la proprietà `build` individuiamo la cartella in cui si trova il Dockerfile e definiamo anche la gestione delle porte, creando inoltre un collegamento tra il `backend` e il `database`, così che quest'ultimo sia avviato prima che il `backend` venga reso disponibile.

Listato 9.57 Il file `./docker-compose.yml`.

```
version: '2'
services:
  frontend:
    build: frontend
    ports:
      - "4200:4200"
    container_name: frontend
    restart: always
  backend:
    build: backend
    ports:
      - "3000:3000"
    container_name: backend
    restart: always
```

```
links:  
  - database  
database:  
  build: database  
  container_name: database  
  restart: always
```

Ora, per creare le immagini e lanciare i container, non ci resta che eseguire il comando `docker-compose up --build -d` e controllare lo stato dei container con il comando `docker container ps`; quando l'app sarà in esecuzione, potremo accedervi utilizzando il `frontend` e inserendo nel browser l'indirizzo `localhost:4200`.

Come possiamo vedere, creare uno stack MEAN *dockerizzato* è molto semplice; la potenza di un esempio simile sta nel fatto che separa l'applicazione usando un approccio molto simile a quello dei microservizi, strutturando il progetto in modo che ogni servizio abbia il proprio Dockerfile; in questo caso, le tecnologie scelte si prestano molto bene al caso d'uso e il progetto è molto semplice, ma può essere facilmente usato come base per attività più complesse. Utilizzando Docker Compose è infatti possibile eseguire un'app composta da più container, creando collegamenti tra servizi in qualunque momento.

Uso di Apache Kafka

Apache Kafka è nato da un'idea di *LinkedIn*, la quale stava affrontando diversi problemi per via della bassa latenza rispetto alla richiesta di enormi quantità di dati da parte della piattaforma. Come soluzione, nel 2010 è stato sviluppato Apache Kafka, poiché nessuna delle soluzioni al momento disponibili era adatta ad affrontare quel tipo di problematica. Successivamente è stato donato alla Apache Software Foundation. Kafka è scritto in Scala e Java ed è un sistema di messaggistica ad alta tolleranza d'errore basato sul paradigma PubSub (*Publisher and Subscriber*). È veloce, scalabile e distribuito in base alla progettazione.

Gli sviluppatori usano Apache Kafka quando hanno bisogno di gestire la comunicazione tra produttori e consumatori utilizzando argomenti basati su messaggi. Una delle migliori caratteristiche di Kafka è la sua alta disponibilità e la resilienza agli errori, con supporto al ripristino automatico. Questa funzione rende Apache Kafka ideale per la comunicazione e l'integrazione tra componenti di sistemi di dati su larga scala.

Inoltre, questa tecnologia sostituisce i broker di messaggi convenzionali, con la capacità di offrire *throughput*, affidabilità e replica; inoltre, l'astrazione principale Kafka offre diversi componenti, come un broker Kafka, un produttore Kafka e un consumatore Kafka. Il *broker Kafka* è un nodo che fa parte del cluster Kafka, e il suo utilizzo è quello di salvare e replicare i dati; un *produttore Kafka* invia il messaggio al container dei messaggi chiamato Kafka Topic; il *consumatore Kafka* estrae il messaggio dal topic e lo consuma.

Prima di procedere in questo progetto, diamo uno sguardo al sistema di messaggistica in Kafka; quando trasferiamo dati da un'applicazione a un'altra, utilizziamo il sistema di messaggistica. In questo modo, senza preoccuparsi di come condividere le informazioni, le applicazioni possono concentrarsi solo sull'elaborazione. I messaggi vengono messi in coda in modo asincrono tra le applicazioni client e il sistema di messaggistica e sono disponibili due tipi di schemi di messaggistica: *point-to-point* e *pubblicazione-sottoscrizione* (pub-sub), di cui vedremo l'implementazione a breve; un esempio è rappresentato nella Figura 9.4.

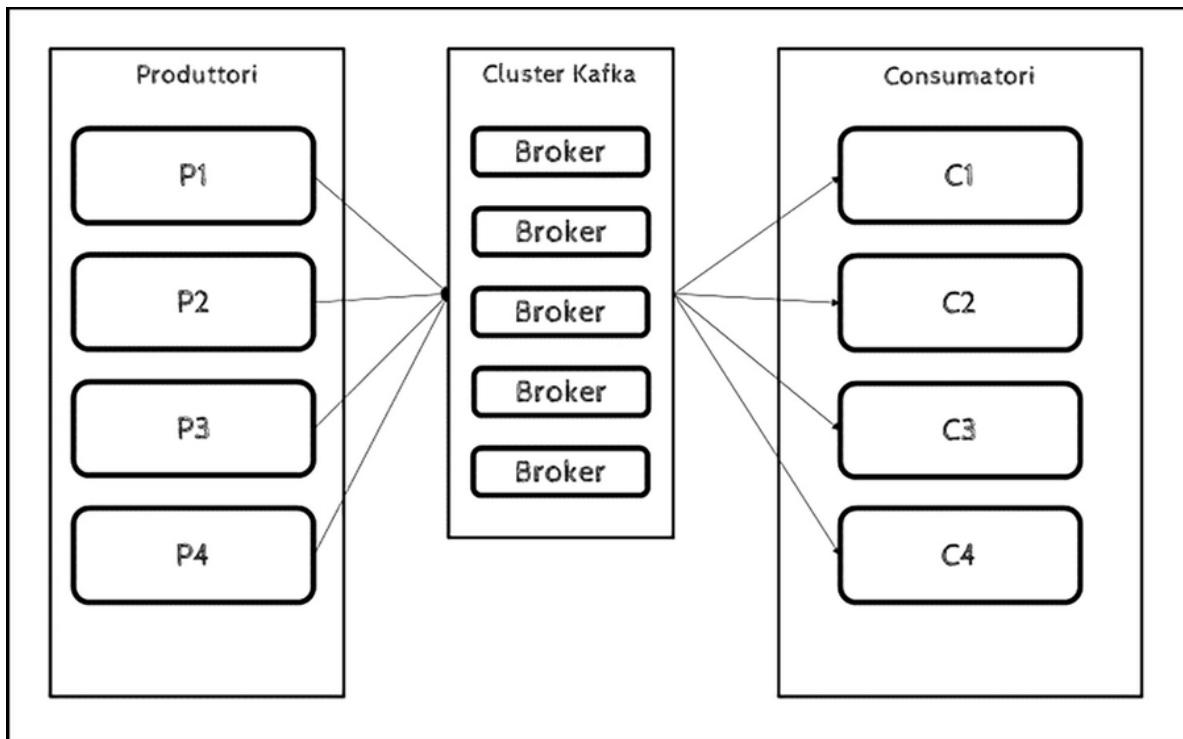


Figura 9.4 Esempio di cluster Kafka.

I produttori pubblicano dati sugli argomenti di loro scelta; i gruppi di consumatori possono iscriversi a uno o più topic. Ognuno di questi gruppi può essere configurato con più consumatori.

Kafka memorizza flussi di record (messaggi) nei topic. Ogni record è costituito da una chiave, un valore e la data. I produttori scrivono dati nei topic e i consumatori li ricevono e leggono.

Un *topic* è una categoria in cui vengono pubblicati i record; può avere zero, uno o più consumatori, che si abbonano ai dati scritti su di esso. Per ogni argomento, il cluster Kafka mantiene un registro partizionato. Poiché Kafka è un sistema distribuito, gli argomenti vengono partizionati e replicati su più nodi.

Ogni messaggio in un topic viene recapitato a una delle istanze del consumatore all'interno del gruppo sottoscritto a tale argomento. Tutti i messaggi con la stessa chiave arrivano allo stesso consumatore.

Ora vediamo come configurare Kafka con Docker con un esempio molto semplice: utilizzeremo due immagini Docker, che ci serviranno per mettere in piedi la soluzione, ossia le immagini `wurstmeister/kafka` e `wurstmeister/zookeeper`, scaricabili da Docker Hub.

Come prima cosa, cloniamo o scarichiamo il progetto ufficiale di Kafka, presente all'URL <https://github.com/wurstmeister/kafka-docker.git> e modifichiamo il file `docker-compose.yml`: la variabile d'ambiente `KAFKA_ADVERTISED_HOST_NAME` andrà impostata con un indirizzo IP valido, per esempio `172.17.0.1`.

Listato 9.58 Il file `./docker-compose.yml`.

```
version: '2'
services:
  zookeeper:
    image: wurstmeister/zookeeper:3.4.6
    expose:
    - "2181"
  kafka:
    image: wurstmeister/kafka:2.11-2.0.0
    depends_on:
    - zookeeper
    ports:
    - "9092:9092"
    environment:
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
```

Analizzando nel dettaglio il file fornito all'interno del repository ufficiale appena scaricato, notiamo che sono definiti due servizi, ossia

`zookeeper` e `kafka`. Del secondo abbiamo già parlato, ma il primo potrebbe suonare nuovo. In realtà, Zookeeper è un software di alto livello sviluppato da Apache che funge da servizio centralizzato e viene utilizzato per conservare i dati di configurazione e per fornire una sincronizzazione flessibile e solida all'interno dei sistemi distribuiti. Zookeeper tiene traccia dello stato dei nodi del cluster Kafka e anche dei topic, delle partizioni di Kafka e così via.

Zookeeper consente a più client di eseguire letture e scritture simultanee e funge da servizio di configurazione condiviso all'interno del sistema. Il protocollo *Zookeeper Atomic Broadcast* (ZAB) è il cervello di tutto il sistema, che rende possibile a Zookeeper di comportarsi come un sistema di trasmissione atomica ed emettere aggiornamenti ordinati.

Se eseguiamo il comando `docker-compose up -d` con la configurazione attuale, avremmo a disposizione un solo broker e un'istanza di Zookeeper; possiamo decidere di scalare i broker con il seguente comando, impostandoli a 5.

Listato 9.59 Il comando per scalare i broker.

```
root@vbox:/home# docker-compose scale kafka=5
```

Una volta che i container sono stati tutti avviati (verifichiamolo lanciando il comando `docker-compose ps`), accediamo alla shell del container in SSH e lanciamo il seguente comando.

Listato 9.60 Il comando per avviare Kafka.

```
$ ./start-kafka-shell.sh <DOCKER_HOST_IP>
```

Per verificare che l'installazione e la configurazione siano andate a buon fine, proviamo a creare un *topic* e ad avviare un produttore, con i seguenti comandi.

Listato 9.61 Il comando per avviare Kafka.

```
$ $KAFKA_HOME/bin/kafka-topics.sh --create --topic test \
--partitions 4 --replication-factor 2 \
--bootstrap-server `broker-list.sh` \
$ $KAFKA_HOME/bin/kafka-topics.sh --describe --topic test \
--bootstrap-server `broker-list.sh`
```

Il file `broker-list.sh` non è altro che uno *script bash* che definisce due variabili, `CONTAINERS` e `BROKERS`, e assegna loro un valore recuperando i container attivi al momento dell'esecuzione del comando sulla porta 9092; con il primo comando eseguiamo quindi lo script `kafka-topics.sh` con

l'opzione `-create`, che ci permette di creare un topic di nome `test` con 4 partizioni e un fattore di replicazione pari a 2.

Nel secondo comando recuperiamo invece le informazioni sui broker, e in particolare il dettaglio delle partizioni e dei vari nodi presenti nel cluster; questo avviene eseguendo sempre lo stesso script, ma passando come opzione aggiuntiva `--describe`. Il risultato dovrebbe essere simile al seguente.

```
Topic:test      PartitionCount:4      ReplicationFactor:2      Configs:  
Topic: test      Partition: 0      Leader: 0      Replicas: 0      Isr: 0
```

Inizializziamo ora un produttore e un consumatore, eseguendo i due comandi riportati.

Listato 9.62 Creazione di un produttore e di un consumatore.

```
$ $KAFKA_HOME/bin/kafka-console-producer.sh --topic=test \  
--broker-list=`broker-list.sh`  
>> Hello World!  
>> I'm the producer  
$ $KAFKA_HOME/bin/kafka-console-consumer.sh --topic=test \  
--from-beginning --bootstrap-server `broker-list.sh`
```

Con il primo comando, grazie allo script `kafka-console-producer.sh`, creiamo un produttore con cui inviamo messaggi; nel secondo comando invece abbiamo creato un consumatore tramite lo script `kafka-console-consumer.sh`, che riceverà tutti i messaggi inviati nel *topic* `test`.

Se eseguiamo ciascuno dei comandi sopra riportati in terminali diversi, dovremmo essere in grado di digitare i messaggi nel terminale del produttore e vederli apparire nel terminale del consumatore.

Come potremo vedere, tutto funziona perfettamente quando sia il produttore sia il consumatore si trovano in una rete. Prestiamo attenzione alla variabile d'ambiente `KAFKA_ADVERTISED_LISTENERS`, impostata nel file `docker-compose.yml`: Kafka invia il valore di questa variabile ai client durante la loro connessione. Dopo aver ricevuto quel valore, i client lo usano per inviare e consumare record con il broker Kafka.

Poiché abbiamo definito la variabile come `PLAINTEXT://localhost:9092`, sia il produttore sia il consumatore l'hanno ricevuta sulla connessione iniziale e l'hanno utilizzata per ulteriori comunicazioni con il broker attraverso la porta 9092.

La chiave nel funzionamento di questo esempio è che in questo caso i client usano l'indirizzo Kafka specificato (ossia i valori di `--bootstrap-`

server e --broker-list) solo per la connessione iniziale. Kafka li reindirizza al valore specificato nella variabile `KAFKA_ADVERTISED_LISTENERS`, che i client utilizzano per produrre e consumare record.

Finora abbiamo eseguito i vari comandi all'interno di terminali in esecuzione sui container; se invece volessimo poter esporre il servizio al di fuori dello stesso, dovremo modificare il file `docker-compose.yml` aggiungendo delle variabili d'ambiente che permettano di esporre i broker Kafka sul sistema host.

Vediamo alcune delle variabili d'ambiente che imposteremo a breve.

- `KAFKA_ADVERTISED_LISTENERS`: riporta l'elenco di indirizzi disponibili che punta al broker Kafka. Kafka li invierà ai client sulla loro connessione iniziale.
- `KAFKA_LISTENERS`: contiene l'elenco degli indirizzi (0.0.0.0:9093, 0.0.0.0:9092) e i nomi dei *listener* (`INSIDE`, `OUTSIDE`) su cui il broker Kafka ascolterà le connessioni in entrata.
- `KAFKA_LISTENER_SECURITY_PROTOCOL_MAP`: associa i nomi di *listener* definiti sopra (ossia `INSIDE` e `OUTSIDE`) al protocollo Kafka.
- `KAFKA_INTER_BROKER_LISTENER_NAME`: punta a un nome di *listener* che verrà utilizzato per la comunicazione tra broker.

In questo caso, definiremo due *listener* (`INSIDE: //0.0.0.0: 9093, OUTSIDE: //0.0.0.0: 9092`) rispettivamente per il traffico interno nella rete Docker e il traffico esterno dalla macchina host.

In breve, abbiamo definito i due tipi di client Kafka - esterno e interno - e configurato Kafka per inviare loro indirizzi diversi sulle loro connessioni iniziali.

Listato 9.63 Il file `./docker-compose.yml`.

```
...
kafka:
  image: wurstmeister/kafka:2.11-2.0.0
  depends_on:
    - zookeeper
  ports:
    - "9092:9092"
  expose:
    - "9093"
  environment:
    KAFKA_ADVERTISED_LISTENERS: INSIDE://kafka:9093,OUTSIDE://localhost:9092
    KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: INSIDE:PLAINTEXT,OUTSIDE:PLAINTEXT
    KAFKA_LISTENERS: INSIDE://0.0.0.0:9093,OUTSIDE://0.0.0.0:9092
```

*****ebook converter DEMO

Watermarks*****

```
KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181  
KAFKA_INTER_BROKER_LISTENER_NAME: INSIDE
```

Proviamo quindi a inviare un messaggio dal produttore.

Listato 9.64 Il file `./docker-compose.yml`.

```
$KAFKA_HOME/bin/kafka-console-producer.sh --broker-list kafka:9093 --topic test  
>Hi there!
```

Se il consumatore lo riceve correttamente, vuol dire che la configurazione è andata a buon fine. In questo caso, abbiamo sostanzialmente aggiunto delle modifiche per poter esporre i broker Kafka con il computer host; chiaramente, è impossibile trattare qui tutte le potenzialità di Kafka: abbiamo visto solo la punta dell'iceberg di quel che è possibile fare con questa tecnologia. Come sappiamo, al giorno d'oggi c'è un'enorme quantità di dati: i cosiddetti *Big Data*; quando si tratta di Big data, ci sono due sfide principali: una è quella di raccogliere tutto questo volume di dati, e l'altra è quella di analizzare i dati raccolti. Per superare tali sfide, abbiamo bisogno di un sistema distribuito come quello appena accennato; Kafka permette infatti di tracciare le attività web, memorizzando e inviando gli eventi dei vari processi in tempo reale, oppure di trasformare i dati secondo uno standard, avendo un'elaborazione continua di dati in streaming sui vari topic.

Questa tecnologia sta entrando in forte concorrenza con alcune delle applicazioni più popolari, come ActiveMQ, RabbitMQ e via dicendo, proprio grazie al suo ampio utilizzo.

Configurazione di un cluster Hadoop

Apache Hadoop è un popolare framework per Big Data molto utilizzato nel settore del software. Come sistema distribuito, Hadoop funziona su cluster che vanno dal singolo nodo a migliaia di nodi; è un progetto della Apache Software Foundation e offre due elementi importanti:

- *un file system distribuito*, chiamato HDFS (*Hadoop Distributed File System*);

- un framework e delle API per la creazione e l'esecuzione di attività chiamate *MapReduce*.

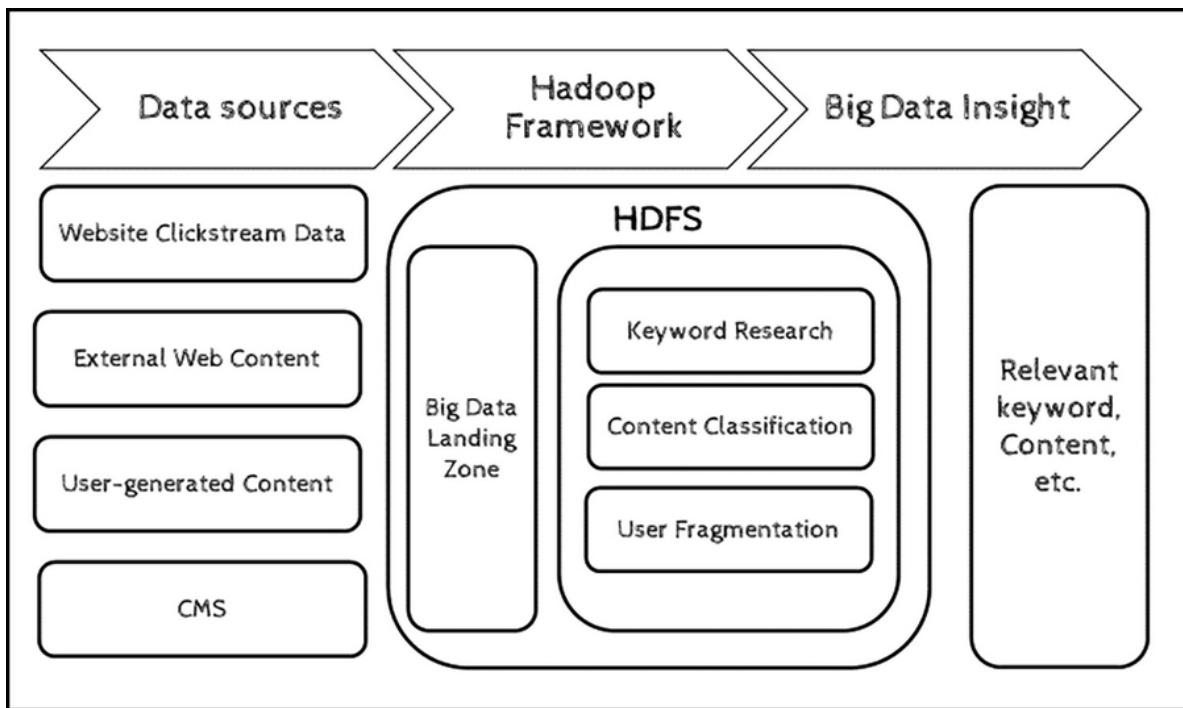


Figura 9.5 L'architettura Hadoop.

HDFS è strutturato in modo simile a un normale file system Unix, tranne per il fatto che l'archiviazione dei dati è distribuita su più macchine. Non intende sostituire un normale file system, ma piuttosto intende rappresentare un livello *simile* a un file system, che può essere utilizzato da grandi sistemi distribuiti. Dispone di meccanismi integrati per gestire le interruzioni della macchina ed è ottimizzato per la velocità effettiva anziché per la latenza.

La seconda parte fondamentale di Hadoop è *MapReduce*, che è costituito da due sottocomponenti:

- un'API per la scrittura di flussi di lavoro *MapReduce* in Java;
- un insieme di servizi per la gestione dell'esecuzione di questi flussi di lavoro.

La vera potenza di queste API sta nel fatto che non esiste alcuna dipendenza tra due elementi dello stesso compito. Per fare il proprio lavoro, un'attività `map()` non ha bisogno di conoscere altre attività del suo stesso tipo e allo stesso modo una singola attività `reduce()` ha tutto il contesto necessario per aggregare su una particolare chiave, e non condivide nessuno stato con altre attività di riduzione.

Nel complesso, questo progetto rappresenta una distribuzione perfettamente gestibile delle diverse fasi della pipeline di un sistema complesso. I flussi di lavoro che richiedono insiemi di dati di grandi dimensioni possono essere facilmente distribuiti su centinaia di macchine, perché non vi sono dipendenze intrinseche tra le attività che si trovano sulla stessa macchina.

Per sottoporre a test Hadoop o se al momento non avessimo accesso a una grande rete di cluster, possiamo impostare un cluster Hadoop sul sistema locale, usando Docker.

In questo progetto, configureremo con Docker un cluster Hadoop a tre nodi ed eseguiremo il classico programma Hadoop di conteggio delle parole, col solo scopo di sottoporre a test il sistema.

Per installare Hadoop in un container Docker, è necessario avere un'immagine di Hadoop. Per generare l'immagine, utilizzeremo il repository *Big Data Europe*; se Git è installato nel sistema, lanciamo il seguente comando, altrimenti dovremo scaricare il file `.zip` dal repository specificato.

Listato 9.65 Clonazione del progetto.

```
$ git clone git@github.com:big-data-europe/docker-hadoop.git
```

Una volta che abbiamo terminato di scaricare il repository, dovremo modificare il file `docker-compose.yml` per abilitare alcune porte di ascolto e cambiare il punto in cui *Docker Compose* estrae le immagini, nel caso in cui esse siano già state scaricate. Apriamo quindi il file `docker-compose.yml` e sostituiamo il contenuto con il seguente.

Listato 9.66 Il file `./docker-compose.yml`.

```
version: "2"
services:
  namenode:
    build: ./namenode
    image: bde2020/hadoop-namenode:1.1.0-hadoop2.7.1-java8
    container_name: namenode
    volumes:
```

```

        - hadoop_namenode:/hadoop/dfs/name
environment:
    - CLUSTER_NAME=test
env_file:
    - ./hadoop.env
ports:
    - "9870:50070"
resourcemanager:
    build: ./resourcemanager
    image: bde2020/hadoop-resourcemanager:1.1.0-hadoop2.7.1-java8
    container_name: resourcemanager
    depends_on:
        - namenode
        - datanode1
        - datanode2
    env_file:
        - ./hadoop.env
    ports:
        - "8089:8088"
historyserver:
    build: ./historyserver
    image: bde2020/hadoop-historyserver:1.1.0-hadoop2.7.1-java8
    container_name: historyserver
    depends_on:
        - namenode
        - datanode1
        - datanode2
    volumes:
        - hadoop_historyserver:/hadoop/yarn/timeline
    env_file:
        - ./hadoop.env
nodemanager1:
    build: ./nodemanager
    image: bde2020/hadoop-nodemanager:1.1.0-hadoop2.7.1-java8
    container_name: nodemanager1
    depends_on:
        - namenode
        - datanode1
        - datanode2
    env_file:
        - ./hadoop.env
datanode1:
    build: ./datanode
    image: bde2020/hadoop-datanode:1.1.0-hadoop2.7.1-java8
    container_name: datanode1
    depends_on:
        - namenode
    volumes:
        - hadoop_datanode1:/hadoop/dfs/data
    env_file:
        - ./hadoop.env
datanode2:
    build: ./datanode
    image: bde2020/hadoop-datanode:1.1.0-hadoop2.7.1-java8
    container_name: datanode2
    depends_on:
        - namenode
    volumes:
        - hadoop_datanode2:/hadoop/dfs/data
    env_file:
        - ./hadoop.env

```

*******ebook converter DEMO**
Watermarks*****

```

datanode3:
  build: ./datanode
  image: bde2020/hadoop-datanode:1.1.0-hadoop2.7.1-java8
  container_name: datanode3
  depends_on:
    - namenode
  volumes:
    - hadoop_datanode3:/hadoop/dfs/data
  env_file:
    - ./hadoop.env
volumes:
  hadoop_namenode:
  hadoop_datanode1:
  hadoop_datanode2:
  hadoop_datanode3:
  hadoop_historyserver:

```

In questo file definiamo i componenti principali di una struttura *Hadoop*, tra cui il `namenode`, ossia il nodo principale che gestirà l’HDFS, tre nodi *slave* (`datanode1`, `datanode2` e `datanode3`), un gestore delle risorse YARN (`resourcemanager`), un server per tenere traccia delle attività (`historyserver`) e un gestore dei nodi (`nodemanager1`).

I cluster Hadoop sono composti da tre diversi tipi di nodi: nodi master, nodi di lavoro e nodi client. Sapere a che cosa servono i diversi tipi di nodo ci aiuterà a pianificare il cluster e a configurare il numero e il tipo di nodi appropriati durante la creazione del cluster.

I *nodi master* sovrintendono alle seguenti operazioni chiave, che vengono svolte tramite Hadoop: memorizzazione dei dati nel *HDFS* ed esecuzione di calcoli paralleli su tali dati utilizzando *MapReduce*. *NameNode* coordina la funzione di archiviazione dei dati (con HDFS), mentre *JobTracker* supervisiona e coordina l’elaborazione parallela dei dati utilizzando *MapReduce*.

I *nodi di lavoro* costituiscono la maggior parte delle macchine virtuali e svolgono il lavoro di archiviazione dei dati ed esecuzione dei calcoli. Ogni nodo di lavoro esegue sia un servizio *DataNode* sia un *TaskTracker*, con i quali comunica; il servizio *TaskTracker* è subordinato al *JobTracker* e il servizio *DataNode* è subordinato al *NameNode*.

I *nodi client* hanno Hadoop installato con tutte le impostazioni del cluster, ma non sono né nodi master né nodi di lavoro. Il nodo client carica i dati nel cluster, si occupa dei lavori *MapReduce*, descrivendo come tali dati debbano essere elaborati, quindi recupera o visualizza i risultati del lavoro al termine dell’elaborazione.

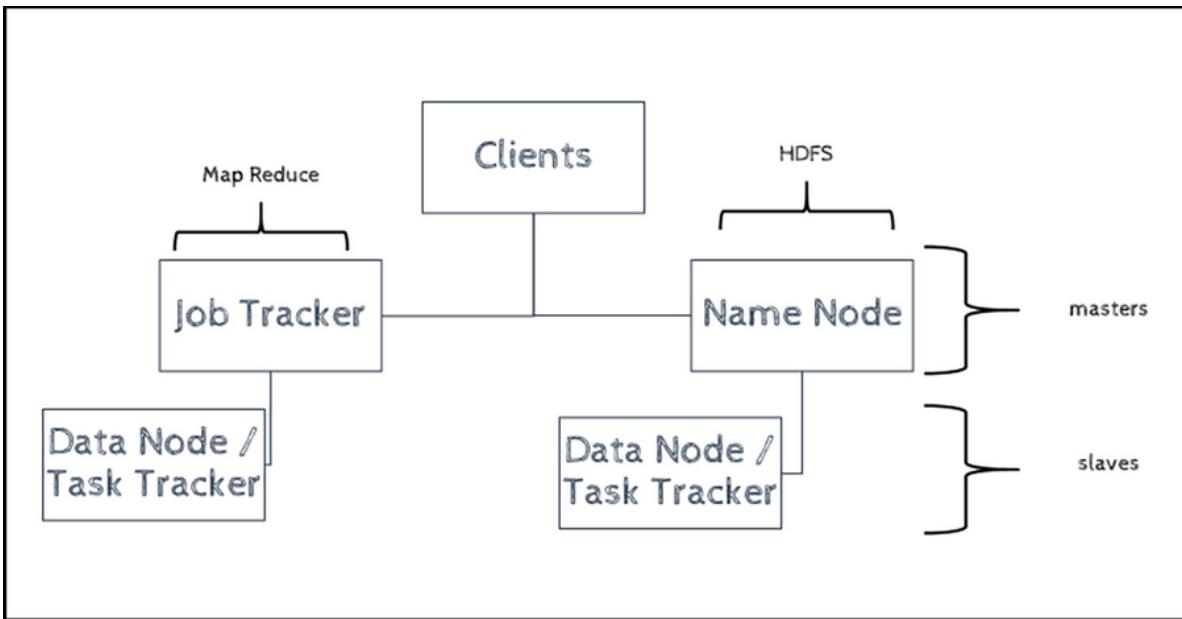


Figura 9.6 Rappresentazione dei ruoli dei diversi nodi.

Docker Compose provvederà a estrarre le immagini dal registro Docker Hub, se le immagini non sono disponibili localmente, a creare le immagini e ad avviare i container con il comando `docker-compose up -d`. Possiamo utilizzare il comando `docker ps` per verificare la presenza di container attualmente in esecuzione e andare su `http://localhost:9870` per visualizzare lo stato corrente del sistema dal `namenode`.

Ora possiamo sottoporre a test il cluster Hadoop, eseguendo il classico programma di conteggio delle parole; accediamo al terminale del container `namenode` lanciando il seguente comando.

Listato 9.67 Avvio del terminale di namenode.

```
$ docker exec -it namenode bash
```

Una volta entrati, creeremo alcuni semplici file di testo di input, per inserirli nel programma WordCount; li chiameremo `file1.txt` e `file2.txt` e li collocheremo in una cartella chiamata `input`. Nei file possiamo inserire frasi a scelta. Per esempio, inseriamo “Hello Docker” nel primo file e “Hello world” nel secondo.

Ora creiamo la directory di input su HDFS, tramite il seguente comando.

Listato 9.68 Creazione cartella input.

```
$ hadoop fs -mkdir -p input
```

*****ebook converter DEMO

Watermarks*****

Inseriamo i file su tutti i `datanode` presenti, usando il seguente comando:

Listato 9.69 Copia dei file sui datanode.

```
$ hdfs dfs -put ./input/* input
```

Scarichiamo il programma di conteggio parole da qualsiasi sito web e salviamolo in una cartella. Ora dobbiamo copiare il programma *WordCount* dal computer locale nel container `namenode`. Usiamo il comando `docker container ls` per scoprire l'ID del container e inseriamolo nel seguente comando, sostituendo il path del file `.jar` con quello in cui è stato salvato il file scaricato in precedenza.

Listato 9.70 Copia del programma nel container.

```
$ docker cp ..../hadoop-mapreduce-examples-2.7.1-sources.jar 123abcdefg:hadoop-mapreduce-examples-2.7.1-sources.jar
```

Ora che il programma è stato copiato, possiamo eseguirlo lanciando la seguente istruzione dal terminale del container.

Listato 9.71 Esecuzione del programma WordCount.

```
root@namenode:/# hadoop jar hadoop-mapreduce-examples-2.7.1-sources.jar  
org.apache.hadoop.examples.WordCount input output
```

Per vedere il risultato dell'esecuzione del programma, possiamo lanciare il comando `hdfs dfs` con l'opzione `-cat`, che ne stampa il contenuto.

Listato 9.72 Output dell'esecuzione del programma.

```
root@namenode:/# hdfs dfs -cat output/part-r-00000  
World 1  
Docker 1  
Hello 2
```

Se il risultato sarà simile al precedente, congratulazioni: abbiamo appena configurato un semplice cluster Hadoop!

Che cosa abbiamo imparato

- Come configurare con Docker Compose un'applicazione composta da Laravel, Nginx e MySQL, installando e definendo le impostazioni di base di ciascun componente, scrivendo un Dockerfile personalizzato e avviando i container come servizi.

- Come creare un'applicazione che funga da API REST tramite Express.js, per recuperare dati da un database MongoDB.
- Come creare un'applicazione tramite la libreria Flask di Python, per esporre delle API REST e recuperare dei dati da un database SQLite.
- Come installare Drupal tramite Docker Compose e come configurarlo affinché sia raggiungibile in rete locale.
- Come installare e gestire un sito Wordpress.
- Come creare un'applicazione utilizzando uno stack MEAN, sfruttando Docker Compose per gestire e avviare i diversi servizi.
- Come funziona Apache Kafka, quali servizi offre e come sfruttarlo per creare un servizio di pub-sub tramite diversi container.
- Come funziona Apache Hadoop, quali sono i diversi tipi di nodi previsti e come creare un cluster per eseguire il conteggio delle parole.

Kubernetes

Le grandi aziende di Internet, come Google, Twitter ed eBay, hanno utilizzato architetture basate su container per anni; mentre da un lato i container, secondo quanto visto finora, rappresentano una soluzione elegante per creare applicazioni basate su microservizi, Docker non include tutto il necessario per distribuire e gestire più container che devono lavorare insieme e ampliare la propria scala all'aumentare della domanda.

Quando queste aziende ha iniziato a creare i propri strumenti di sviluppo, e a gestirne l'implementazione e il ridimensionamento, si sono rese conto che l'impegno era troppo grande perché Docker potesse gestire tutto il carico.

Google ha rilasciato *Kubernetes* come progetto *open source*, con l'obiettivo di standardizzare la gestione delle applicazioni a container; Kubernetes, spesso abbreviato in *K8S*, funge da overlay rispetto al data center di un'azienda e i diversi progetti potrebbero lavorare sulla risoluzione di altri aspetti della gestione dei container (come la creazione di sistemi operativi semplificati o strumenti di amministrazione grafica).

NOTA

Perché Kubernetes viene abbreviato in K8S? I cosiddetti *numeronimi* sono apparsi alla fine degli anni Ottanta. Ci sono molte storie su come le persone abbiano effettivamente iniziato a usarli; tutte condividono la stessa idea che per semplificare la comunicazione, il settore IT abbia iniziato a utilizzare i *numeronimi* per abbreviare le parole: prendendo la prima e l'ultima lettera di una parola e mettendo in mezzo un numero che ne desse più o meno il suono corretto, riduceva di molto il lavoro. Per esempio, il termine *i18n* deriva da *internationalization*: inizia con la lettera "i", più 18 lettere, più la lettera "n". Analogamente, Kubernetes è composta dalla lettera "k", più 8 lettere, più la lettera "s".

Nel luglio 2015, Kubernetes è stato rilasciato in versione 1.0, indicando con ciò che il progetto era pronto per essere utilizzato in

produzione; Google ha in questo modo lanciato e costituito la *Cloud Native Computing Foundation* (CNCF), creando K8S come progetto collaborativo in associazione con la *Linux Foundation* (<http://collabprojects.linuxfoundation.org>). Le aziende che aderiscono al CNCF comprendono Red Hat, eBay, AT&T, Cisco, IBM, Intel, Twitter, Mesosphere, VMware e molte altre.

L’obiettivo di questo capitolo è quello di illustrare le funzionalità di Kubernetes e di descrivere come può migliorare l’intera esperienza di distribuzione dei container Docker in un *data center*. Utilizzare Kubernetes dà l’opportunità di usare gli stessi strumenti che oggi usano le più grandi aziende nel mondo che distribuiscono applicazioni basate su container, tenendo fede a tutte le premesse in materia di resilienza dell’architettura o del servizio, di robustezza e di applicazione delle *best practice* viste e applicate finora.

Kubernetes, fondamentalmente, è un sistema per l’esecuzione e il coordinamento di applicazioni a container in un cluster di macchine. È una piattaforma progettata per gestire completamente il ciclo di vita di applicazioni e servizi “containerizzati”, utilizzando metodi che offrono prevedibilità, scalabilità e alta disponibilità.

NOTA

Il nome *Kubernetes* deriva da un’antica parola greca che sta per “timoniere” (chi dirige la nave, per esempio una nave *portacontainer*) e ritroviamo questo nome nel logo del progetto, che mostra il timone di una nave. Il timone ha sette raggi, in omaggio al nome originale del progetto; Beda, McLuckie e il suo team inizialmente intendevano chiamare la tecnologia Kubernetes “Project Seven”. Il suo nome è un omaggio a *Seven of Nine*, personaggio di “Star Trek” proveniente dal gruppo dei Borg.

Come utenti che utilizzano Kubernetes, possiamo definire in che modo debbano funzionare le applicazioni e come dovrebbero poter interagire con altre applicazioni o con il mondo esterno. Possiamo ridimensionare i servizi, eseguire aggiornamenti continui e trasferire il traffico tra diverse versioni delle applicazioni, per sottoporre a test funzionalità o rollback di distribuzioni problematiche. K8S fornisce interfacce componibili che consentono di definire e gestire le applicazioni con elevati livelli di flessibilità, potenza e affidabilità.

La creazione di un singolo container, da eseguire per uso personale, non richiede molte risorse né qualche tipo di pianificazione; abbiamo visto quanto sia semplice avviare un servizio Docker ed eseguire alcuni comandi per avviarlo e renderlo disponibile. Al contrario, la creazione di

container su cui fare affidamento per fornire servizi più complessi in modo sicuro, affidabile, aggiornabile e scalabile comporta tutta una nuova serie di sfide.

Per esempio, attività come le seguenti rappresentano comuni operazioni che le aziende devono svolgere per produrre dei servizi, e sono tutte attività che richiedono dispendio di tempo ed energie con le sole tecnologie viste finora.

- *Distribuire un'applicazione che includa più servizi* (come un server web, un database e un server di autenticazione) che si desidera conservare in container separati, ma che comunque devono interagire tra loro.
- *Tenere traccia dei servizi forniti da un container*, in modo che un altro container sappia dove trovare i servizi di cui ha bisogno (come un server web che vuole sapere dove si trova il database per recuperare i dati da esporre).
- *Configurare il sistema per replicare la soluzione con dei container aggiuntivi* in caso di arresto anomalo del container o se la richiesta supera la capacità dei container esistenti.
- *Gestire centralmente i container che girano su un set di sistemi host*.
- *Duplicare i container su altri sistemi host*, in modo da poter arrestare un host per manutenzione, garantendo comunque la continuità del servizio.

Kubernetes presenta un nuovo modo di pensare alla creazione e gestione delle applicazioni, dallo sviluppo alla messa in produzione. Alcuni dei vantaggi che si hanno utilizzando questa tecnologia sono i seguenti.

- *Operare secondo il modello DevOps*: Kubernetes cerca di far parte di un modello più grande, chiamato *DevOps Movement*, in cui comunicazioni più strette tra gli sviluppatori di applicazioni e il personale IT rendono possibile il *continuous deployment*, senza causare interruzioni nel *data center* in uso. Nel modello DevOps, gli sviluppatori di software hanno una maggiore responsabilità nel creare tutto il software di cui un'applicazione ha bisogno, piuttosto

che affidarsi agli amministratori di sistema per installare componenti compatibili sul sistema host.

- *Creazione di insiemi di servizi comuni*: oggi le applicazioni spesso richiedono un servizio appartenente a un'altra applicazione, avendo a disposizione un indirizzo IP e un numero di porta. Con Kubernetes, è possibile definire applicazioni a container che forniscono servizi disponibili per altri container; Kubernetes gestisce infatti i dettagli relativi alle connessioni tra il *container* che fornisce il servizio e quello che lo richiede.
- *Rendere i computer host più generici*: invece di configurare i sistemi operativi in modo diverso a seconda delle applicazioni, i computer host possono essere più duttili. Pertanto, ogni organizzazione all'interno di un'azienda non deve disporre di un proprio set di computer da distribuire e gestire; al contrario, i servizi di un'organizzazione vengono eseguiti sullo stesso pool di macchine fisiche utilizzate da tutti gli altri membri dell'azienda, ma sempre restando separati dai servizi di altre organizzazioni, tramite l'adozione di un diverso set di etichette che ne identifica l'attività.
- *Stabilizzazione del data center*: Kubernetes mira a creare API coerenti che si traducono in ambienti stabili per l'esecuzione di applicazioni tramite container. Gli sviluppatori possono così essere in grado di creare applicazioni che funzionino su qualsiasi provider cloud che supporti tali API. Questa tecnologia consente agli sviluppatori di identificare la versione di Kubernetes e i servizi di cui hanno bisogno e di non doversi preoccupare della particolare configurazione del data center.

I prossimi paragrafi descrivono le funzionalità di Kubernetes che è possibile utilizzare in questo momento, grazie anche agli strumenti che abbiamo imparato a utilizzare nei capitoli precedenti; questo capitolo tratterà però Kubernetes in modo molto pratico, lasciando eventuali approfondimenti teorici o implementativi alla documentazione disponibile online.

Mentre Docker gestisce entità come le immagini e i container, Kubernetes racchiude tali entità in *pod*, concetto che vedremo nel dettaglio in seguito; al momento, possiamo dire che un *pod* può contenere uno o più container ed è il componente fondamentale gestito da

Kubernetes. L'utilizzo di elementi come i pod garantisce una serie di semplificazioni nello svolgere determinate attività rispetto alla gestione individuale dei singoli container, come le seguenti.

- *Gestione di più nodi*: invece di distribuire un container su un singolo host, Kubernetes può distribuire un insieme di pod su più nodi. In sostanza, un nodo fornisce l'ambiente in cui viene eseguito un container.
- *Replicazione*: Kubernetes può fungere da controller di replicazione per un *pod*. Ciò significa che è possibile impostare il numero di un pod specifico che deve essere sempre in esecuzione. Kubernetes valuterà se impostare un numero più elevato di repliche, se quello fornito dall'utente non soddisfa il requisito funzionale.
- *Definizione dei servizi*: la parola “servizio” viene utilizzata in molti contesti informatici, e ne abbiamo visto un esempio tramite Docker Compose. Per quanto riguarda Kubernetes, il termine “servizio” indica che è possibile assegnare un nome (più precisamente, un *ID*) a un determinato indirizzo IP, e quindi assegnare un *pod* per fornire quel servizio. Kubernetes tiene traccia internamente della posizione di quel servizio, così da sapere come indirizzare qualsiasi richiesta proveniente da un altro *pod* verso quel servizio all'indirizzo e alla porta corretti.

NOTA

Kubernetes nasce come strumento Open Source e ha una community molto attiva per quanto riguarda la manutenzione e la gestione dei registri pubblici e privati, oltre che della documentazione. Infatti, più di 1400 collaboratori di diverse aziende come Red Hat, Google e Microsoft fanno parte dell'elenco di collaboratori *stabili* del progetto Kubernetes. Recentemente, Alibaba e Amazon sono diventate anche loro tra le grandi aziende a utilizzare questa tecnologia. La fondazione del *cloud computing* supervisiona comunque questa tecnologia nel suo complesso. Inoltre, aziende leader come Intel, Mozilla, Pivotal, Oracle e molte altre stanno contribuendo con il codice a questa tecnologia fortemente infrastrutturale.

Architettura

In questo capitolo vedremo molti concetti che ci aiuteranno a muoverci nel mondo K8S, alcuni dei quali saranno fondamentali per i prossimi

paragrafi. Per esempio, i concetti riportati di seguito rappresentano i mattoni su cui si basa Kubernetes, ne costituiscono l'architettura di base.

Per capire come Kubernetes è in grado di fornire queste funzionalità, è utile avere un'idea di come è progettato e organizzato ad alto livello; Kubernetes può essere visualizzato come un sistema integrato in livelli: ogni livello superiore sottrae la complessità riscontrata nei livelli inferiori.

Alla sua base, Kubernetes riunisce singole macchine fisiche o virtuali in un cluster, utilizzando una rete condivisa per comunicare tra ciascun server; questo cluster è la piattaforma fisica in cui sono configurati tutti i componenti, le capacità e i carichi di lavoro di Kubernetes.

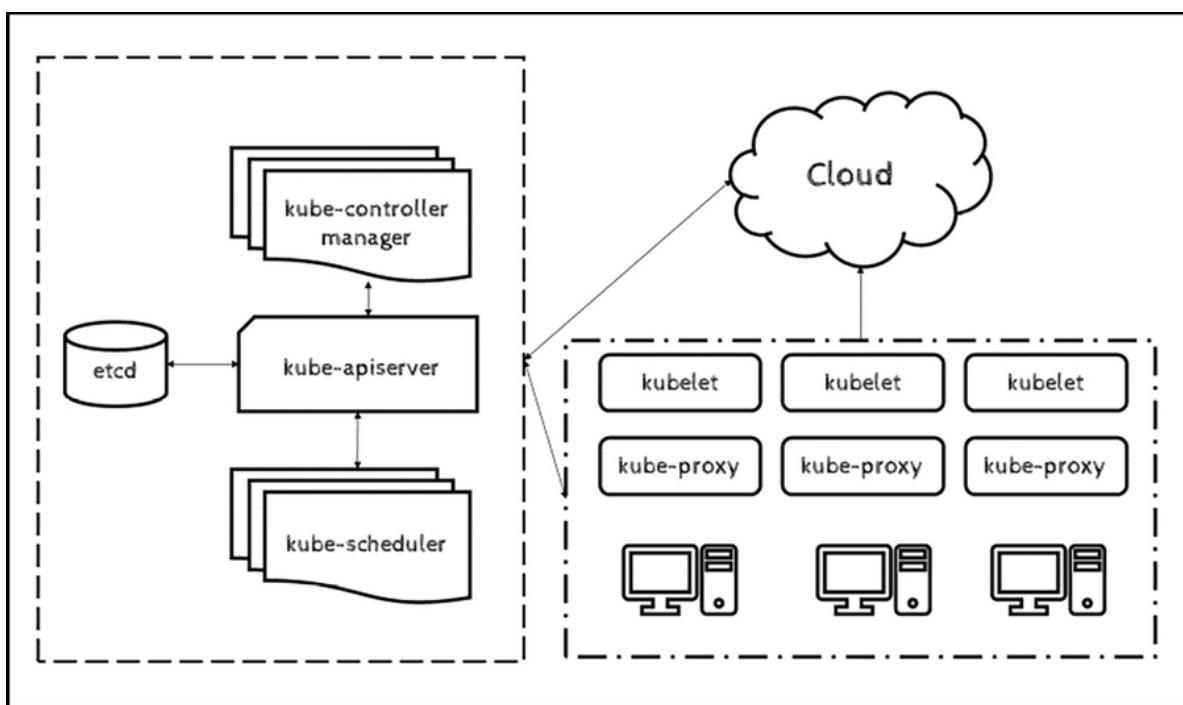


Figura 10.1 L'architettura di K8S.

A ciascuna macchina del cluster viene assegnato un ruolo all'interno dell'ecosistema Kubernetes: un *server* (o un piccolo gruppo in distribuzioni a elevata disponibilità) funziona come un server principale, ed è quello che nella Figura 10.1 si trova sulla sinistra, il cosiddetto *Kubernetes Master*. Questo server funge da gateway e oltre che da “cervello” del cluster, esponendo una serie di API per utenti e client e verificando l'integrità degli altri server, oppure decidendo il modo migliore per suddividere e assegnare il lavoro e orchestrando la

comunicazione tra altri componenti. Il server principale funge da punto di contatto primario con il cluster ed è responsabile della maggior parte della logica centralizzata fornita da Kubernetes.

Le altre macchine del cluster sono designate come nodi (a destra nella Figura 10.1) definiti come *Kubernetes minions*, sono i server responsabili dell'accettazione e dell'esecuzione dei carichi di lavoro utilizzando le risorse locali ed esterne. Per aiutare nelle attività di isolamento, gestione e flessibilità, Kubernetes esegue applicazioni e servizi in container, quindi ogni nodo deve essere dotato di un'istanza del container. Il nodo riceve le istruzioni di lavoro dal server principale e crea o distrugge i container di conseguenza, configurando le regole di rete per instradare e inoltrare il traffico in modo appropriato.

Come accennato in precedenza, le applicazioni e i servizi vengono eseguiti sul cluster all'interno di container. I componenti sottostanti assicurano che lo stato desiderato delle applicazioni corrisponda allo stato effettivo del cluster. Gli utenti interagiscono con il cluster comunicando con il server API principale direttamente o con client e librerie. Per avviare un'applicazione o un servizio, viene inviato un piano dichiarativo in formato JSON o YAML, che definisce che cosa creare e come dovrebbe essere gestito il tutto. Il server principale utilizza questo “piano” e procede nell'esecuzione sull'infrastruttura esaminando i requisiti e lo stato corrente del sistema. Questo gruppo di applicazioni definite dall'utente in esecuzione secondo un piano specificato rappresenta il livello finale di Kubernetes.

Componenti di base

Mantenere una buona astrazione dei componenti K8s può essere complicato. Vedremo dunque come questi elementi chiave si incastrano in modo perfetto per riuscire a comporre questa potente piattaforma.

Per prima cosa esamineremo i sei strati di astrazione e le parti che li compongono; poi vedremo altri sette oggetti chiave delle API di K8s. Supponendo di avere un'applicazione che non abbia necessità di memorizzare informazioni sul proprio stato, utilizzando K8S si potrebbe rappresentare la sua astrazione secondo lo schema rappresentato nella Figura 10.2.

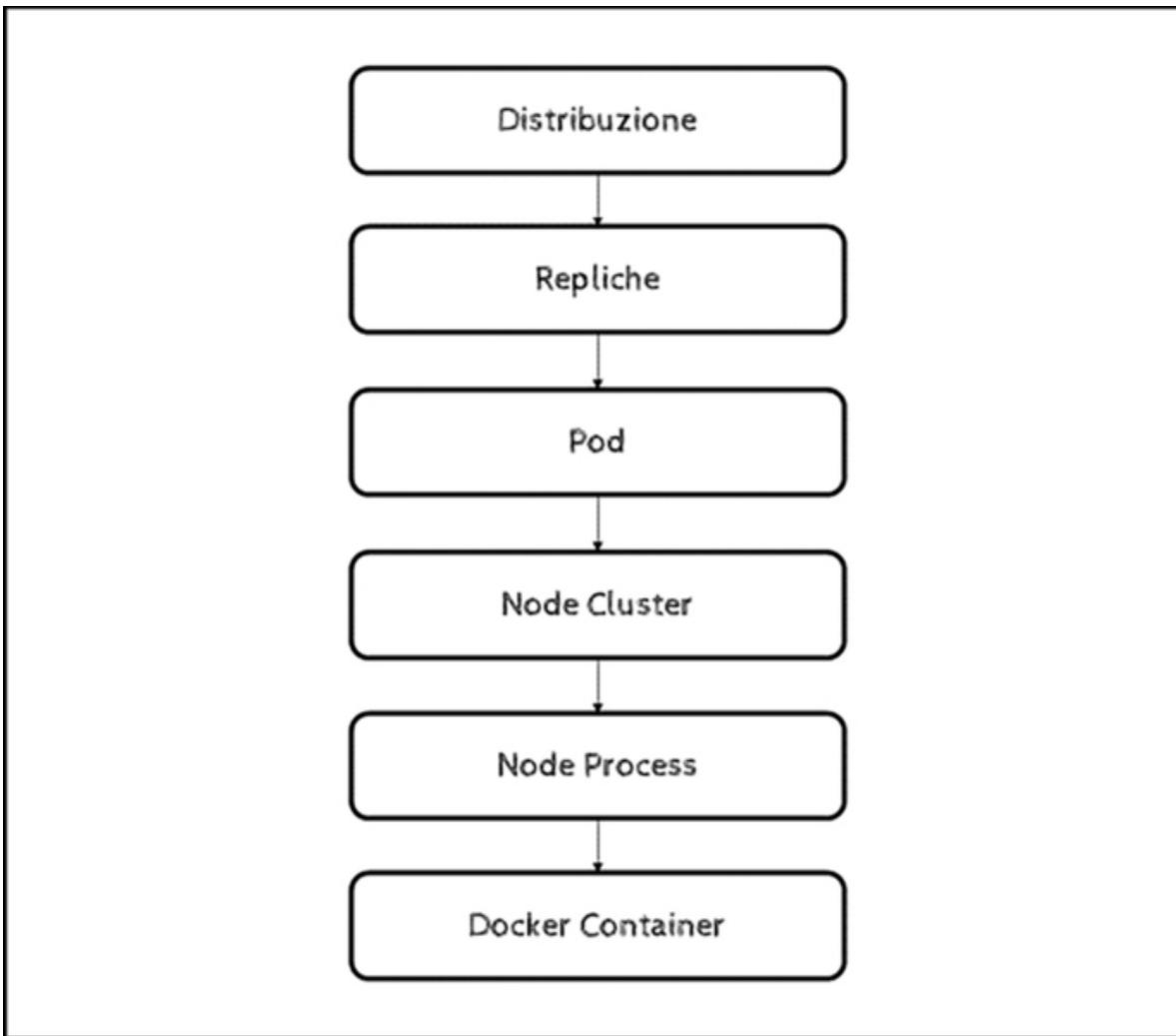


Figura 10.2 I sei livelli di astrazione.

Le distribuzioni creano e gestiscono i set di repliche, i quali creano e gestiscono i *pod*, i quali a loro volta eseguono i *nodi*, che sono creati a partire da *processi*, che contengono *container*... Per quanto la spiegazione possa suonare come la *Fiera dell'Est* di Angelo Branduardi, in realtà questo schema è molto rappresentativo rispetto al funzionamento dei vari oggetti in K8S: come abbiamo visto anche in precedenza, i nodi gestiscono diversi elementi, tra cui *kubelet*, *kube-proxy* e i *container* in esecuzione; potremmo dunque rivedere lo schema precedente e aggiornarlo nel modo rappresentato nella Figura 10.3.

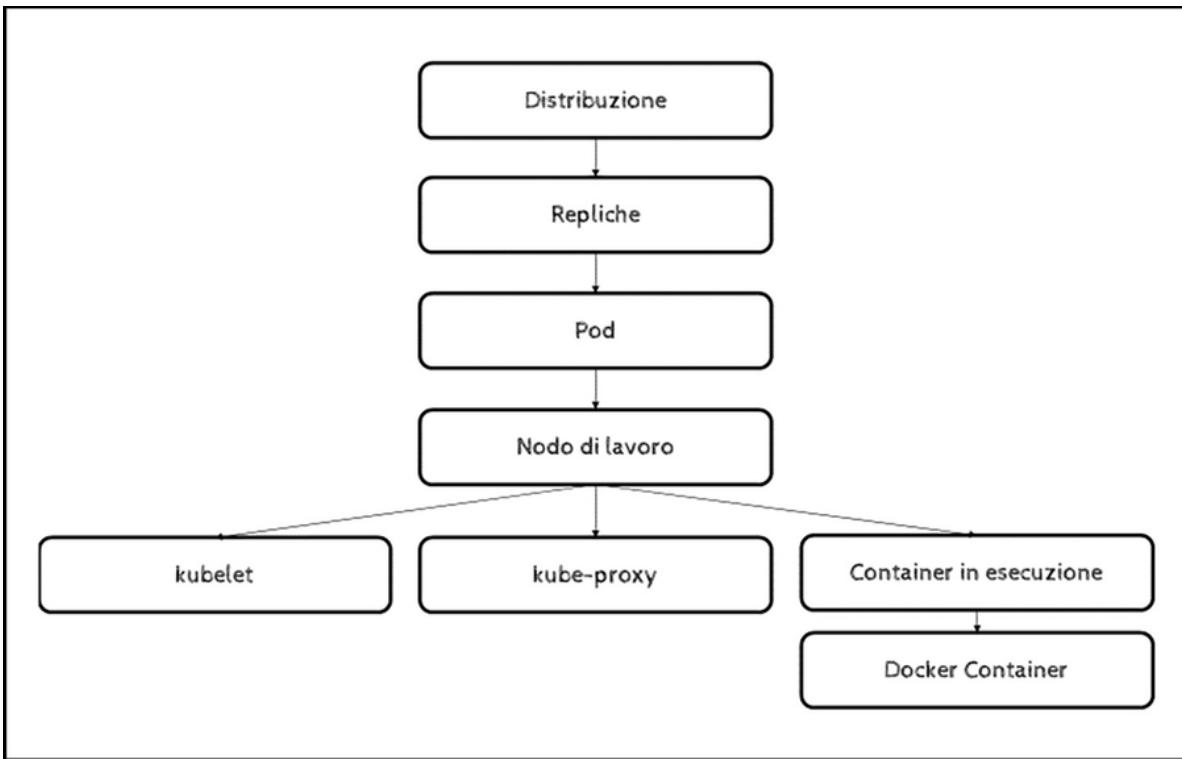


Figura 10.3 I sei livelli di astrazione con i sottolivelli del nodo di lavoro esplicitati.

A livello generico, possiamo vedere questi elementi pensando che...

- Se desideriamo creare un'applicazione senza stato che verrà eseguita in modo continuo, per esempio un server http, sarà necessario creare una distribuzione. Queste consentono infatti di aggiornare un'applicazione in esecuzione senza introdurre tempi di inattività.
- La distribuzione può creare un insieme di repliche che garantirà che il sistema disponga di un dato numero di *pod* e questo stesso si occuperà di crearli o ridimensionarli in base a quanto specificato dalla distribuzione.
- Il *pod* è l'elemento più atomico, ed è quello che contiene uno o più *container*, anche se, per motivi che vedremo a breve, normalmente ne contiene uno. Questi oggetti si interfacciano con volumi e *container* e, come questi ultimi, sono effimeri.

Il concetto di cluster integra tutti questi oggetti: il *cluster*, genericamente parlando, è composto da un master e una serie di nodi di

lavoro: questi ultimi spesso sono abbreviati in nodi, e sono l’astrazione più vicina a una macchina; possono contenere uno o più pod, e seguono le indicazioni ricevute dal master.

Il *master* viene chiamato in moltissimi modi: Kubernetes master, nodo master o anche solo master; in ogni caso, il master gestisce i nodi di lavoro. I componenti principali di quest’ultimo li vedremo a breve, ma possiamo riassumerli nello schema della Figura 10.4; ognuno di questi gestisce attività indipendenti, che servono a coordinare il lavoro dei vari nodi.

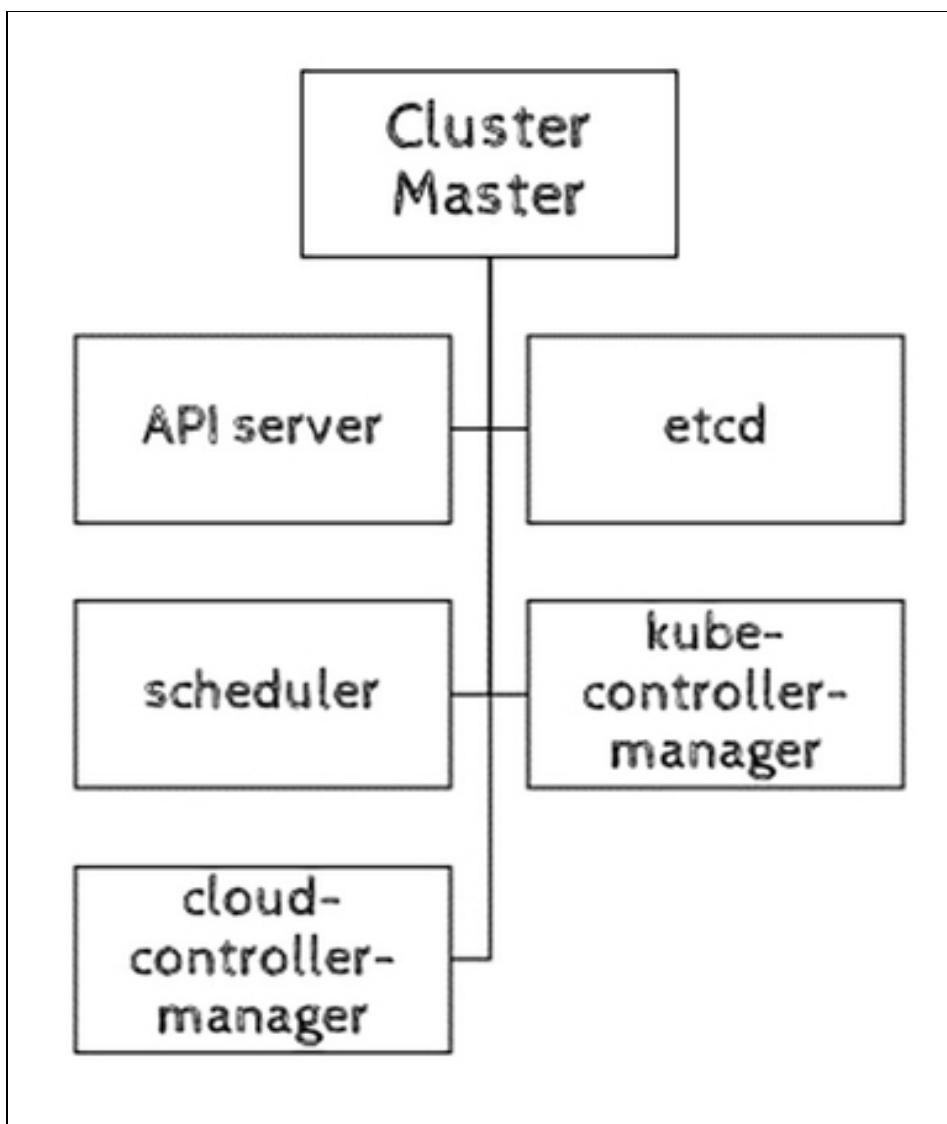


Figura 10.4 Rappresentazione del master.

Nei prossimi paragrafi definiamo più in dettaglio ogni singolo aspetto dell’architettura, così da comprenderne il funzionamento complessivo prima di iniziare a metterci le mani.

Componenti del master

Come abbiamo visto, il server principale funge da gestore di controllo primario per i cluster Kubernetes e da punto di contatto principale per amministratori e utenti. Nel complesso, i componenti sul server master lavorano insieme per accettare le richieste degli utenti, determinare i modi migliori per pianificare la gestione dei container e del loro carico di lavoro, autenticare i client e i nodi, configurare le reti a livello di cluster e gestire il ridimensionamento e le responsabilità di controllo dell’integrità.

Questi componenti possono essere installati su un singolo computer o distribuiti su più server. Ora daremo uno sguardo a ciascuno dei componenti associati ai server master.

etcd

Uno dei componenti fondamentali di Kubernetes è la *gestione della propria configurazione*, in modo da essere accessibile a livello globale. Il progetto *etcd*, sviluppato dal team di CoreOS, è un *archivio di coppie chiave-valore* molto leggero, che può essere configurato per estendersi su più nodi.

Kubernetes utilizza *etcd* per archiviare i dati di configurazione, ai quali possono accedere tutti i nodi del cluster; può essere utilizzato per individuare il servizio e può aiutare i componenti a configurarsi o riconfigurarsi in base agli aggiornamenti delle informazioni. Aiuta anche a mantenere lo stato del cluster, tramite funzionalità come l’elezione del leader; grazie a semplici API, l’interfaccia per l’impostazione o il recupero dei valori è molto semplice.

Come la maggior parte dei componenti, *etcd* può essere configurato su un singolo server master o, in scenari di produzione, distribuito su più macchine. L’unico requisito è che sia accessibile e disponibile in rete da ciascuna delle macchine Kubernetes.

kube-apiserver

Uno dei servizi principali del server API; *rappresenta il principale punto di gestione dell'intero cluster*, in quanto consente all'utente di configurare i carichi di lavoro e le unità organizzative di Kubernetes. È anche responsabile di assicurarsi che i dati di etcd e i dettagli del servizio dei container distribuiti siano sempre aggiornati. Funge da ponte tra i vari componenti per mantenere lo stato di salute dei cluster e inviare informazioni e comandi.

Il server API implementa un'interfaccia RESTful, il che significa che molti strumenti e librerie differenti possono comunicare con esso in modo molto semplice. Il client *kubectl* è disponibile come metodo predefinito per interagire con il cluster Kubernetes da un computer locale.

kube-controller-manager

Il controller manager è *un servizio generico che però ha molte responsabilità*: in primo luogo, gestisce diversi controller che regolano lo stato del cluster, gestiscono i cicli di vita del carico di lavoro ed eseguono attività di routine. Per esempio, un controller di replicazione garantisce che il numero di repliche definite per un pod corrisponda al numero attualmente distribuito sul cluster. I dettagli di queste operazioni vengono definiti tramite etcd: il responsabile del controller controlla le modifiche tramite il server API.

Quando viene rilevata una modifica, il controller legge le nuove informazioni e implementa la procedura che soddisfa lo stato desiderato. Ciò può comportare il ridimensionamento di un'applicazione, la configurazione degli endpoint presenti e così via.

kube-scheduler

È il processo che *assegna effettivamente i carichi di lavoro ai nodi nel cluster*: questo servizio legge i requisiti operativi di un carico di lavoro, analizza l'ambiente di infrastruttura corrente e posiziona il lavoro su uno o più nodi.

Lo scheduler è responsabile del monitoraggio della capacità disponibile su ciascun host, per assicurarsi che i carichi di lavoro non siano pianificati in eccesso rispetto alle risorse disponibili. Deve conoscere la

capacità totale e le risorse già allocate ai carichi di lavoro esistenti su ciascun server.

cloud-controller-manager

Kubernetes può essere distribuito in molti ambienti differenti e può interagire con vari provider per comprendere e gestire lo stato delle risorse nel cluster. Mentre Kubernetes lavora con rappresentazioni generiche di risorse, come nel caso del bilanciamento del carico, ha comunque bisogno di un modo per mappare queste risorse a quelle effettivamente fornite da provider cloud non omogenei.

I gestori di controller cloud fungono da collante per consentire a Kubernetes di interagire con i provider e le relative funzionalità, mantenendo al contempo costrutti relativamente generici al proprio interno. Ciò consente a Kubernetes di aggiornare le informazioni sullo stato in base alle informazioni raccolte dal provider, di adattare le risorse cloud quando sono necessarie modifiche nel sistema e di creare e utilizzare servizi aggiuntivi per soddisfare i requisiti di lavoro inviati al cluster.

Componenti dei nodi

In Kubernetes, i server che eseguono attività tramite l'utilizzo di container sono chiamati *nodi*; questi server prevedono alcuni requisiti, necessari per comunicare con i componenti master, per configurare la rete del container e per eseguire i carichi di lavoro effettivi loro assegnati. Un nodo è essenzialmente una macchina (fisica o virtuale) che esegue Kubernetes e sulla quale è possibile programmare dei pod. Il nodo può essere il nodo principale o un nodo di lavoro.

Container runtime

Il primo componente che ogni nodo deve avere è un *container runtime*; in genere, questo requisito è soddisfatto installando ed eseguendo Docker, ma sono disponibili anche alternative, come *rkt* e *runc*.

Il container runtime è responsabile dell'avvio e della gestione dei container, delle applicazioni incapsulate in un ambiente operativo

relativamente isolato ma leggero. Ogni unità di lavoro sul cluster è implementata come uno o più container, che devono essere distribuiti; il container runtime su ciascun nodo è il componente che esegue i container definiti nei carichi di lavoro inviati al cluster.

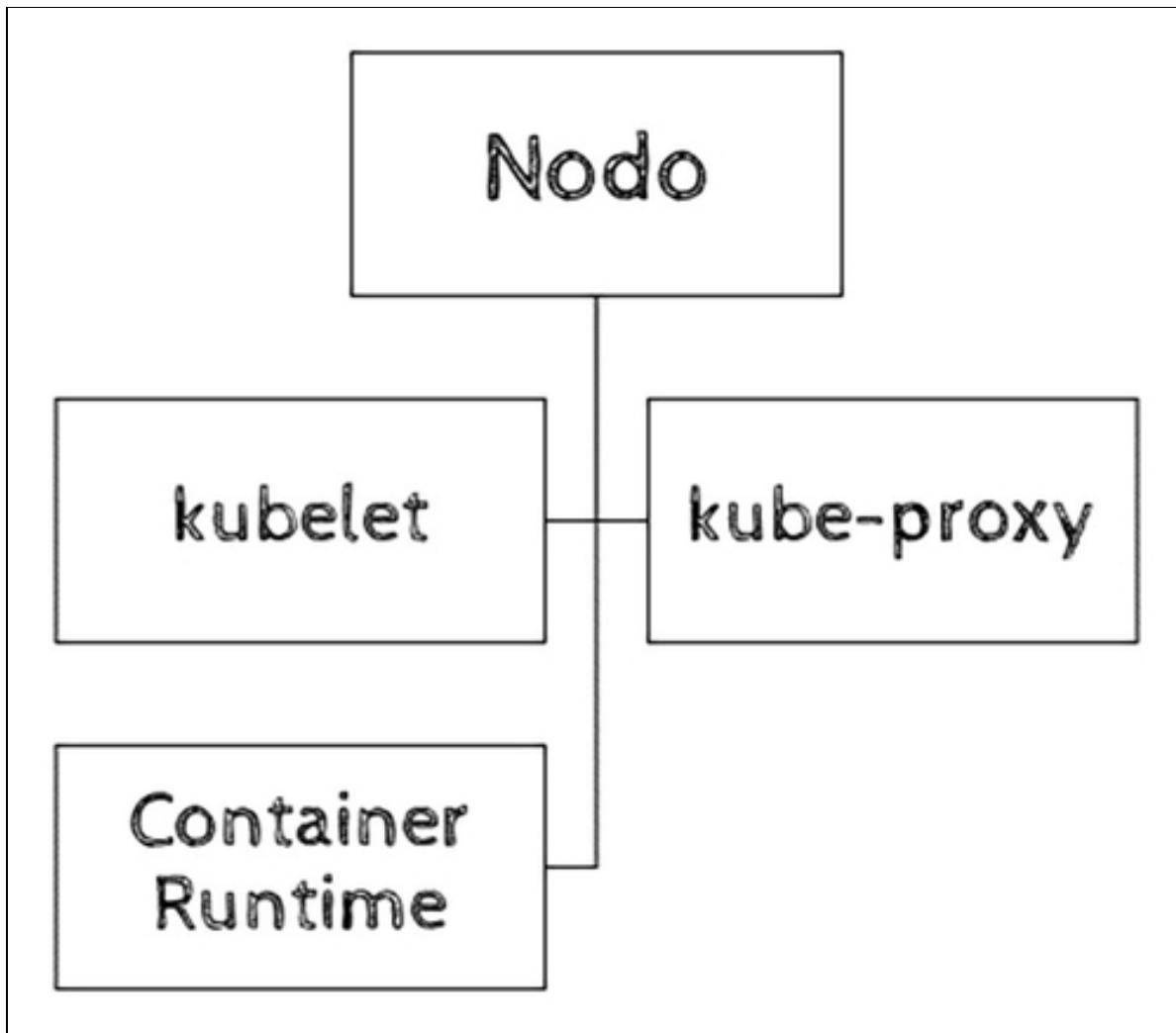


Figura 10.5 Rappresentazione di un nodo.

kubelet

Il punto di contatto principale per ciascun nodo con il gruppo cluster è un piccolo servizio chiamato *kubelet*: questo servizio è responsabile dell'inoltro delle informazioni da e verso i servizi del piano di controllo,

nonché dell’interazione con *etcd*, per leggere i dettagli di configurazione o salvare nuovi valori.

Il servizio kubelet comunica con i componenti master per autenticarsi sul cluster e ricevere istruzioni per poter operare; il lavoro viene ricevuto sotto forma di *manifest*, il quale definisce il carico di lavoro e i parametri operativi. Il processo kubelet si assume quindi la responsabilità di mantenere lo stato del lavoro sul server del nodo. Controlla il container runtime per avviare o distruggere i container secondo le necessità della soluzione.

kube-proxy

Per gestire la sottorete di singoli host e rendere disponibili i servizi ad altri componenti, su ciascun nodo server viene eseguito un piccolo servizio proxy chiamato *kube-proxy*; questo processo inoltra le richieste ai container giusti, può eseguire un bilanciamento del carico di base ed è generalmente responsabile del fatto che la rete sia correttamente configurata e accessibile, ma anche isolata, se necessario.

Mentre i container sono il meccanismo utilizzato per distribuire le applicazioni, Kubernetes utilizza ulteriori livelli di astrazione sull’interfaccia di un container per fornire funzionalità di ridimensionamento, resilienza e gestione del ciclo di vita. Invece di gestire direttamente i container, gli utenti definiscono e poi interagiscono con istanze, composte da varie primitive fornite dal modello a oggetti di Kubernetes.

Pod

Un pod è l’unità di base con cui lavora Kubernetes; i container non vengono assegnati agli host; al contrario, uno o più container, strettamente accoppiati sono incapsulati in un oggetto chiamato pod.

Un pod, in genere, rappresenta uno o più container, che devono essere controllati come fossero una singola applicazione. I pod sono costituiti da container che operano in stretta collaborazione, condividono un ciclo di vita e devono sempre essere programmati sullo stesso nodo. Sono gestiti interamente come un’unità e condividono il loro ambiente, i volumi e lo spazio IP. Nonostante la loro implementazione a container, dovremmo

generalmente considerare i pod come un'unica applicazione monolitica, per concettualizzare al meglio il modo in cui il cluster gestirà le risorse e la pianificazione del pod.

Di solito, i pod sono costituiti da un container principale che soddisfa lo scopo generale del carico di lavoro e, facoltativamente, da alcuni container di supporto che facilitano attività strettamente correlate. Si tratta di programmi che beneficiano dell'esecuzione e della gestione nei propri container, ma sono strettamente legati all'applicazione principale. Per esempio, un pod può avere un container che esegue il server primario dell'applicazione e un container di supporto che estrae file dal file system condiviso quando vengono rilevate modifiche in un repository esterno. Il ridimensionamento orizzontale è generalmente sconsigliato a livello di pod, perché esistono altri oggetti di livello superiore più adatti a quell'attività.

In generale, *gli utenti non dovrebbero gestire i pod*, poiché non forniscono alcune delle funzionalità tipicamente necessarie nelle applicazioni (come la gestione del ciclo di vita e il ridimensionamento). Al contrario, gli utenti sono incoraggiati a lavorare con oggetti di livello superiore, i quali utilizzano pod o modelli di pod come componenti di base, ma che implementano funzionalità aggiuntive.

Replicare un pod

Spesso, quando si lavora con Kubernetes, anziché lavorare con singoli pod, si gestiscono gruppi di pod identici e replicati. Questi sono creati a partire da modelli di pod e possono essere ridimensionati orizzontalmente da controller noti come controller di replicazione.

Un *controller di replicazione* è un oggetto che definisce un modello di pod e controlla i parametri per ridimensionare orizzontalmente varie repliche identiche di un pod, aumentando o diminuendo il numero di copie in esecuzione. Questo è un modo semplice per distribuire il carico e aumentare la disponibilità in modo nativo all'interno di Kubernetes; il controller di replicazione sa come creare nuovi pod secondo necessità, perché un modello che ricorda da vicino la definizione di un pod è incorporato nella configurazione del controller di replicazione.

Il controller di replicazione è dunque responsabile di garantire che il numero di pod distribuiti nel cluster corrisponda al numero di pod nella

sua configurazione; se un pod o un host presenta errori, il controller avvierà nuovi pod per garantire comunque la disponibilità del servizio. Se il numero di repliche nella configurazione di un controller cambia, il controller avvia o arresta i container in modo che corrispondano al numero desiderato. I controller possono inoltre eseguire aggiornamenti continui per passare un set di pod a una nuova versione attuando le modifiche una alla volta e riducendo al minimo l'impatto sulla disponibilità dell'applicazione.

Un set di repliche (in inglese *Replica Set*) garantisce che sul pod sia in esecuzione il numero di repliche corretto; viene spesso considerato come un sostituto del controller di replicazione, anche se la differenza chiave tra il set di repliche e il controller di replicazione è che quest'ultimo rappresenta una versione più vecchia della funzionalità, e che nel file di configurazione per la gestione delle repliche, come vedremo in seguito, vengono utilizzate specifiche differenti, come la gestione del selettore di pod basato sull'uguaglianza di una parola chiave.

Così come con i pod, sia i controller di replicazione sia i set di repliche sono raramente unità con cui si lavora direttamente.

Servizio

Un servizio è l'interfaccia esterna di uno o più pod che fornisce endpoint con cui è possibile richiamare l'applicazione o le applicazioni rappresentate dal servizio stesso. Un servizio è ospitato da un singolo indirizzo IP, ma fornisce zero o più endpoint, a seconda dell'applicazione interfacciata dal servizio. I servizi sono collegati ai pod tramite selettori di etichette. I pod hanno infatti delle etichette, il che rappresenta un modo per richiamare il pod da un client esterno.

Un client esterno utilizza questo nome e la porta con la quale è esposta una particolare applicazione, senza però conoscerne i dettagli implementativi; il servizio instrada le richieste verso l'applicazione a uno dei pod selezionati utilizzando il selettore.

In questo modo, il servizio è un'astrazione di alto livello per una raccolta di applicazioni, che lascia la scelta di utilizzo di quale pod utilizzare per eseguire una richiesta inviata al servizio stesso; spesso viene utilizzato anche per il bilanciamento del carico.

Nodi

Un *nodo* è la più piccola unità di elaborazione di Kubernetes ed è una rappresentazione di una singola macchina del cluster. Nella maggior parte dei sistemi di produzione, un nodo sarà probabilmente una macchina fisica in un data center o una macchina virtuale ospitata su un provider cloud come Google Cloud Platform.

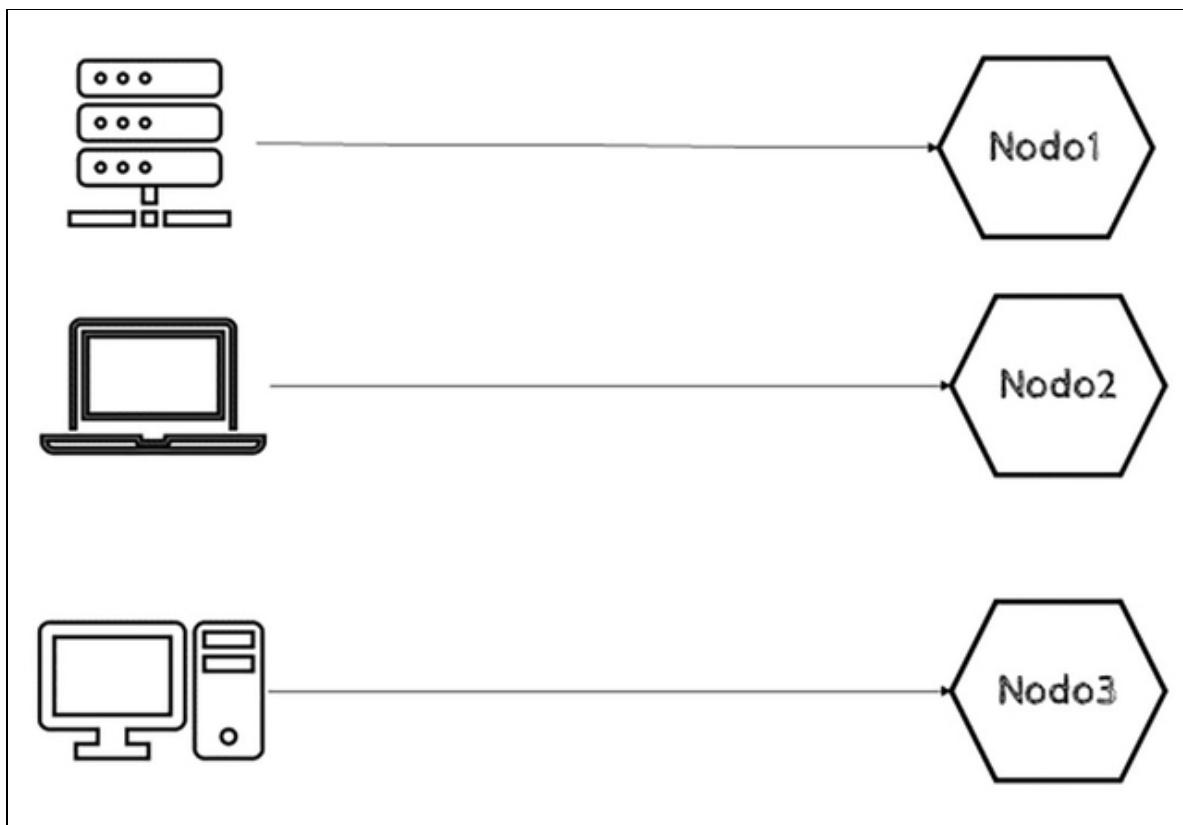


Figura 10.6 Rappresentazione di un nodo.

Pensare a una macchina come a un “nodo” ci permette di inserire uno strato di astrazione. Ora, invece di preoccuparci delle caratteristiche uniche di ogni singola macchina, possiamo invece semplicemente vedere ogni macchina come un insieme di risorse CPU e RAM, che possono essere utilizzate. In questo modo, qualsiasi macchina può sostituire qualsiasi altra macchina in un cluster Kubernetes. Possiamo vedere un nodo come una parte hardware dell’architettura.

Sebbene lavorare con singoli nodi possa essere utile, non è il modo in cui lavora Kubernetes. In generale, dovremmo pensare al cluster nel suo

insieme, invece di preoccuparci dello stato dei singoli nodi.

I nodi riuniscono le loro risorse per formare una macchina più potente. Quando si distribuiscono programmi nel cluster, la distribuzione del lavoro ai singoli nodi viene gestita in modo intelligente. Se vengono aggiunti o rimossi nodi, il cluster riorganizzerà il lavoro, se necessario.

Poiché i programmi in esecuzione sul cluster non hanno alcuna garanzia di essere eseguiti su un determinato nodo, i dati non possono essere salvati in una posizione arbitraria nel file system. Se un programma tenta di salvare dei dati in un file, ma viene trasferito su un nuovo nodo, quel file non sarà più nella stessa posizione in cui si aspetta che sia. Per questo motivo, l'archiviazione locale tradizionale associata a ciascun nodo viene trattata come una cache temporanea per contenere i programmi, ma non è possibile pretendere che i dati salvati localmente siano persistenti.

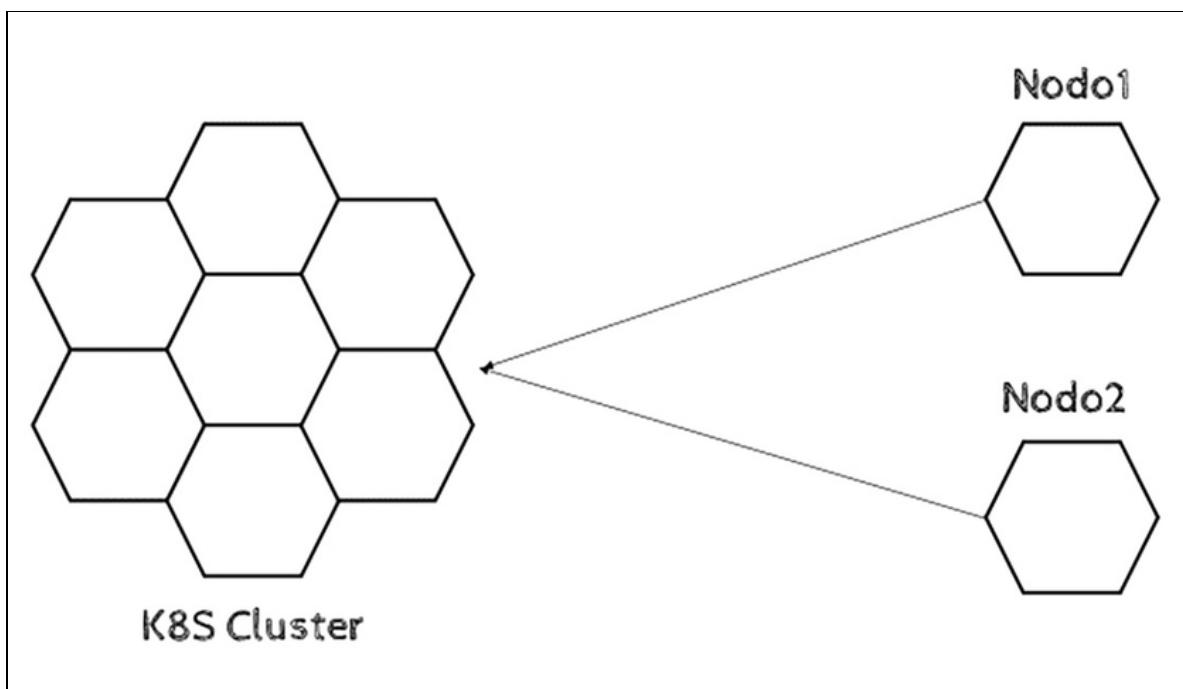


Figura 10.7 Rappresentazione di un cluster.

Per archiviare i dati in modo permanente, Kubernetes utilizza i *volumi persistenti*. Mentre le risorse di CPU e RAM di tutti i nodi sono effettivamente raggruppate e gestite dal cluster, la memorizzazione dei file persistenti non lo è. Al contrario, le unità locali o cloud possono essere collegate al cluster come volumi persistenti. Questo è paragonabile

a collegare al cluster un disco rigido esterno; i volumi persistenti forniscono infatti un file system che può essere montato sul cluster, senza essere associato ad alcun nodo in particolare.

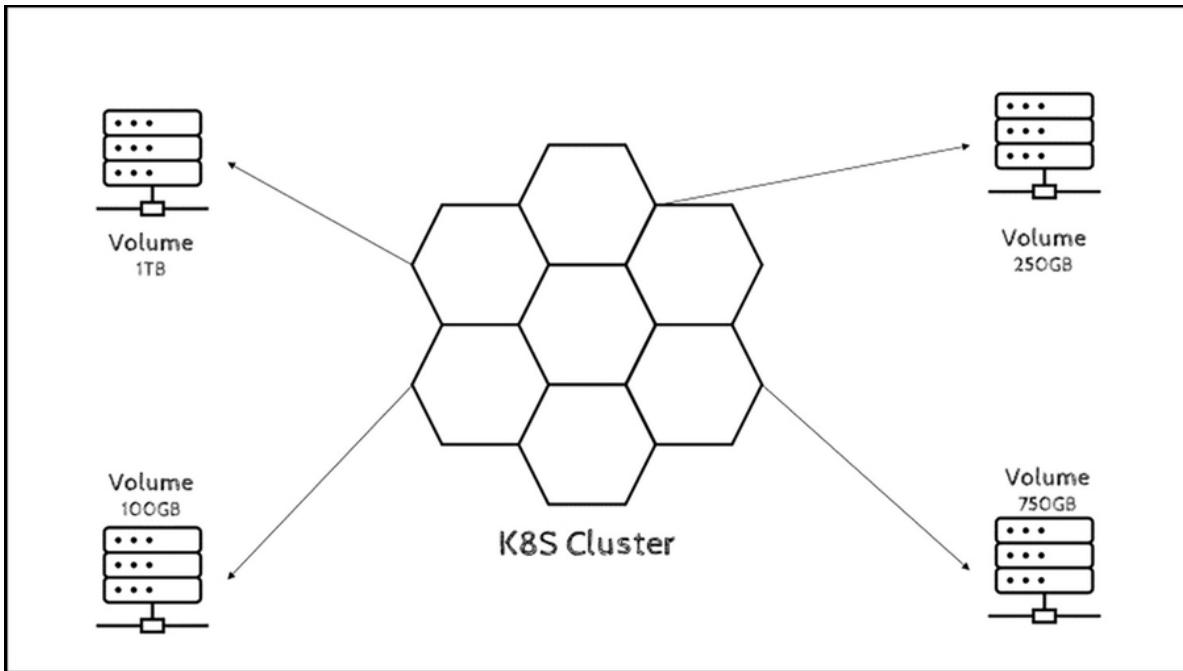


Figura 10.8 Rappresentazione dell'uso di volumi persistenti in un cluster.

Anche i volumi persistenti rientrano in quella sfera di elementi che possono essere considerati come parte dell'hardware del sistema.

I programmi in esecuzione su Kubernetes sono impacchettati come *container* Linux. I container sono uno standard ampiamente accettato, quindi esistono già molte immagini predefinite che possono essere distribuite su Kubernetes.

La “containerizzazione” consente di creare ambienti di esecuzione Linux autonomi; qualsiasi programma, e tutte le sue dipendenze, possono essere raggruppati in un singolo file e quindi condivisi. Chiunque può scaricare il container e distribuirlo sulla propria infrastruttura con pochissime attività di installazione.

Possiamo anche aggiungere più programmi in un singolo container ma, se possibile, è necessario limitarsi a un processo per container. È meglio avere molti piccoli container che un solo grande container. Se ogni container ha uno scopo ben preciso, gli aggiornamenti sono più facili da distribuire e i problemi sono più facili da diagnosticare. Come abbiamo

visto in precedenza, il container può essere considerato l’istanza di un’immagine, e quindi un elemento software del sistema K8S.

A differenza di altri sistemi che potremmo aver usato in passato, Kubernetes non esegue direttamente i container; invece avvolge uno o più container in una struttura di livello superiore chiamata *pod*. Qualsiasi container nello stesso pod condividerà le stesse risorse e la stessa rete locale. I container possono comunicare facilmente con gli altri container dello stesso pod, come se fossero sulla stessa macchina, mantenendo al contempo un grado di isolamento dagli altri.

I pod vengono utilizzati come unità di replicazione in Kubernetes. Se l’applicazione richiede molte risorse e una singola istanza del pod non riesce a sostenere il carico, Kubernetes può essere configurato per distribuire nuove repliche del pod nel cluster, se necessario. Anche quando non è sotto forte carico, è normale avere più copie di un pod in esecuzione in qualsiasi momento, per consentire il bilanciamento del carico e la resistenza ai guasti.

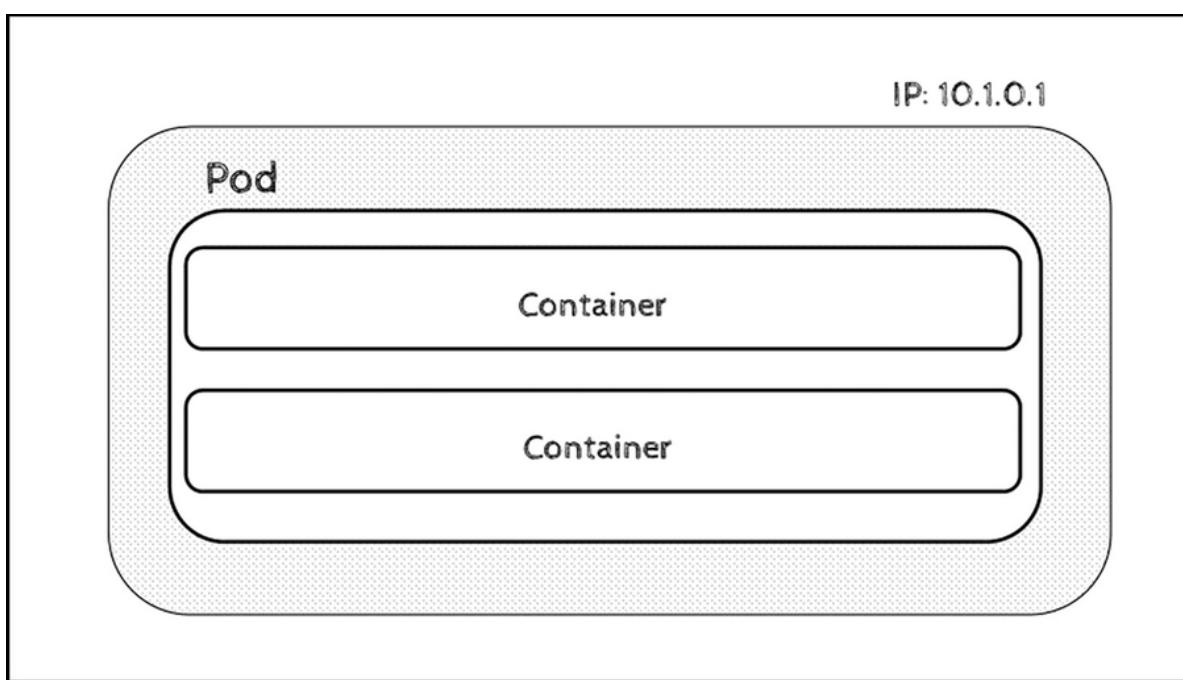


Figura 10.9 Rappresentazione di un pod.

I pod possono contenere più container, ma dovremo limitarli, quando possibile; dal momento che i pod sono ridimensionati continuamente come unità, tutti i container in un pod devono ridimensionarsi di

conseguenza, indipendentemente dalle loro esigenze individuali, e questo può portare a uno spreco di risorse. Per risolvere questo problema, i pod dovrebbero rimanere il più possibile piccoli, in genere definendo solo un processo principale e i relativi container.

Sebbene i pod siano l'unità base di calcolo in Kubernetes, in genere non vengono lanciati direttamente su un cluster. Invece, i pod sono generalmente gestiti da un ulteriore livello di astrazione: la *distribuzione* (*deployment* in inglese).

Lo scopo principale di una distribuzione è quello di dichiarare quante repliche di un pod devono essere avviate contemporaneamente. Quando al cluster viene aggiunta una distribuzione, verrà automaticamente generato il numero richiesto di pod, i quali verranno poi monitorati. Se un pod “muore”, la distribuzione lo ricrea automaticamente.

Utilizzando una distribuzione, non è necessario gestire i pod manualmente, ma possiamo semplicemente dichiarare lo stato desiderato del sistema e questo stato verrà gestito in maniera automatica.

Utilizzando i concetti sopra descritti, è possibile creare un cluster di nodi e avviare su di esso le distribuzioni di pod. C'è un ultimo problema da risolvere, tuttavia: consentire il traffico esterno all'applicazione.

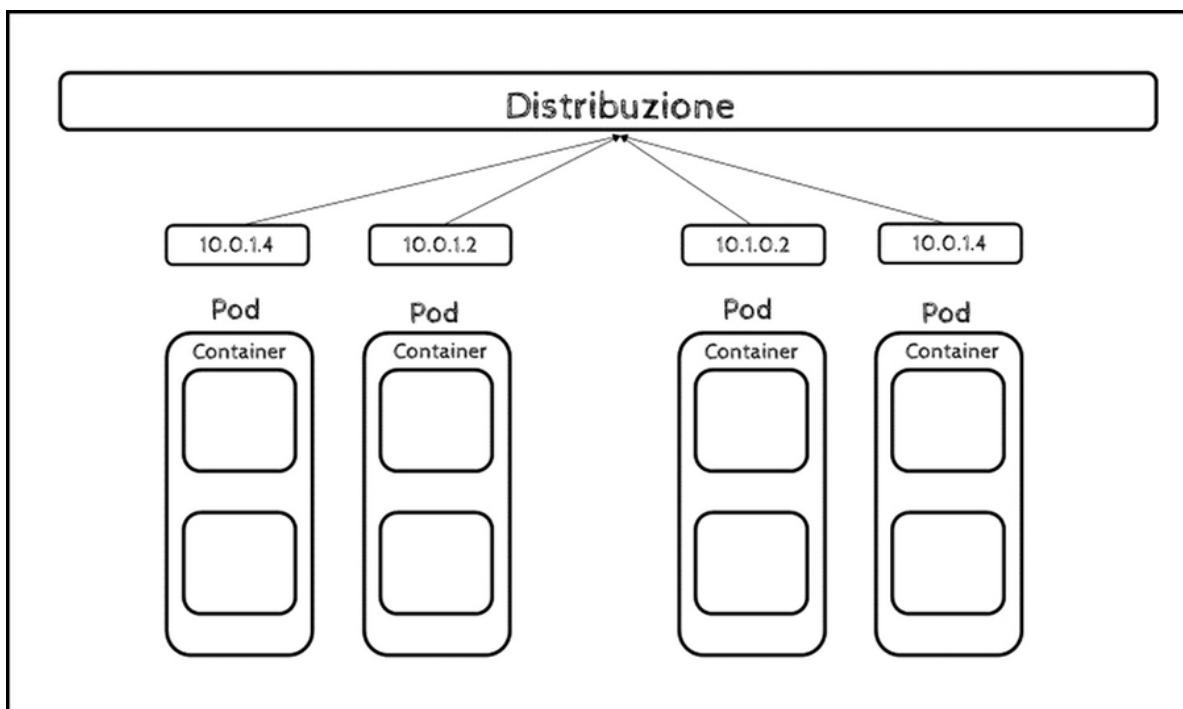


Figura 10.10 Rappresentazione di una distribuzione con quattro repliche.

Per impostazione predefinita, Kubernetes garantisce l’isolamento tra i pod e il mondo esterno. Per comunicare con un servizio in esecuzione in un pod, è necessario aprire un canale per la comunicazione. Questo è indicato come *ingresso*.

Esistono diversi modi per aggiungere un ingresso al cluster. I modi più comuni sono aggiungendo un controller *Ingress* o *LoadBalancer*. Gli esatti compromessi tra queste due opzioni non rientrano nell’ambito di questo paragrafo, ma dobbiamo essere consapevoli che l’ingresso è qualcosa che è necessario gestire prima di poter sperimentare con Kubernetes.

Installazione

Esistono diversi modi per provare Kubernetes.

- Creare delle macchine virtuali su cui installarlo.
- Installare Kubernetes direttamente sul sistema host.
- Impiegare uno strumento di distribuzione per macchine virtuali, come Vagrant.
- Utilizzare un provider cloud.

Inoltre, come specificato anche nelle guide del progetto, esistono due soluzioni che possono essere installate: quella di sviluppo - ed è quella che utilizzeremo - oppure quella per gli ambienti di produzione; in questo secondo caso, è meglio fare riferimento a uno dei partner ufficiali, che offrono provider certificati ai quali rivolgersi; quanto a questo capitolo, installeremo Kubernetes utilizzando macchine virtuali, per mostrare i diversi comandi da eseguire a seconda del sistema host.

Per Ubuntu/Debian

Come si fa normalmente a ogni installazione, innanzitutto aggiorniamo il sistema, se necessario, lanciando il comando seguente.

Listato 10.1 Verifica degli aggiornamenti.

```
root@vbox:/# apt update
```

*****ebook converter DEMO

Watermarks*****

In questi passaggi, spiegheremo come installare Kubernetes su un sistema Ubuntu e anche come distribuire Kubernetes su un cluster Ubuntu a due nodi. Il cluster a due nodi che formeremo in questo paragrafo sarà costituito da *un nodo master e un nodo slave*. Entrambi devono avranno Kubernetes installato; pertanto, vediamo come installare Kubernetes su entrambi i nodi.

Per aggiungere la chiave di firma Kubernetes su entrambi i nodi, lanciamo il comando seguente.

Listato 10.2 Aggiunta della chiave.

```
root@vbox:/# curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add
```

Se curl non è installato sul sistema, possiamo installarlo come segue.

Listato 10.3 Installazione di curl.

```
root@vbox:/# apt install curl
```

Ora aggiungiamo il repository di Kubernetes su entrambi i nodi e poi lanciamo il seguente comando su entrambi i nodi.

Listato 10.4 Aggiunta del repository

```
root@vbox:/# sudo apt-add-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"
```

Il passaggio finale del processo prevede l'installazione dei pacchetti necessari al funzionamento di Kubernetes su entrambi i nodi, tramite il seguente comando.

Listato 10.5 Installazione kubeadm

```
root@vbox:/# apt install -y kubeadm kubelet kubectl
```

Installeremo questi pacchetti su tutti i sistemi; ognuno di essi ha un compito specifico.

- `kubeadm`: il comando per avviare il cluster.
- `kubelet`: il componente che gira su tutte le macchine del cluster e si occupa di attività come l'avvio di *pod* e *container*.
- `kubectl`: lo strumento a riga di comando per comunicare con il cluster.

`kubeadm` non installa o gestisce `kubelet` o `kubectl`, quindi dovremo assicurarci che corrispondano alla versione di Kubernetes che vogliamo

installare. In caso contrario, potrebbe sorgere un problema di disallineamento della versione che può portare a comportamenti imprevedibile e a errori.

Possiamo controllare il numero di versione dei tre componenti e verificare l'installazione tramite i seguenti comandi.

Listato 10.6 Verifica di installazione.

```
root@vbox:/# kubeadm version  
root@vbox:/# kubelet version  
  
root@vbox:/# kubectl version
```

È necessario *disabilitare la memoria swap* su entrambi i nodi, poiché Kubernetes non funziona correttamente su un sistema che utilizza lo swap; per farlo, lanciamo il comando seguente su entrambi i nodi.

Listato 10.7 Verifica di installazione.

```
root@vbox:/# swapoff -a
```

Per RHEL/Centos/Fedora

Per prima cosa, è necessario aggiungere il repository, creando un file tramite il seguente comando, con cui apriremo un editor di testi da terminale, in cui copieremo il testo sotto riportato, con le informazioni sul repository Kubernetes.

Listato 10.8 Aggiunta del repository.

```
root@vbox:/# nano /etc/yum.repos.d/kubernetes.repo  
[kubernetes]  
name=Kubernetes  
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64  
enabled=1  
gpgcheck=1  
repo_gpgcheck=1  
  
gpgkey=https://packages.cloud.google.com/yum/doc/yum-  
key.gpg https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
```

Salviamo e chiudiamo il file. Ora lanciamo i seguenti comandi, per verificare se sono presenti aggiornamenti e poi procediamo con l'installazione dei pacchetti necessari.

Listato 10.9 Installazione di Kubernetes.

```
root@vbox:/# dnf install -y kubelet kubeadm kubectl --disableexcludes=kubernetes
```

Quando l'installazione è terminata, è necessario abilitare il servizio `kubelet` affinché sia disponibile e possa essere avviato.

Listato 10.10 Abilitazione servizio `kubelet`.

```
root@vbox:/# systemctl enable --now kubelet
```

Ora che il servizio è abilitato, dobbiamo impostare SELinux in modalità permissiva, eseguendo i comandi riportati; questi lo disabilitano efficacemente e ciò è necessario per consentire ai container di accedere al file system host, come avviene per esempio per le reti di pod.

Puoi comunque lasciare SELinux abilitato, se hai le competenze necessarie per configurarlo; tuttavia, potrebbe richiedere impostazioni che non sono supportate da `kubeadm`.

Listato 10.11 Impostare SELinux in modalità permissiva.

```
root@vbox:/# setenforce 0  
root@vbox:/# sed -i 's/^SELINUX=enforcing$/SELINUX=permissive/' /etc/selinux/config
```

L'ultimo passaggio richiede di disabilitare la memoria swap, perché Kubernetes non funziona correttamente su un sistema con swap.

Listato 10.12 Disabilitare la memoria swap.

```
root@vbox:/# swapoff -a
```

Ciò disabilita temporaneamente la swap. Per disabilitarla permanentemente (non verrà riattivata al riavvio), apriamo il file `fstab` e modifichiamolo trasformando in commento la riga che inizia come segue.

Listato 10.13 Una riga del file `/etc/fstab`.

```
/dev/mapper/cl-swap
```

Kubernetes utilizza varie porte per la comunicazione e per l'accesso, e queste porte devono essere accessibili e non limitate dal firewall. Impostiamo quindi le seguenti regole tramite `firewall-cmd`, per abilitarle e renderle disponibili.

Listato 10.14 Abilitazione delle porte.

```
root@vbox:/# firewall-cmd --permanent --add-port=6443/tcp  
root@vbox:/# firewall-cmd --permanent --add-port=2379-2380/tcp  
root@vbox:/# firewall-cmd --permanent --add-port=10250/tcp  
root@vbox:/# firewall-cmd --permanent --add-port=10251/tcp  
root@vbox:/# firewall-cmd --permanent --add-port=10252/tcp  
root@vbox:/# firewall-cmd --permanent --add-port=10255/tcp  
root@vbox:/# firewall-cmd -reload
```

```
root@vbox:/# modprobe br_netfilter
root@vbox:/# echo '1' > /proc/sys/net/bridge/bridge-nf-call-iptables
```

Per Windows

Per aggiungere `kubectl` su Windows, dobbiamo prima di tutto scaricare il pacchetto dal sito ufficiale oppure usando il comando `curl` come di seguito.

Listato 10.15 Download di kubectl in Windows.

```
root@vbox:/#
curl -LO https://storage.googleapis.com/kubernetes-
release/release/v1.18.0/bin/windows/amd64/kubectl.exe
```

Dobbiamo poi inserire all'interno della variabile di ambiente `PATH` il percorso in cui è stato installato, affinché possa essere richiamato tramite riga di comando; per verificare che l'installazione sia andata a buon fine, possiamo aprire un terminale e digitare il comando seguente.

Listato 10.16 Verifica dell'installazione.

```
$ kubectl version
```

Per macOS

Per aggiungere `kubectl` su macOS, dobbiamo prima di tutto scaricare il pacchetto con il comando `curl`.

Listato 10.17 Download di kubectl.

```
root@macos:/# curl -LO https://storage.googleapis.com/kubernetes-
release/release/$(curl -s https://storage.googleapis.com/kubernetes-
release/release/stable.txt)/bin/darwin/amd64/kubectl
```

Rendiamo eseguibile il file aggiungendo i permessi di esecuzione e poi spostiamo il file binario nella cartella degli eseguibili del sistema.

Listato 10.18 Cambio dei permessi e aggiunta agli eseguibili.

```
root@macos:/# chmod +x ./kubectl
root@macos:/# sudo mv ./kubectl /usr/local/bin/kubectl
```

Per verificare che l'installazione sia andata a buon fine, possiamo aprire un terminale e digitare il seguente comando.

Listato 10.19 Verifica dell'installazione.

```
root@macos:/# kubectl version -client
```

Avendo a disposizione il gestore dei pacchetti Homebrew, è anche possibile lanciare il seguente comando per installare Kubernetes.

Listato 10.20 Installazione tramite brew

```
root@macos:/# brew install kubernetes-cli
```

Creare un cluster

Prima di cominciare a lavorare con i pod, è necessario configurare un cluster Kubernetes: riprendendo quanto fatto finora, per il momento utilizzeremo i nodi trattati nelle installazioni precedenti e definiremo *un nodo come master e l'altro come slave*.

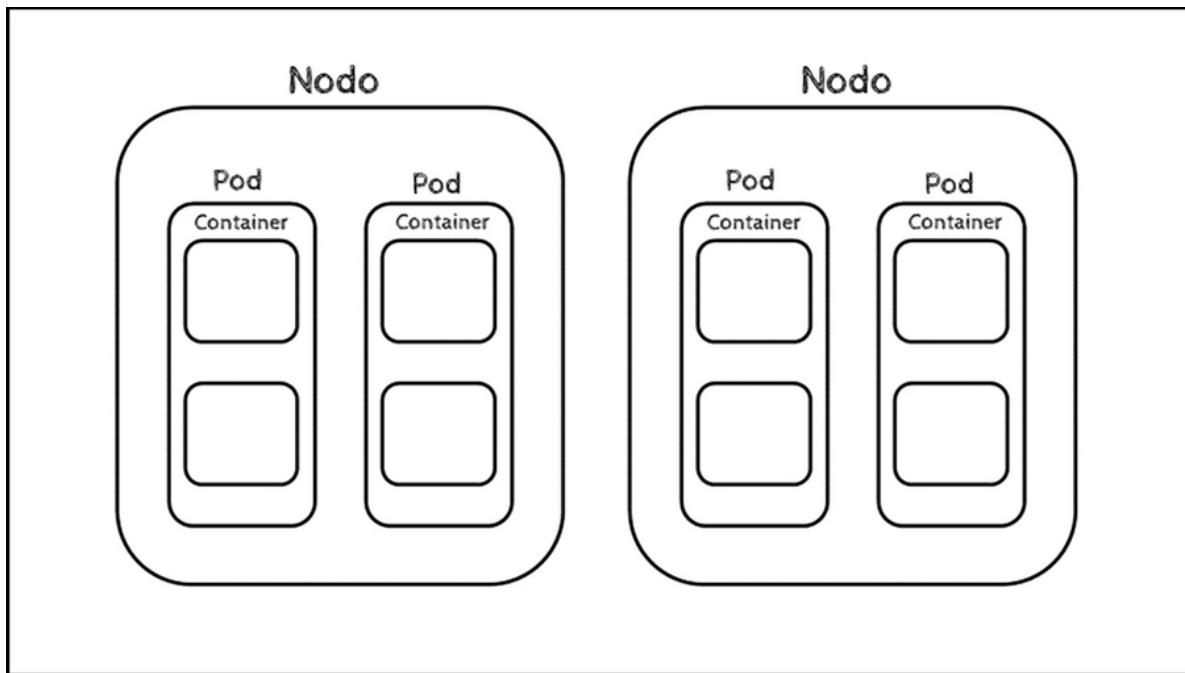


Figura 10.11 Esempio di Cluster con due nodi.

Ognuno di essi deve avere degli *hostname* unici, per cui impostiamoli tramite terminale, lanciando i seguenti comandi.

Listato 10.21 Configurazione dell'hostname sul nodo master.

```
root@master:/# hostnamectl set-hostname master-node
```

Listato 10.22 Configurazione dell'hostname sul nodo slave.

```
root@slave:/# hostnamectl set-hostname slave-node
```

A questo punto, possiamo inizializzare Kubernetes sul nodo master; lanciando il seguente comando, verrà avviato in maniera automatica il processo di configurazione del nodo.

Listato 10.23 Inizializzazione nodo master.

```
root@slave:/# kubeadm init
```

Una volta inizializzato correttamente Kubernetes, è necessario abilitare l'utente a iniziare a utilizzare il cluster. Nel nostro scenario, utilizzeremo l'utente *root*; è molto importante prestare attenzione all'output del comando precedente: questo fornirà le informazioni sullo stato di configurazione e anche i comandi da eseguire, non appena la configurazione è terminata, per poter avviare il cluster. Sarà infatti necessario creare una cartella per il cluster e poi copiare al suo interno i file necessari all'avvio. Nell'output, ci verranno forniti comandi simili ai seguenti.

Listato 10.24 Comandi per l'avvio del cluster.

```
mkdir -p $HOME/.kube  
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config  
  
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Dopo averli eseguiti, il cluster è pronto per essere avviato: possiamo dunque aggiungere delle macchine ai nodi. Sempre dall'output del comando precedente traiamo il comando per farlo, che sarà simile al seguente.

Listato 10.25 Aggiunta di una macchina al nodo.

```
kubeadm join 192.168.0.47:6443 --token XXX \ --discovery-token-ca-cert-hash  
ha256:XYZ
```

Questo comando verrà eseguito più in avanti per lavorare con i nodi di lavoro; per il momento, verifichiamo che `kubectl` sia avviato e attivo correttamente.

Listato 10.26 Elenco dei nodi.

```
root@master:/# kubectl get nodes
```

In questo momento, potremmo vedere che lo stato del nodo principale è `NotReady`: questo perché *non abbiamo ancora distribuito la rete pod sul*

cluster.

La rete pod è la rete di overlay per il cluster, che viene distribuita sopra all'attuale rete di nodi ed è progettata per consentire la connettività attraverso il pod.

La distribuzione del cluster di rete è un processo estremamente flessibile, a seconda delle esigenze e sono infatti disponibili molte opzioni. Poiché vogliamo mantenere la nostra installazione la più semplice possibile, utilizzeremo il plugin *Weavenet*, che non richiede alcuna configurazione o codice aggiuntivo e fornisce un indirizzo IP per ogni pod.

Eseguiamo quindi questi comandi per ottenere la configurazione della rete pod.

Listato 10.27 Configurazione della rete pod.

```
root@master:/# export kubever=$(kubectl version | base64 | tr -d '\n')  
root@master:/# kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-  
version=$kubever"
```

Una volta terminata l'esecuzione, proviamo nuovamente a digitare il comando `kubectl get nodes` (Listato 10.26) e verifichiamo che lo stato sia Ready.

Adesso, per aggiungere alla rete il nodo slave, con lo scopo di formare un cluster, eseguiamo il comando che ci era stato restituito come output del comando `kubeadm init`.

Listato 10.28 Configurazione della rete pod.

```
root@master:/# kubeadm join 192.168.0.47:6443 --token XXX \           --discovery-  
token-ca-cert-hash ha256:XYZ
```

Ora, lanciando il comando seguente sul nodo principale, verrà confermato che i due nodi, ossia il nodo master e il nodo di lavoro, sono in esecuzione sul sistema. L'esecuzione di `kubectl get nodes`, infatti, mostra che il cluster a due nodi ora è attivo e in esecuzione attraverso il sistema di gestione dei container Kubernetes.

Avviare un pod

Un pod è una raccolta di container che condividono una rete e sfruttano un unico *namespace*; è anche considerato l'unità base di distribuzione in

Kubernetes. Tutti i container di pod sono programmati e gestiti dallo stesso nodo.

La seguente riga di comando di kubectl crea una distribuzione denominata `my-nginx`, per garantire che il pod contenente l'omonimo server web sia sempre in esecuzione.

Listato 10.29 Avvio di un pod.

```
root@master:/# kubectl create deployment --image nginx my-nginx
```

È possibile creare una distribuzione utilizzando un file in formato `.yaml`, così come abbiamo visto con Docker Compose; quando creiamo un oggetto in Kubernetes, dobbiamo fornire le specifiche dell'oggetto che descrivono lo stato desiderato, nonché alcune informazioni di base sull'oggetto (come un nome). Quando si utilizzano le API Kubernetes per creare l'oggetto, la richiesta deve includere tali informazioni come JSON; molto spesso, queste informazioni vengono specificate più agilmente in un file `.yaml`, che `kubectl` converte e utilizza per definire l'oggetto.

Ecco un esempio di file `.yaml` che mostra i campi richiesti e le specifiche dell'oggetto per una distribuzione Kubernetes.

Listato 10.30 Esempio di file `.yaml` per la distribuzione.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rss-deployment
  labels:
    app: web
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: front-end
          image: nginx
          ports:
            - containerPort: 80
        - name: rss-reader
          image: nickchase/rss-php-nginx:v1
          ports:
            - containerPort: 88
```

In questo file stiamo definendo il tipo di oggetto, tramite l'attributo kind; successivamente specifichiamo i metadati. Qui stiamo specificando il nome della distribuzione, nonché l'etichetta che useremo per identificarla tramite Kubernetes.

Infine, specificheremo gli oggetti reali che compongono la distribuzione: la proprietà spec include tutti i container, i volumi di archiviazione o altri elementi che Kubernetes deve conoscere, nonché le proprietà (per esempio se riavviare il container in caso di errore). Le specifiche comprendono varie informazioni, come il container che verrà utilizzato, il numero di repliche, ossia il numero di pod che verranno avviati, il limite delle risorse che deve utilizzare, nonché le porte da esporre.

Andiamo avanti e creiamo la distribuzione. Aggiungiamo il contenuto del file .yaml a un file chiamato deployment.yaml e usiamo kubectl per crearlo.

Listato 10.31 Comando kubectl create.

```
root@master:/# kubectl create -f deployment.yaml
deployment "rss-deployment" created
```

Per vedere se l'esecuzione è stata avviata correttamente, possiamo controllare l'elenco delle distribuzioni presenti e attive, tramite il comando kubectl get deployments.

Listato 10.32 Comando kubectl get deployments.

```
root@master:/# kubectl get deployments
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
rss-deployment   2         2         2           1          7s
```

Come possiamo vedere, Kubernetes ha avviato entrambe le repliche, ma ne troviamo disponibile solo una. È possibile controllare il registro degli eventi tramite il comando kubectl describe, che riporta tutte le informazioni sulla distribuzione appena creata.

Listato 10.33 Comando kubectl describe.

```
root@master:/# kubectl describe deployment rss-deployment
Name:                   rss-site
Namespace:              default
CreationTimestamp:      Mon, 09 Jan 2017 17:42:14 +0000=
Labels:                 app=web
Selector:               app=web
Replicas:               2 updated | 2 total | 1 available | 1 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
```

```

OldReplicaSets:          <none>
NewReplicaSet:           rss-site-4056856218 (2/2 replicas created)
Events:
FirstSeen   LastSeen   Count  From             SubobjectPath  Type    Reason
                  Message
-----  -----  -----  ----  -----  -----  -----
-
46s        46s       1      {deployment-
controller}           Normal  ScalingReplicaSet  Scaled up replica set rss-
site-4056856218 to 2

```

Come possiamo vedere, non c'è nessun problema, ma semplicemente Kubernetes non ha ancora finito di estendere di scala l'applicazione. Ancora qualche secondo ed, eseguendo nuovamente il comando, potremo vedere che entrambi i pod sono in esecuzione.

Allo stesso modo, possiamo utilizzare i file `.yaml` per definire un altro tipo di oggetto, come un servizio o un pod; l'esempio seguente crea un pod a partire da un'immagine `nginx`, e possiamo vedere che gli attributi utilizzati sono molto simili a quelli visti pocanzi.

Listato 10.34 Esempio di file `.yaml` per la definizione di un pod.

```

apiVersion: v1
kind: Pod
metadata:
  name: rss-pod
  labels:
    app: web
spec:
  containers:
    - name: frontend
      image: nginx
      ports:
        - containerPort: 80
    - name: rss-reader
      image: nickchase/rss-php-nginx:v1
      ports:
        - containerPort: 88

```

Avendo a disposizione un file di questo tipo, possiamo utilizzare `kubectl` per creare l'oggetto in questione a partire da questo file, utilizzando il comando `create`.

Listato 10.35 Comando `kubectl create`.

```

root@master:/# kubectl create -f pod.yaml
pod "rss-pod" created

```

L'output del comando ci comunica che il pod il cui nome è stato specificato nei metadati del file `pod.yaml` è stato correttamente creato; per

verificare che l'avvio sia andato a buon fine, possiamo utilizzare il comando `kubectl get pods`, il quale ci restituirà un elenco di pod attivi.

Listato 10.36 Comando `kubectl get pods`.

```
root@master:/# kubectl get pods
NAME        READY     STATUS    RESTARTS   AGE
rss-pod     1/1      Running   0          9m
```

A volte, ovviamente, le cose non vanno come previsto. Potremmo riscontrare un problema di rete oppure potremmo aver sbagliato a digitare qualcosa nel file `.yaml`. Lanciando il comando precedente, infatti, potremmo riscontrare un output come quello riportato nel Listato 10.37.

Listato 10.37 Comando `kubectl get pods` con errore.

```
root@master:/# kubectl get pods
NAME        READY     STATUS    RESTARTS   AGE
rss-pod     0/1      ErrImagePull   0          9m
```

In questo caso, tramite l'output restituito è possibile vedere che il container non è stato avviato perché si è verificato un problema con l'immagine; per cercare di risolvere il problema, possiamo usare l'istruzione `kubectl describe` e richiedere ulteriori informazioni sul pod.

Listato 10.38 Comando `kubectl describe`.

```
root@master:/# kubectl describe pod rss-pod
Name:           rss-pod
Namespace:      default
Node:          10.0.10.7/10.0.10.7
Start Time:    Sun, 08 May 2020 08:43:47 +0000
Labels:         app=web
Status:        Pending
IP:             10.200.18.2
Controllers:   <none>
Containers:
  front-end:
    Container
    ID:      7b0310759
    Image:      nginx
    Image
    ID:      2678d257735fad84a15f
    Port:      80/TCP
    State:      Running
    Started:   Sun, 08 May 20 08:43:49 +0000
    Ready:      True
    Restart Count: 0
    Environment Variables: <none>
  rss-reader:
    Container ID: nickchase/rss-php-nginx
    Image:
```

```

Image ID: [REDACTED]
Port: 88/TCP
State: Waiting
  Reason: ErrImagePull
Ready: False
Restart Count: 0
Environment Variables: <none>
Conditions:
  Type Status
  Initialized True
  Ready False
  PodScheduled True
No volumes.
QoS Tier: BestEffort
Events:
  FirstSeen LastSeen Count From SubobjectPath Type
  e Reason Message
  ----- -----
  45s 45s 1 {default-} Normal Scheduled Successfully
scheduler} assigned rss-site to 10.0.10.7
  44s 44s 1 {kubelet 10.0.10.7} spec.containers{front-
end} Normal Pulling pulling image "nginx"
  45s 43s 2 {kubelet 10.0.10.7} Warning MissingClusterDNS kubelet does
not have ClusterDNS IP configured and cannot create Pod using "ClusterFirst"
policy. Falling back to DNSDefault policy.
  43s 43s 1 {kubelet 10.0.10.7} spec.containers{front-
end} Normal Pulled Successfully pulled image "nginx"
  43s 43s 1 {kubelet 10.0.10.7} spec.containers{front-
end} Normal Created Created container with docker id a42edaa6dfbf
  43s 43s 1 {kubelet 10.0.10.7} spec.containers{front-
end} Normal Started Started container with docker id a42edaa6dfbf
  43s 29s 2 {kubelet 10.0.10.7} spec.containers{rss-reader}
Normal Pulling pulling image "nickchase/rss-php-nginx"
  42s 26s 2 {kubelet 10.0.10.7} spec.containers{rss-reader}
Warning Failed Failed to pull image "nickchase/rss-php-nginx": Tag
latest not found in repository docker.io/nickchase/rss-php-nginx
  42s 26s 2 {kubelet 10.0.10.7} Warning FailedSync Error syncing
pod, skipping: failed to "StartContainer" for "rss-reader" with ErrImagePull: "Tag
latest not found in repository docker.io/nickchase/rss-php-nginx"
  41s 12s 2 {kubelet 10.0.10.7} spec.containers{rss-reader}
Normal BackOff Back-off pulling image "nickchase/rss-php-nginx"
  41s 12s 2 {kubelet 10.0.10.7}

```

Come possiamo vedere, questo comando restituisce molte informazioni; al fine di risolvere il problema, siamo più interessati agli eventi, in particolare quando vengono visualizzati *warning* o errori. In una delle ultime righe troviamo una descrizione piuttosto estesa dell'errore: ci dice che *non ha trovato l'ultima immagine nel repository*, e questo perché non abbiamo specificato la versione da scaricare nel file YAML.

Per risolvere il problema, dobbiamo innanzitutto eliminare il *pod*; questo può essere fatto tramite il comando `kubectl delete`.

Listato 10.39 Comando kubectl delete.

```
root@master:/# kubectl delete pod rss-pod
pod "rss-pod" deleted
```

Dopo aver sistemato il file YAML, è possibile avviare nuovamente il pod.

L'uso di YAML per le definizioni di K8 offre numerosi vantaggi, tra cui i seguenti.

- *Comodità*: non dovremo più aggiungere tutti i parametri alla riga di comando, creando un'istruzione lunghissima e soggetta a errori.
- *Manutenibilità*: i file YAML possono essere facilmente gestiti tramite controllo del codice sorgente, in modo da poter tenere traccia delle modifiche.
- *Flessibilità*: saremo in grado di creare strutture molto più complesse utilizzando un file YAML rispetto a quanto si possa fare tramite riga di comando.

YAML, che sta per *Yet Another Markup Language* o anche *YAML Ain't Markup Language* è un formato di testo leggibile per specificare le informazioni sul tipo di configurazione.

Modalità imperativa vs dichiarativa

Esistono due modi di base per distribuire applicazioni tramite Kubernetes: *imperativamente*, utilizzando i numerosi comandi di `kubectl`, o *dichiarativamente*, utilizzando file che ne rappresentano la struttura e che poi vengono utilizzati tramite il comando `kubectl apply`. Il primo modo è utile per l'apprendimento e la sperimentazione interattiva; il secondo per le distribuzioni riproducibili, come per esempio per gli ambienti di produzione, sebbene sia comunque possibile utilizzare alcuni comandi `kubectl` anche per il debug in produzione.

Modalità imperativa

Il modo più breve per distribuire su Kubernetes consiste nell'usare il comando `kubectl run`. Sostituendo `myapp` e `myrepo:mytag` con il nome dell'applicazione e il nome e il tag dell'immagine da utilizzare, possiamo avviare un cluster che contenga dei container, come richiesto. Il comando può sembrare familiare, avendo già usato Docker per avviare un container in locale, ma la somiglianza si ferma qui: `kubectl`, infatti, traduce il comando imperativo in un oggetto dichiarativo di distribuzione Kubernetes, che utilizza poi per compiere tutti i passaggi necessari per completare l'operazione richiesta.

Per esempio, questo vale per i comandi `kubectl get`, `kubectl describe`, `kubectl logs` e così via; sono tutti comandi molto utili, ma che restituiscono informazioni di sola lettura e che ci aiutano a chiedere a K8S di fare quello che desideriamo.

Modalità dichiarativa

Il più delle volte, useremo semplicemente `kubectl apply` e i file di configurazione in formato YAML (o JSON), che verranno salvati da Kubernetes nella cartella `etc`. Questa modalità ci aiuta a definire di che cosa abbiamo bisogno, così che K8S possa poi lavorare in maniera autonoma, utilizzando le indicazioni che abbiamo fornito; gli oggetti Kubernetes, in questo modo, possono essere facilmente creati, aggiornati ed eliminati, memorizzando più file di configurazione di oggetti in una directory e usando `kubectl` applicare per creare ricorsivamente e aggiornare tali oggetti secondo necessità. Questo metodo mantiene le modifiche apportate agli oggetti, senza dover gestire in modo diretto gli oggetti stessi; tramite il comando `kubectl diff` è anche possibile avere un'anteprima delle modifiche che verranno applicate.

YAML è un superset di JSON, il che significa che qualsiasi file JSON valido è anche un file YAML valido. Quindi, da un lato, conoscendo JSON, non ci sono molte altre nozioni da apprendere; in più, ci sono solo due tipi di strutture che è necessario conoscere per utilizzare un file YAML, ossia le *liste* e le *mappe* (i dizionari). Questi due oggetti possono essere annidati molteplici volte, ma permettono di descrivere qualunque tipo di oggetto in modo molto semplice.

Così come abbiamo visto per i container, è possibile utilizzare un comando che ci permetta di avviare una sessione SSH con il terminale

del container stesso e che ci permetta di lavorare al suo interno; `kubectl` dispone infatti del sottocomando `exec`, che possiamo utilizzare per esempio per avviare una finestra *bash*.

Listato 10.40 Comando `kubectl exec`.

```
root@master:/# kubectl exec -it rss-pod -- /bin/bash
```

Gestire un servizio

Esiste un ulteriore livello di astrazione che riguarda i pod e viene rappresentato dal concetto di *servizio*: si tratta di una tecnica che permette di esporre un'applicazione, in esecuzione su un insieme di pod, come se fosse un servizio in rete. Un servizio consente infatti l'accesso in rete a una serie di pod presenti in un cluster Kubernetes.

I servizi selezionano i pod in base alle loro etichette o selettori; quando viene effettuata una richiesta di rete a un servizio, tale servizio seleziona nel cluster tutti i pod corrispondenti al selettore del servizio, ne sceglie uno e inoltra la richiesta di rete.

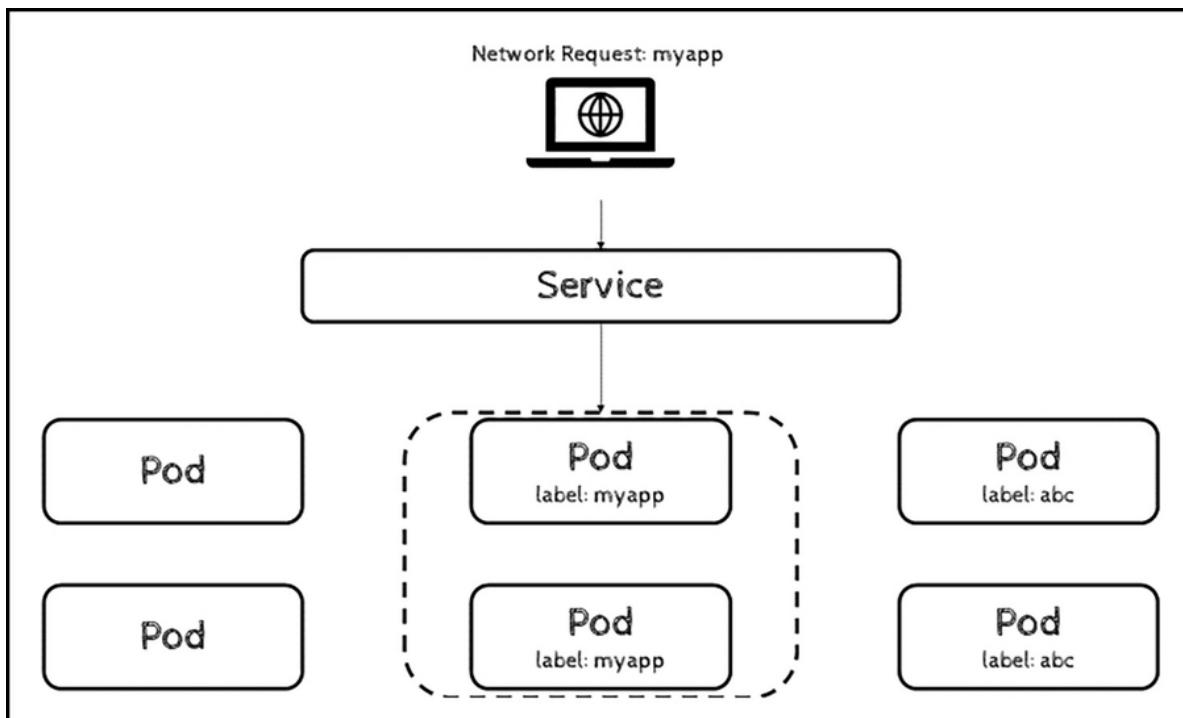


Figura 10.12 Rappresentazione del funzionamento di un servizio.

Potrebbe dunque sorgere spontanea la domanda: che cosa c'è di diverso rispetto a una distribuzione?

Una distribuzione è responsabile della *gestione di una serie di pod in esecuzione*, mentre un servizio è responsabile dell'*abilitazione dell'accesso alla rete a una serie di pod*.

Potremmo utilizzare una distribuzione senza un servizio per mantenere un insieme di pod identici in esecuzione nel cluster Kubernetes. La distribuzione potrebbe essere ridimensionata e ridotta e i pod potrebbero essere replicati, così come è possibile accedere a ciascun pod singolarmente tramite richieste di rete dirette (anziché astrarre il tutto dietro un servizio). Viceversa, potremmo anche utilizzare un servizio senza disporre di una distribuzione; in questo caso, dovremmo creare ogni pod singolarmente e comunque il nostro servizio potrebbe indirizzare le richieste di rete a quei pod, selezionandoli in base alle loro etichette.

Vediamo un esempio di servizio che viene utilizzato per gestire le richieste ricevute da una rete esterna. In questo caso, usiamo un servizio di tipo *NodePort*, così che quel servizio possa essere utilizzato tramite una porta specificata; nel seguente file definiamo una porta statica, per cui, quando viene ricevuta una richiesta, il servizio si attiva per selezionare tutti quei pod che abbiano, come selettore, app con il valore myapp.

Listato 10.41 Il file service.yaml.

```
apiVersion: v1
kind: Service
metadata:
  name: hostname-service
spec:
  type: NodePort
  selector:
    app: myapp
  ports:
    - nodePort: 30063
      port: 8080
      targetPort: 80
```

Nel campo `ports` abbiamo utilizzato l'attributo `nodePort`, perché intendiamo assegnare una porta statica a ogni nodo; gli attributi `port` e `targetPort` espongono, rispettivamente, la porta interna al cluster e la porta del container al quale inviare le richieste. Ciò vuol dire che tutto il

traffico che viene inviato alla porta 30063 viene diretto al servizio, mentre i diversi container rimangono in ascolto tramite la porta 80.

Esistono quattro tipi di servizi che è possibile definire: `ClusterIP`, `NodePort`, `ExternalName` e `LoadBalancer`, e servono a determinare il modo in cui il servizio viene esposto nella rete.

- `ClusterIP`: valore predefinito, per cui il servizio è accessibile solo all'interno del cluster Kubernetes: ciò vuol dire che non è possibile effettuare richieste ai propri pod dall'esterno del cluster.
- `NodePort`: questo rende il servizio accessibile su una porta statica su ciascun nodo nel cluster. Ciò significa che il servizio può gestire richieste che provengono dall'esterno del cluster.
- `LoadBalancer`: il servizio diventa accessibile esternamente tramite la funzionalità di bilanciamento del carico di un provider cloud, come per esempio GCP, AWS, Azure e OpenStack. Il provider cloud creerà un *load balancer*, che inoltrerà automaticamente le richieste al servizio Kubernetes.
- `ExternalName`: questo tipo associa il servizio ai contenuti del campo `externalName` (per esempio `pluto.miosito.com`) e lo fa restituendo un valore per il record `CNAME`.

Per vederne il funzionamento nel dettaglio, proviamo una serie di esempi, in cui mostriamo, anche graficamente, il funzionamento del sistema; cominciamo dicendo che le quattro tipologie di servizi lavorano in maniera molto diversa, ma tutti, tranne il servizio di tipo `ExternalName`, sono strettamente correlati. Se infatti andassimo a creare un `NodePort`, questo creerebbe anche un servizio `ClusterIP`; se creassimo un `LoadBalancer`, questo andrebbe a creare un `NodePort` e di conseguenza un `ClusterIP`.

Fatta questa premessa, immaginiamo di avere la seguente tipologia: due nodi e due pod, aventi gli indirizzi IP specificati nella Figura 10.14; se uno dei due pod venisse arrestato, l'altro non potrebbe più raggiungerlo e avremmo un disservizio. Sappiamo per certo che tutti i pod possono comunicare tra loro usando gli indirizzi IP interni, anche se si trovano su nodi diversi; sappiamo anche che esistono diverse strategie di riavvio e di gestione della rete degli elementi di Kubernetes, ma immaginiamo che i due pod non possano più comunicare.

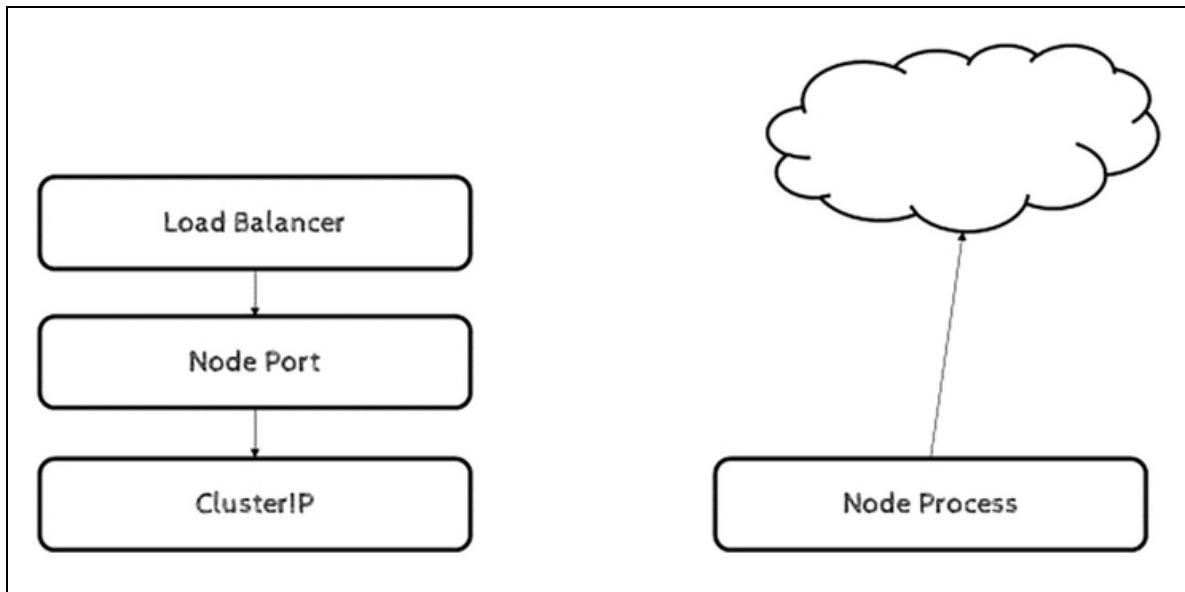


Figura 10.13 Rappresentazione della cooperazione tra le tipologie di servizio.

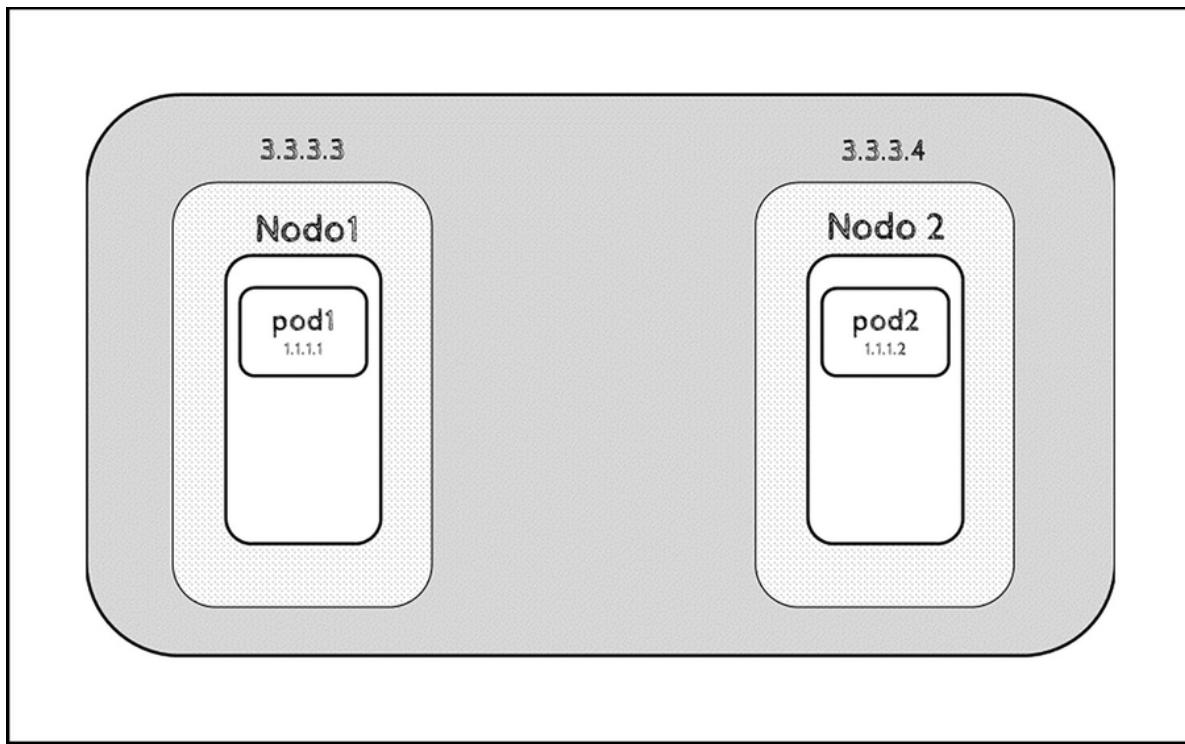


Figura 10.14 Esempio di cluster senza l'uso di servizi.

In questa situazione, interviene il servizio: utilizzando un ClusterIP, questo viene reso disponibile all'interno del cluster e quindi è

raggiungibile dai diversi nodi; possiamo immaginarlo come un elemento che si pone in mezzo alla comunicazione tra i due e che permette ai nodi di interrogarlo per richiedere di poter inviare dati a un altro nodo, come illustrato nella Figura 10.15.

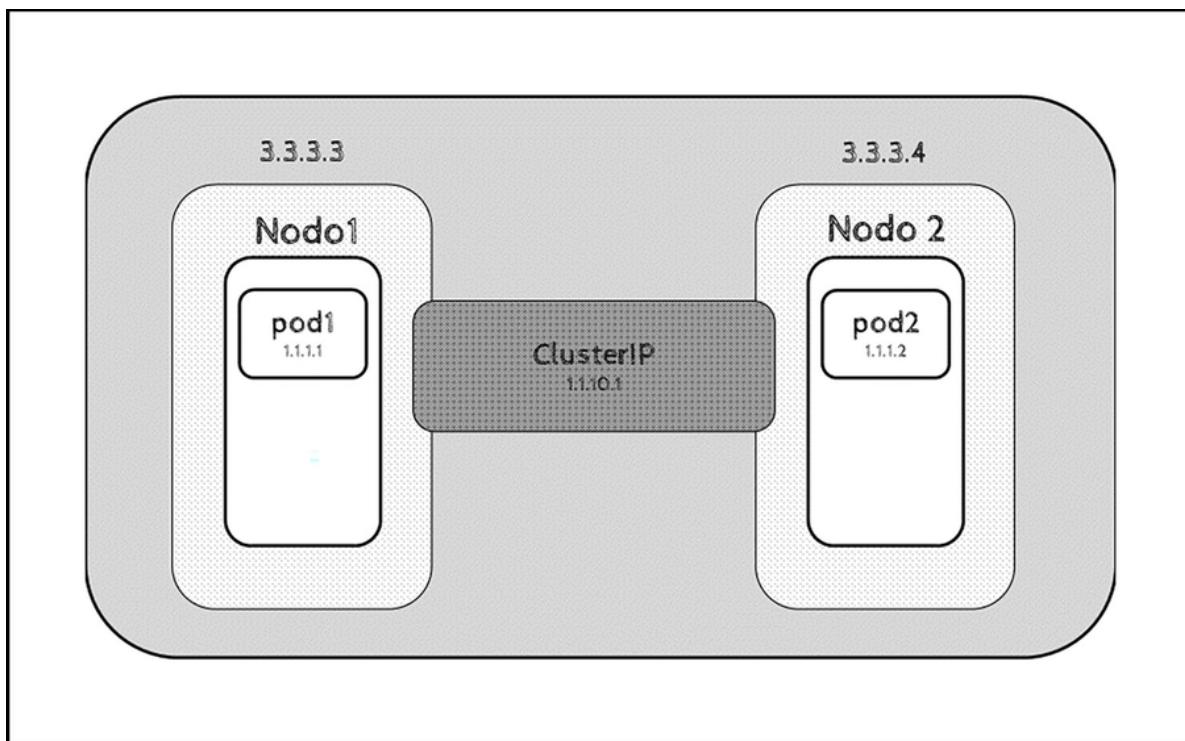


Figura 10.15 Esempio modificato tramite l'uso di un servizio ClusterIP.

Questo ci permette di gestire facilmente anche l'aggiunta di altri pod nel corso dello sviluppo e della progettazione del cluster: quando l'architettura si evolverà, non avremo bisogno di lavorare sui singoli pod, ma potremo modificare il servizio in modo che questo raggiunga anche i nuovi arrivati; il servizio di tipo `ClusterIP` distribuisce infatti le richieste in base a un approccio casuale o tramite strategia *round robin*, rendendo i pod disponibili all'interno del cluster tramite un nome e un indirizzo IP.

Listato 10.42 Il file clusterip.yaml.

```
apiVersion: v1
kind: Service
metadata:
  name: service-test
spec:
  ports:
  - port: 3000
    protocol: TCP
```

```

targetPort: 443
selector:
  run: pod2

type: ClusterIP

```

Se ora volessimo rendere disponibile anche dall'esterno il servizio appena creato, possiamo utilizzare un `NodePort`; questo ci permetterà di definire una porta attraverso la quale un nodo che riceva una richiesta, possa rimandarne la gestione al servizio, che sceglierà a quale pod comunicare le informazioni. Supponiamo dunque di creare un servizio di questo tipo, al posto di quello precedente, e di definire come porta la 30033: nella Figura 10.16 vediamo in che modo viene gestita la ricezione delle richieste dall'esterno e in che modo il servizio le instrada verso il pod appropriato.

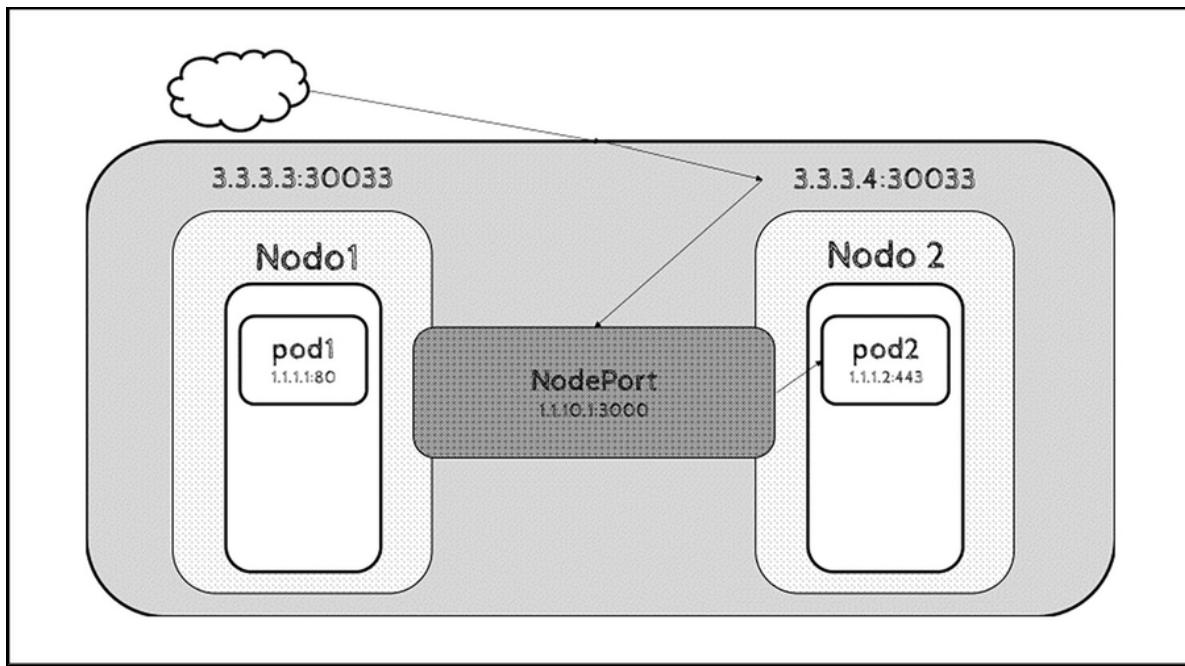


Figura 10.16 Esempio modificato tramite l'uso di un servizio NodePort.

In questo caso, stiamo definendo un comportamento tale per cui tutte le richieste ricevute dall'esterno verso la porta 30033, vengono dirottate al servizio `NodePort` con porta 3000 e inviate ai pod con porta 443. Per renderlo ancora più esplicito, mostriamo il codice del file YAML di questo servizio.

Listato 10.43 Il file nodeport.yaml.

*****ebook converter DEMO

Watermarks*****

```
apiVersion: v1
```

```
kind: Service
```

```
metadata:
```

```
  name: service
```

```
spec:
```

```
  ports:
```

```
    - port: 3000
```

```
      protocol: TCP
```

```
      targetPort: 443
```

```
      nodePort: 30033
```

```
  selector:
```

```
    run: pod2
```

```
type: NodePort
```

Questo renderà il servizio raggiungibile da ogni nodo interno al cluster, così come dall'esterno, attraverso la sola porta 30033.

Il servizio di tipo `LoadBalancer` ha un funzionamento analogo, ma viene scelto se si desidera avere un singolo indirizzo IP di riferimento per distribuire le varie richieste ai nodi presenti nel cluster.

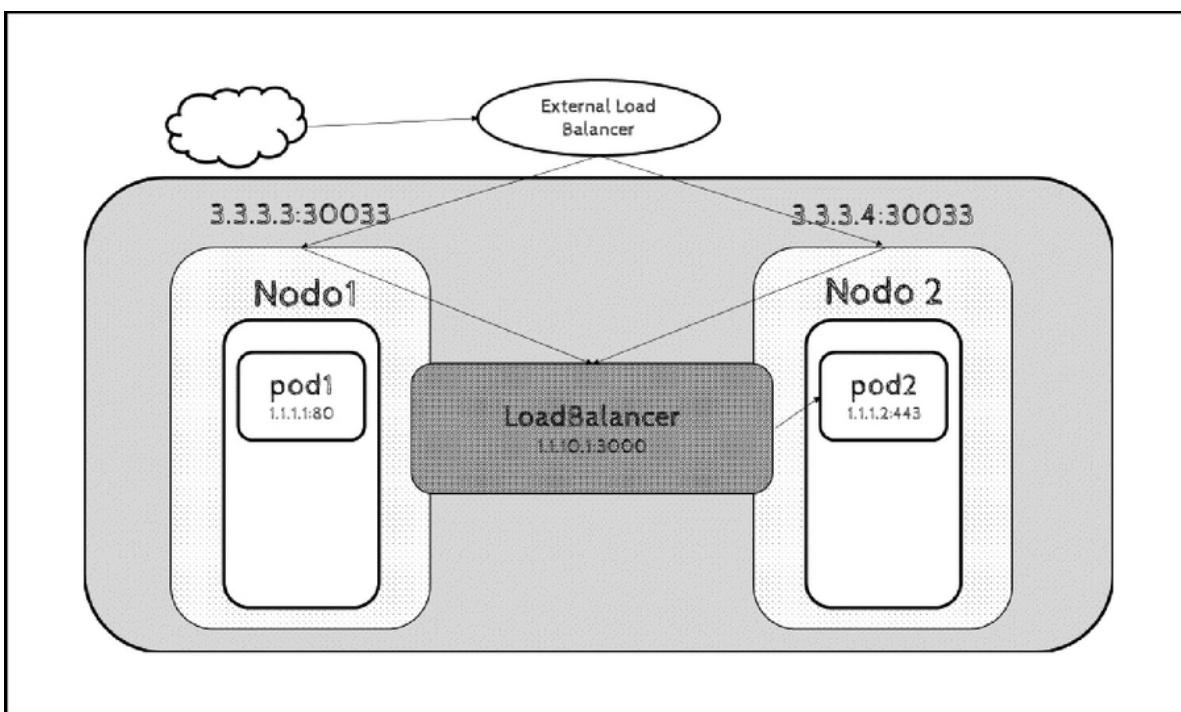


Figura 10.17 Esempio modificato tramite l'uso di un servizio Load Balancer.

Listato 10.44 loadbalancer.yaml.

```
apiVersion: v1
```

```
kind: Service
```

```
metadata:
```

```
  name: service-test
```

```
spec:
```

*****ebook converter DEMO

Watermarks*****

```

ports:
- port: 3000
  protocol: TCP
  targetPort: 443
  nodePort: 30033
selector:
  run: pod2

type: LoadBalancer

```

Infine, il servizio `ExternalName`: questo crea direttamente un *endpoint* che punta a un nome DNS, creando però un servizio interno; riprendendo l'esempio iniziale, immaginiamo che uno dei pod debba raggiungere un servizio di API esterne e che questa situazione debba essere integrata all'interno della soluzione; creando un servizio di questo tipo, possiamo farlo senza dover aggiungere il servizio all'interno del cluster, ma interrogandolo direttamente dall'interno dello stesso.

Listato 10.45 Il file `externalname.yaml`.

```

apiVersion: v1
kind: Service
metadata:
  name: service-test
spec:
  ports:
  - port: 3000
    protocol: TCP
    targetPort: 443
  type: ExternalName

  externalName: remote.url.com

```

A questo punto, come vediamo nella Figura 10.18, i nodi possono utilizzare questo servizio semplicemente sfruttando l'*endpoint* `http://service-test:3000` e interrogandolo secondo necessità.

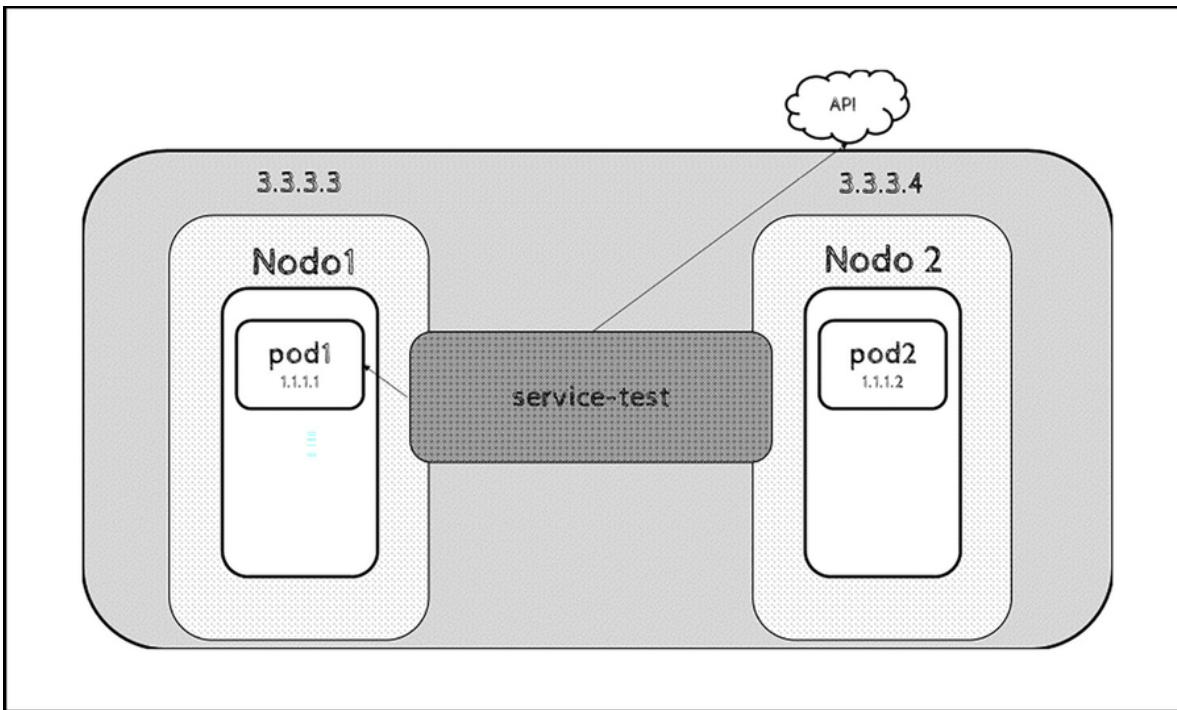


Figura 10.18 Esempio modificato tramite l'uso di un servizio ExternalName.

Gestione di un pod con container multipli

Abbiamo detto che un pod è la più piccola unità che può essere distribuita e gestita da Kubernetes. In altre parole, se è necessario eseguire un singolo container in Kubernetes, occorre creare un pod per quel container. Allo stesso tempo, un pod può contenere più container, se tali container sono strettamente accoppiati; facendo un paragone con un mondo pre-container, questi elementi sarebbero stati eseguiti sullo stesso server.

In questo paragrafo cerchiamo di capire per quale motivo i pod vengono utilizzati anche nel caso di un singolo container: per gestire un container, Kubernetes necessita di informazioni aggiuntive, come la politica di riavvio o il *live probe*. Una politica di riavvio dice che cosa fare con un container quando termina. Invece di sovraccaricare il container esistente con proprietà aggiuntive, come la gestione della propria esecuzione, gli architetti di Kubernetes hanno deciso di utilizzare

una nuova entità, ossia il *pod*, che contiene uno o più container, i quali devono essere gestiti come un'unica entità.

Perché Kubernetes consente di utilizzare più container in un pod? Perché i container vengono eseguiti su un *host logico*: tutti usano lo stesso *namespace* di rete (stesso indirizzo IP e stesso spazio di porte), lo stesso spazio dei nomi IPC e, facoltativamente, possono usare volumi condivisi. Pertanto, questi container possono comunicare in modo efficiente, garantendo un disaccoppiamento delle attività che però rende più modulare il sistema. Inoltre, i pod consentono di gestire più container di applicazioni strettamente accoppiati, che possono essere visti come un solo sistema.

Se un'applicazione avesse bisogno di più container in esecuzione sullo stesso host, perché non creare un singolo container con tutto ciò che si trova in quei container? Innanzitutto, mettere *più* cose in *un* container violerebbe il principio “un processo per container”, che abbiamo visto nel Capitolo 7; questo non viene impedito, in quanto è possibile configurare un container per effettuare più tipi di lavori, ma non rientra nelle attività che vedono Docker protagonista. In secondo luogo, l'utilizzo di più container per un'applicazione la rende più semplice da utilizzare, più trasparente e consente di disaccoppiare le dipendenze del software. Inoltre, è possibile riutilizzare i diversi container scritti con obiettivi generici per più progetti, così da non dover sempre “reinventare la ruota”.

Ci sono diverse tipologie di caso d'uso in cui vengono utilizzati più container in un solo pod: per esempio nel caso dei container cosiddetti “*sidecar*”, che hanno lo scopo di collaborare a stretto contatto con un container principale. Un esempio sono i registri o quelli che servono ad “osservare” eventuali modifiche ai dati, quelli per attività di monitoraggio e così via. Un *log watcher*, per esempio, può essere creato una volta da un team e poi riutilizzato in più applicazioni. Un altro esempio di *container sidecar* è quello che si occupa di caricare i dati in modo massivo, per fornirli poi a un container principale.

Un altro tipo di utilizzo è nel caso di proxy, bridge o adattatori, che collegano il container principale con il mondo esterno. Per esempio, il server HTTP Apache o Nginx possono servire file statici e fungere da proxy inverso per un'applicazione web nel container principale, per registrare e limitare le richieste HTTP.

Mentre è possibile ospitare in un singolo pod un'applicazione a più livelli (come WordPress), il miglior modo per sfruttarli è separarli a seconda del tipo di attività che devono svolgere. Il motivo è semplice: è possibile ridimensionare questi livelli di astrazione in modo indipendente e distribuirli tra i nodi del cluster senza alcun problema di accoppiamento.

In Kubernetes, è possibile utilizzare un volume condiviso in modo semplice ed efficiente, per condividere i dati tra container in un pod. Nella maggior parte dei casi, è sufficiente utilizzare una directory sull'host, condivisa con tutti i container all'interno di un pod. Kubernetes consente infatti ai dati di essere memorizzati anche in caso di riavvio del container, dal momento che questi hanno lo stesso ciclo di vita del pod; ciò significa che finché esiste quel pod, i dati saranno memorizzati. Se quel pod viene eliminato per qualsiasi motivo, anche se viene creata una sostituzione identica, anche il volume condiviso verrà distrutto e creato di nuovo.

Un caso d'uso standard per un pod composto da più container che usano un volume condiviso è quando un container scrive in una directory condivisa (log o altri file) e l'altro container legge dalla stessa cartella. Per esempio, supponendo di avere il seguente pod, definito tramite un file YAML, definiamo un volume denominato `data` inizializzato come `emptyDir`, ossia un volume che viene creato vuoto quando il pod viene assegnato a un nodo e che esiste fintantoché quel pod è in esecuzione su quel nodo. Il primo container esegue il server Nginx e ha il volume condiviso montato nella directory `/usr/share/nginx/html`. Il secondo container usa l'immagine Debian e ha il volume condiviso montato nella directory `/html`. Il secondo container scrive ogni secondo la data e l'ora correnti nel file `index.html` che si trova all'interno del volume condiviso. Il server Nginx legge questo file e lo aggiornaogniqualvolta l'utente invia una richiesta HTTP al server web.

Listato 10.46 Il file pod.yaml.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
spec:
  volumes:
  - name: html
    emptyDir: {}
  containers:
```

```

- name: pod1
  image: nginx
  volumeMounts:
  - name: html
    mountPath: /usr/share/nginx/html
- name: pod2
  image: debian
  volumeMounts:
  - name: data
    mountPath: /html
  command: ["/bin/sh", "-c"]
  args:
  - while true; do
    date >> /html/index.html;
    sleep 1;

done

```

Abbiamo detto che i container in un pod sono accessibili tramite `localhost`, poiché usano lo stesso `namespace` di rete. Per quanto riguarda i singoli container, l'`hostname` è il nome del pod. Poiché i container condividono lo stesso indirizzo IP e le stesse porte, è necessario utilizzare porte diverse per i container per gestire le connessioni in entrata. Per questo motivo, le applicazioni in un pod devono coordinare l'utilizzo delle porte.

Nel seguente esempio, creeremo un pod multi-container, in cui Nginx funge da proxy inverso in un container, e nell'altro espone una semplice applicazione web.

A questo scopo, usiamo un oggetto di tipo `ConfigMap`: si tratta di un dizionario che contiene più impostazioni di configurazione. Questo dizionario è costituito da coppie di stringhe chiave-valore, che Kubernetes fornirà ai container. Come accade con altre tipologie di dizionari (per esempio con le mappe) la chiave consente di recuperare e modificare il valore impostato.

Il fatto di utilizzare una `ConfigMap` mantiene il codice dell'applicazione separato dalla propria configurazione; è una parte importante della creazione di un'applicazione che mantenga il principio dei “dodici fattori”: ciò consente di modificare facilmente la configurazione in base all’ambiente (sviluppo, produzione oppure test) e di modificare dinamicamente la configurazione in fase di esecuzione.

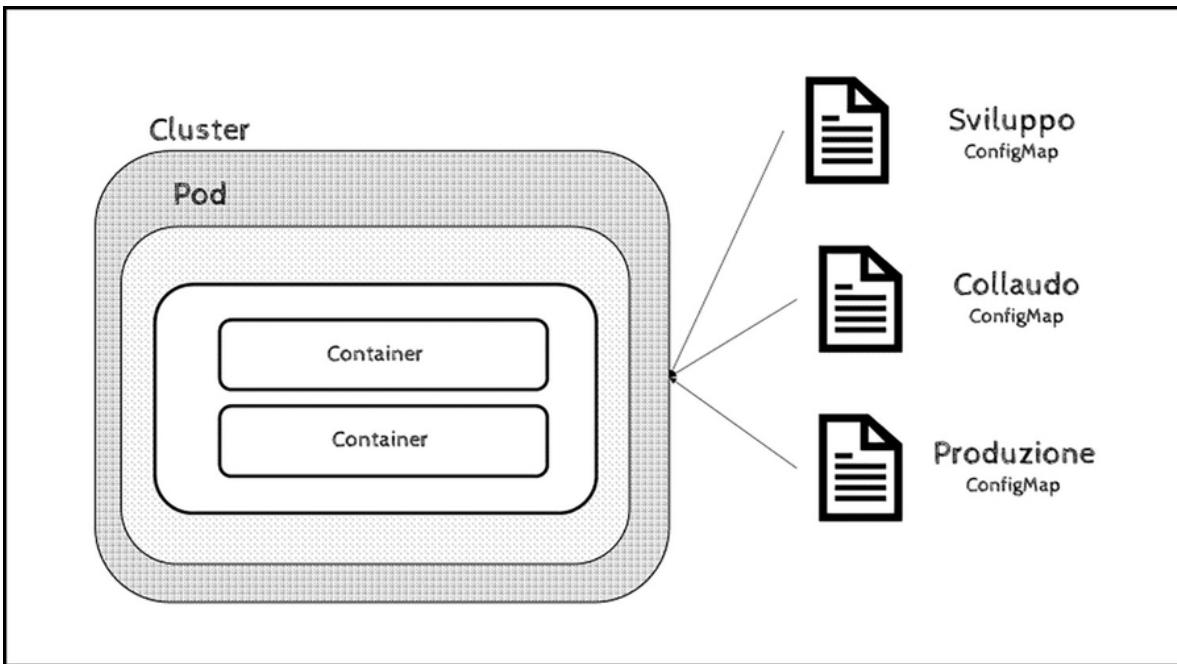


Figura 10.19 Configurazioni differenti per un'unica base di codice.

Creiamo dunque un oggetto `ConfigMap` che configuri l'ambiente del server Nginx; in questo modo, definiremo l'inoltro delle richieste HTTP ricevute sulla porta 80 verso la porta 5000.

Listato 10.47 Il file configMap.yaml.

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-conf
data:
  nginx.conf: |-
    user nginx;
    worker_processes 1;
    error_log /var/log/nginx/error.log warn;
    pid        /var/run/nginx.pid;
    events {
      worker_connections 1024;
    }
    http {
      include      /etc/nginx/mime.types;
      default_type application/octet-stream;
      log_format main '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';
      access_log  /var/log/nginx/access.log  main;
      sendfile      on;
      keepalive_timeout 65;
      upstream myapp {
        server 127.0.0.1:5000;
      }
    }
  
```

```

server {
    listen 80;
    location / {
        proxy_pass      http://myapp;
        proxy_redirect off;
    }
}

```

Nell'ultima parte del file, possiamo leggere la configurazione che gestisce la rete: il server sarà in ascolto sulla porta 80, e “passerà” le richieste http alla porta 5000 del container in cui è presente l'applicazione. Chiaramente, come già ribadito, queste due porte sono visibili solo e soltanto all'interno del container, mentre il pod avrà una sua porta con cui l'utente potrà comunicare.

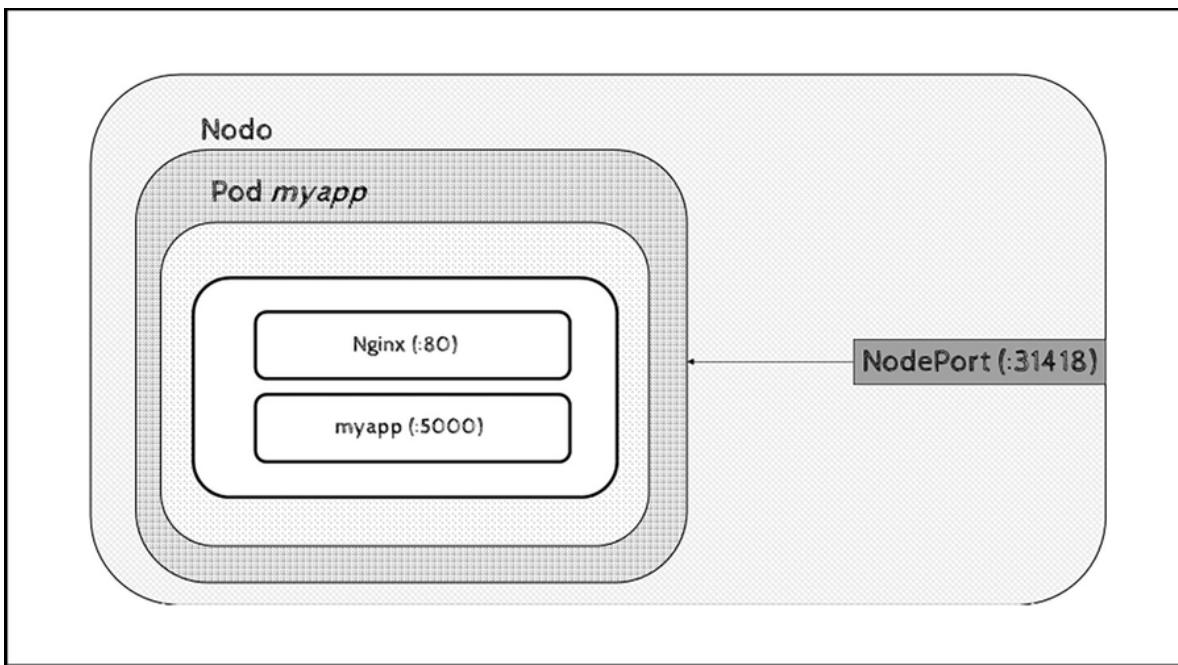


Figura 10.20 Rappresentazione della rete del nodo.

Creiamo a questo punto un pod che prevedrà due container distinti: uno conterrà una semplice app web (denominata `myapp`) e l'altro conterrà il server Nginx. È da notare che per il pod viene definita la sola esposizione della porta Nginx 80; la porta 5000 non sarà accessibile al di fuori del pod; Importeremo il file di configurazione creato in precedenza tramite l'attributo `configMap`. Cambiando ambiente, ma mantenendo il nome del file, sarà dunque possibile utilizzare delle configurazioni *ad hoc* per

l’ambiente prescelto, senza dover fare alcun tipo di modifica alla definizione del pod.

Listato 10.48 Il file pod.yaml.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  labels:
    app: pod1
spec:
  containers:
    - name: myapp
      image: training/webapp
    - name: nginx
      image: nginx:alpine
      ports:
        - containerPort: 80
      volumeMounts:
        - name: nginx-proxy-config
          mountPath: /etc/nginx/nginx.conf
          subPath: nginx.conf
      volumes:
        - name: nginx-proxy-config
          configMap:
            name: nginx-conf
```

Per far sì che la porta del pod sia esposta, lanciamo il primo comando, e poi identifichiamo la porta del nodo, per poi raggiungere l’applicazione tramite browser.

Listato 10.49 Avvio del pod.

```
root@master:/# kubectl expose pod myapp --type=NodePort --port=80
root@master:/# kubectl describe service myapp
...
NodePort:      <unset>   31418/TCP
...
```

Utilizzo dei volumi

Il tema dei volumi è stato lasciato da parte per fare più spazio agli elementi base di Kubernetes; così come per Docker, questa tecnologia nasce come soluzione *stateless*, quindi il presupposto fondamentale è che i suoi componenti non mantengano alcuna informazione in maniera permanente, a meno di utilizzare strumenti come i *volumi*. Questo concetto, che abbiamo visto per Docker, in Kubernetes è gestito

diversamente; ne esistono infatti due tipi principali, ovvero i volumi persistenti e le attestazioni di volumi persistenti.

- Un *volume persistente* (abbreviato in *PV*) è un'area di memoria di rete del cluster che viene fornita da un amministratore. È una risorsa nel cluster così come un nodo; i PV hanno un ciclo di vita indipendente da ogni singolo pod che li utilizza e questo fa sì che anche nel caso in cui a un oggetto K8S venga associato un volume persistente, se l'oggetto viene distrutto, i suoi dati non andranno persi.
- Un'*attestazione di volume persistente* (o *PVC*, da *Persistent Volume Claim*) è una richiesta di archiviazione da parte di un utente ed è simile al concetto di pod. I pod sfruttano le risorse di un nodo, mentre i PVC consumano le risorse di un PV. Questi volumi possono richiedere dimensioni e modalità di accesso specifiche (per esempio, possono essere montati una volta in lettura e scrittura oppure in sola lettura).

Mentre le attestazioni di volumi persistenti consentono a un utente di utilizzare risorse di archiviazione astratte, è usuale che gli utenti abbiano bisogno di volumi persistenti con proprietà diverse, come l'ottimizzazione delle prestazioni, per problemi diversi. Gli amministratori del cluster devono essere in grado di offrire una varietà di volumi persistenti che differiscono in più modi per dimensioni e modalità di accesso, senza coinvolgere gli utenti nei dettagli implementativi di tali volumi; a questo scopo interviene il concetto di `StorageClass`.

Una `StorageClass` fornisce agli amministratori un modo per descrivere le astrazioni o gli schemi del tipo di archiviazione che offrono. Classi diverse potrebbero essere associate a livelli di qualità del servizio, a politiche di backup o a politiche arbitrarie, determinate dagli amministratori del cluster.

Un esempio di file `StorageClass` è il seguente.

Listato 10.50 Esempio di `StorageClass`.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
```

```

provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Retain
allowVolumeExpansion: true
mountOptions:
  - debug
volumeBindingMode: Immediate

```

Come possiamo notare, viene specificato un attributo `provisioner` che specifica il plugin del volume utilizzato per creare il PV; ne esistono diverse tipologie, forniti da diversi provider molto conosciuti, come AWS, GCE o Azure.

I volumi persistenti creati dinamicamente da una `StorageClass` avranno la politica di recupero specificata nel campo `reclaimPolicy` della classe, che può essere `Delete` o `Retain`. Se non viene specificato alcun valore per `reclaimPolicy` quando viene creato un oggetto `StorageClass`, verrà automaticamente impostato su `Delete`.

I volumi persistenti possono essere configurati per essere espandibili; tramite il campo `allowVolumeExpansion` possiamo ridimensionare il volume modificando l'oggetto PVC corrispondente.

Il campo `volumeBindingMode` controlla invece quando deve verificarsi l'associazione di volumi e la creazione dei volumi *on-demand*; per impostazione predefinita, la modalità `Immediate` che abbiamo utilizzato anche nell'esempio indica che l'associazione del volume e il *provisioning* si verificano subito dopo la creazione del PVC.

Vediamo dunque un esempio di gestione dei volumi persistenti e delle attestazioni con un progetto pratico, dove creiamo una distribuzione basata su Node.js e MongoDB; immaginiamo di creare un PV di 10GB, ma supponendo che le distribuzioni del cluster richiedano solo 3GB ognuna.

Come accennato in precedenza, in questa procedura utilizzeremo i file di configurazione YAML per creare gli oggetti; sono sicuramente il modo più semplice e comprensibile per gestire gli oggetti in K8S.

Iniziamo creando una directory di lavoro in cui conserveremo tutti i file YAML, e la chiamiamo `cluster`. Dopo averla creata, al suo interno definiamo un file `mongo-pv.yml`, dove definiremo il PV che utilizzeremo per la nostra applicazione.

Listato 10.51 Il file `/cluster/mongo-pv.yml`.

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongo-pv
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: "/mnt/data/mongo"

```

Nel file appena creato stiamo definendo la capacità del PV pari a 10 GB; per crearlo, lanciamo il comando seguente.

Listato 10.52 Comando per creare il PV.

```
root@master:/# kubectl create -f ./cluster/mongo-pv.yml
```

Una volta eseguito il comando, verifichiamo che il PV sia stato correttamente creato, lanciando il comando `kubectl get`.

Listato 10.53 Comando per elencare i PV.

```
root@master:/# kubectl get persistentvolumes
```

Ora occupiamoci del PVC: anche in questo caso, creiamo un file di configurazione, dove definiremo l'attestazione di volume persistente con una capacità pari a 3 GB.

Listato 10.54 Il file /cluster/mongo-pvc.yml.

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongo-pvc
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi

```

Eseguiamo nuovamente il comando `kubectl create` passando come parametro il file che abbiamo appena creato ed eseguiamo il comando `kubectl get` per verificare che la creazione sia andata a buon fine.

A questo punto possiamo creare l'applicazione con MongoDB e Node.js; avremo bisogno di un file YAML per creare la distribuzione di entrambi e dei servizi che permettono loro di essere utilizzati tramite il cluster: procediamo dunque con la definizione della distribuzione e del servizio per MongoDB, dove useremo l'immagine di base disponibile sui registri ufficiali e aggiungiamo dei punti di accesso ai volumi creati in precedenza, che permettano di memorizzare in maniera permanente le informazioni contenute nel database.

Listato 10.55 Il file /cluster/mongo-deploy.yml.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongo
spec:
  selector:
    matchLabels:
      app: mongo
      role: master
      tier: backend
  replicas: 1
  template:
    metadata:
      labels:
        app: mongo
        role: master
        tier: backend
    spec:
      containers:
        - name: mongo
          image: mongo
          ports:
            - containerPort: 27017
          volumeMounts:
            - name: data
              mountPath: /data/db
      volumes:
        - name: data
          persistentVolumeClaim:
            claimName: mongo-pvc
```

Eseguiamo il comando `kubectl create` per creare la distribuzione tramite il file appena definito; nell'attributo `claimName` sotto `persistentVolumeClaim` abbiamo utilizzato il nome assegnato in precedenza al PVC, ossia `mongo-pvc`. Inoltre, abbiamo definito un alias per questo elemento, `mongodb`, così da poterlo richiamare con facilità.

Definiamo ora il servizio.

Listato 10.56 Il file /cluster/mongo-service.yml.

```

apiVersion: v1
kind: Service
metadata:
  name: mongo
  labels:
    app: mongo
    role: master
    tier: backend
spec:
  ports:
    - port: 27017
      targetPort: 27017
  selector:
    app: mongo
    role: master
    tier: backend

```

Questo servizio troverà tutte le distribuzioni che abbiano l'attributo `app` impostato come `mongo` e il `tier` impostato come `backend` per esporli nella rete del cluster. La configurazione delle porte assicurerà che il traffico passi attraverso la porta predefinita di MongoDB, ossia la 27017. Come al solito, eseguiamo il comando `kubectl create` per creare l'oggetto.

Passiamo alla parte relativa a Node.js: i file YAML sono molto simili a quelli definiti in precedenza, quindi avremo un file per la distribuzione e uno per il servizio.

Listato 10.57 Il file /cluster/nodejs-deploy.yml.

```

apiVersion: v1
kind: Deployment
metadata:
  name: node
spec:
  selector:
    matchLabels:
      app: node
      tier: backend
  replicas: 6
  template:
    metadata:
      labels:
        app: node
        tier: backend
    spec:
      containers:
        - name: node
          image: adnanrahic/boilerplate-api:latest
          ports:
            - containerPort: 3000
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1

```

Listato 10.58 Il file /cluster/nodejs-service.yml.

```
apiVersion: v1
kind: Service
metadata:
  name: node
  labels:
    app: node
    tier: backend
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 3000
  selector:
    app: node
  tier: backend
```

Possiamo vedere da questi file che abbiamo aggiunto questa distribuzione a quella di MongoDB tramite l'attributo `tier`; in aggiunta, creeremo sei repliche e imposteremo una strategia di aggiornamento continuo (`rollingUpdate`) per garantire il massimo tempo di attività anche durante gli aggiornamenti.

L'immagine da cui verranno costruiti i container è una piccola API di base che è disponibile su Docker Hub e che gestisce un'autenticazione di base, il che la rende perfetta per sottoporre a test il cluster e il volume persistente.

Il file di configurazione del servizio fungerà da *load balancer*, ossia si occuperà di indirizzare tutto il traffico attraverso la porta 80 verso tutti i pod replicati nella distribuzione.

Terminata la scrittura di questi file, non ci resta che eseguire il comando `kubectl create` per entrambi gli oggetti e poi possiamo procedere con il test dell'applicazione: intanto, controlliamo che tutte le risorse utilizzate siano attive e funzionanti, lanciando il comando `kubectl get all`. Se tutti gli oggetti sono presenti, possiamo utilizzare un client per sottoporre a test le API come Postman o Insomnia, eseguendo una chiamata all'indirizzo IP del cluster; il servizio Node.js può essere chiamato tramite i seguenti *endpoint*, che permettono di sottoporre a test il servizio oppure di creare un nuovo utente, passando come parametro un JSON che contenga l'indirizzo email, il nome e la password.

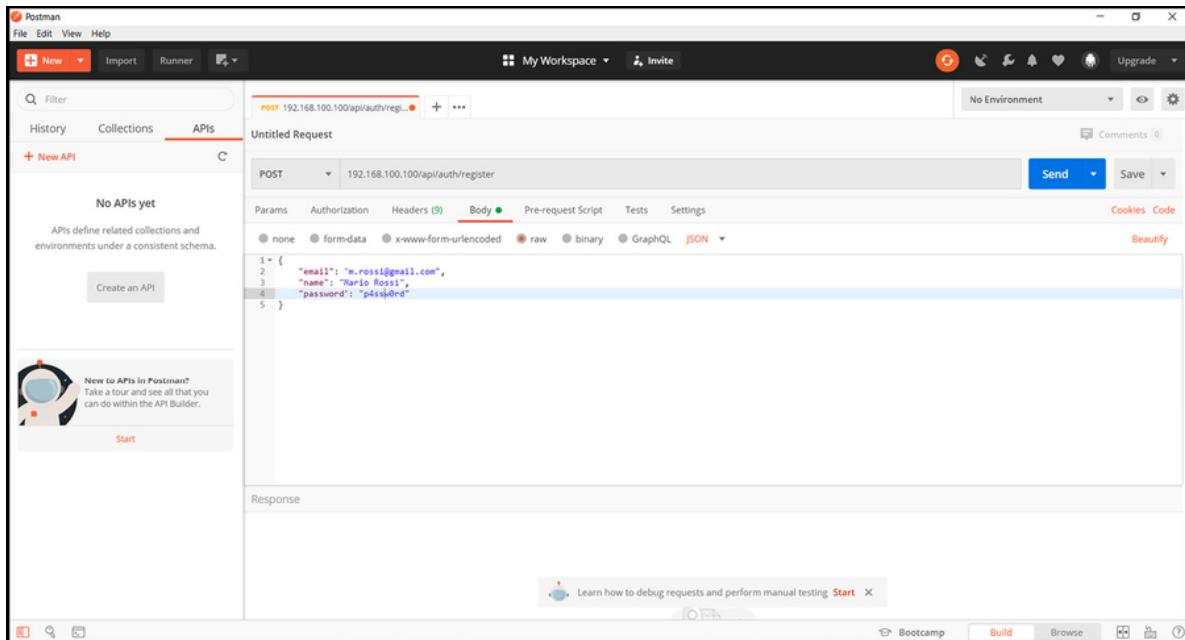


Figura 10.21 Chiamata API per creare un nuovo utente.

Che cosa abbiamo imparato

- Cos'è Kubernetes (K8s) e come è possibile rappresentare la sua architettura di base.
- Che funzione hanno i componenti di base e quali sono le differenze tra nodi master e nodi.
- Quali elementi costituiscono il master e che funzione hanno.
- Quali elementi costituiscono i nodi e come funzionano.
- Che cosa sono i pod, che ruolo hanno all'interno di Kubernetes e in che modo possono essere replicati.
- Cos'è un servizio e in cosa è differente da una distribuzione.
- Come avviare un pod e come utilizzare una modalità di lavoro dichiarativa piuttosto che imperativa per definirne la struttura.
- Come gestire un servizio e quanti tipi di servizi esistono.
- Come gestire un pod composto da più container e come permettere loro di comunicare.

- Quali tipologie di volumi esistono, che differenze ci sono e in che modo utilizzarli per creare una soluzione che memorizzi in maniera persistente i dati.
- Come creare un'applicazione che utilizza Kubernetes e i vari oggetti che lo compongono.

Docker su Cloud

Con l'avvento dei provider cloud pubblici e privati, le aziende hanno spostato un numero crescente di carichi di lavoro usando queste tecnologie. Una parte significativa dell'infrastruttura IT viene ora fornita su cloud pubblici come Amazon Web Services (AWS), Google Compute Engine (GCE) e Microsoft Azure (Azure). Inoltre, le aziende hanno implementato cloud privati per fornire un'infrastruttura self-service per le esigenze IT.

Sebbene Docker, come qualsiasi software, sia spesso eseguito su server cosiddetti *bare metal*, l'esecuzione di un host Docker in un cloud pubblico o privato (ovvero, su macchine virtuali) e di conseguenza l'orchestrazione di container avviati su tali host è sempre più una parte fondamentale delle nuove esigenze dell'infrastruttura IT.

La Figura 11.1 mostra una semplice configurazione in cui si accede a un host Docker remoto nel cloud, utilizzando il client Docker locale.

Questo capitolo tratta tre cloud pubblici (per esempio AWS, GCE e Azure) e alcuni dei servizi Docker che offrono. Se non hai mai usato un cloud pubblico, ora è il momento di vedere come affrontare alcune nozioni di base per iniziare. Vedremo come utilizzare la riga di comando di questi cloud per avviare istanze e installare Docker. Prima vedremo la gestione tramite Google Compute Engine, come organizzare e creare dei container, come definire un registro Docker e come renderlo funzionante similmente al Docker Hub.

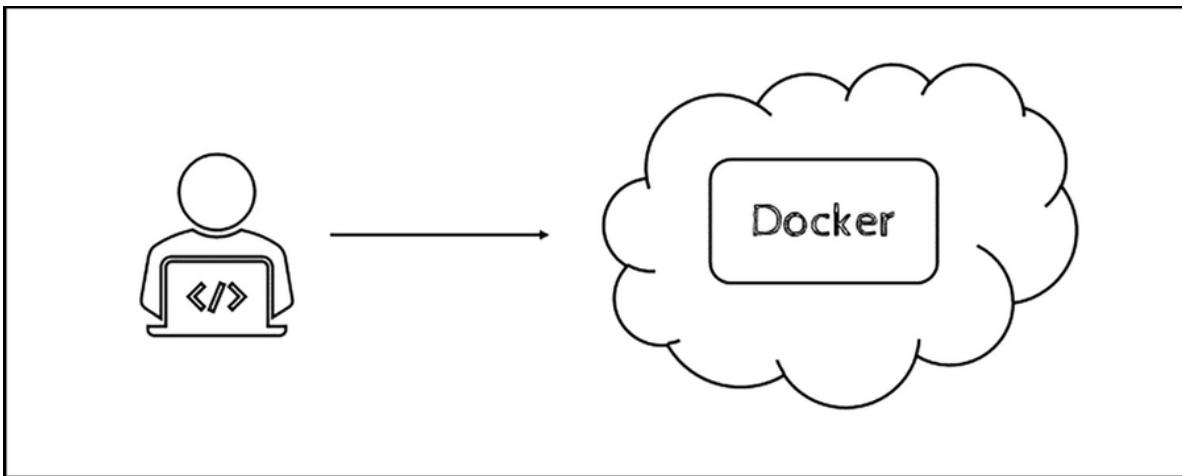


Figura 11.1 Rappresentazione dell'uso di Docker tramite Cloud Provider.

Vedremo inoltre come usare Amazon EC2, un servizio che permette la creazione di un host virtuale per installare Docker e le relative immagini, e come gestire le relative istanze.

Accedere ai cloud provider

Se non hai già un account a uno dei cloud provider pubblici sopracitati, è il momento di crearlo.

- Per *GCE*, è possibile utilizzare una prova gratuita. Abbiamo solo bisogno di un account Google e potremo accedere alla console.
- Per *Azure*, è possibile iniziare con una prova gratuita.
- Per *AWS*, puoi avere accesso a un *tier* gratuito (vedremo dopo in dettaglio di che cosa si tratta); dopo aver creato un account, è possibile accedere alla console.

Accediamo alla console web del provider che intendiamo utilizzare e passiamo alla procedura guidata di un'istanza; dobbiamo assicurarci di poter avviare un'istanza alla quale sia possibile connettersi tramite ssh. Nelle Figure 11.2 e 11.3 vengono riportate le console dei due provider, i quali verranno trattati separatamente.

AWS

Amazon EC2 (*Elastic Compute Cloud*) è uno dei servizi più noti di Amazon Web Services e offre alle aziende la possibilità di eseguire applicazioni sfruttando un servizio di cloud pubblico. Gli sviluppatori possono creare istanze di macchine virtuali e configurare facilmente il ridimensionamento della capacità delle istanze utilizzando l'interfaccia web EC2.

EC2 consente inoltre agli utenti di creare app per automatizzare il ridimensionamento in base alle mutevoli esigenze e ai periodi di punta, e questo semplifica l'implementazione di server virtuali e la gestione dello storage, riducendo la necessità di investire in hardware e aiutando a semplificare i processi di sviluppo.

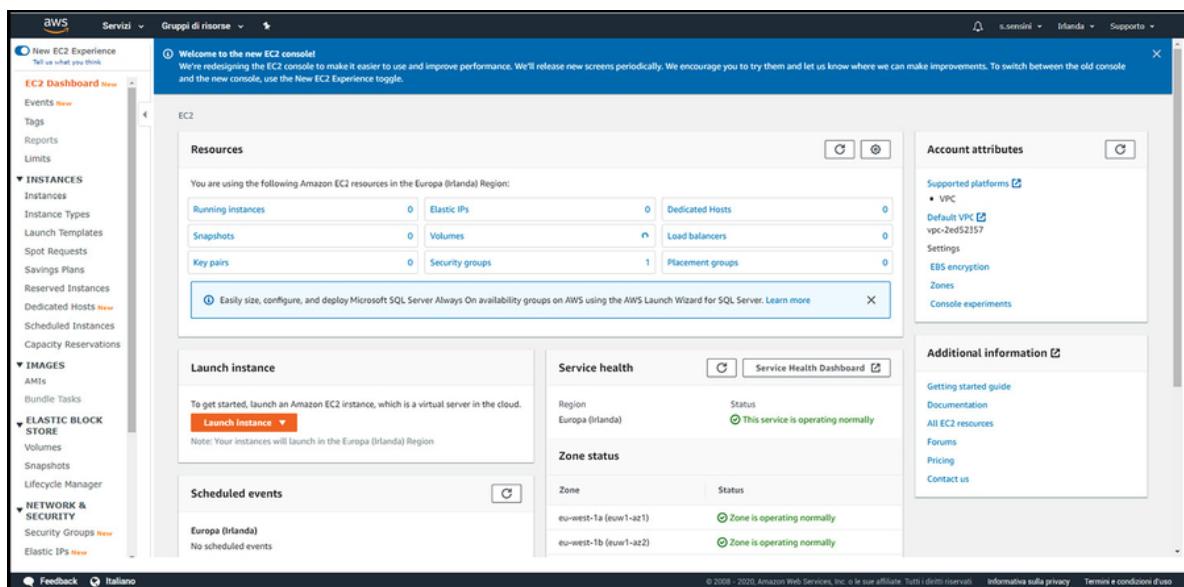


Figura 11.2 Console AWS.

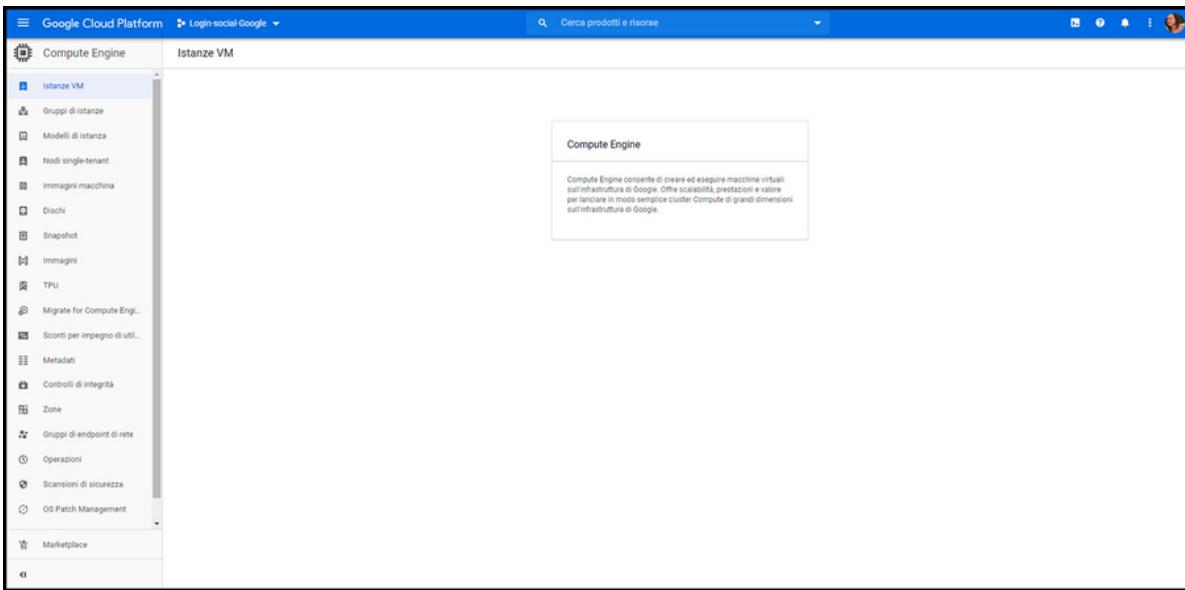


Figura 11.3 Console GCE.

La configurazione di EC2 prevede la creazione di una *Amazon Machine Image* (AMI), che include un sistema operativo, app e configurazioni. Tale AMI viene caricata su *Amazon Simple Storage Service* (S3) ed è registrata con EC2, tramite il quale gli utenti possono avviare macchine virtuali in base alle esigenze.

Amazon offre diversi tipi di istanza di EC2, per requisiti e budget diversi, con tariffe orarie, riservate oppure spot. Creando un account per la prima volta, si ha a disposizione uno spazio gratuito che è possibile utilizzare per fare alcuni test, come quello che vedremo di seguito.

Per creare un’istanza di Amazon EC2, andiamo su <https://aws.amazon.com/ec2> e facciamo clic su *Accedi alla console*. Dopo aver selezionato *EC2* tra i servizi web Amazon elencati, facciamo clic su *Istanze* per elencare le istanze Amazon EC2 già create nell’account (se è la prima volta, non sarà presente alcuna istanza). Facciamo clic su *Launch Instance* per creare una nuova istanza Amazon EC2 come mostrato nella Figura 11.4.

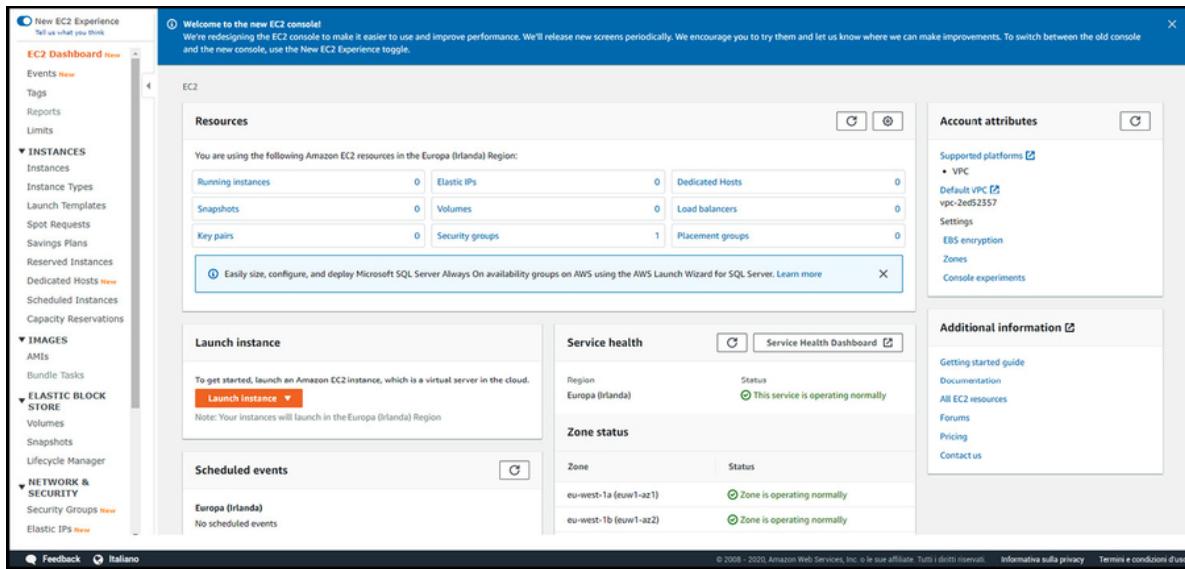


Figura 11.4 Avvio di un'istanza AWS.

Ci verrà chiesto di scegliere un'immagine di base per la macchina che utilizzeremo; sono presenti diversi modelli di macchine (chiamati AMI) con diverse tipologie di sistemi operativi, che possono essere utilizzate secondo le esigenze. Dal momento che vedremo un esempio molto semplice, possiamo utilizzare l'immagine Ubuntu 18.04 come indicato nella Figura 11.5.

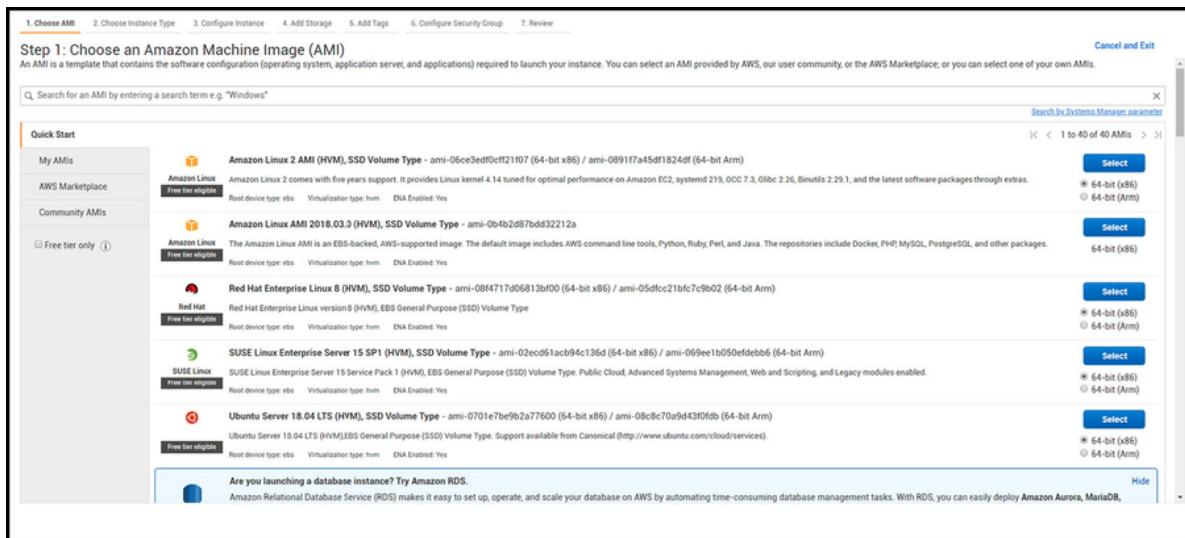


Figura 11.5 Scelta dell'AMI.

*****ebook converter DEMO
Watermarks*****

Nel passo successivo sono disponibili diversi tipi di istanze, che differiscono per funzionalità, come la capacità supportata e il numero di CPU virtuali (*vCPU*). Selezioniamo uno dei tipi di istanza disponibili, per esempio *General Purpose t2.micro* e facciamo clic su *Review and Launch*, come mostrato nella Figura 11.6.

Come ultimo passo ci verrà riepilogato quanto selezionato finora e ci verrà chiesto di lanciare l'istanza; facciamo clic su *Launch*, come indicato nella Figura 11.7.

Family	Type	vCPUs	Memory (GiB)	Instance Storage (GiB)	EBS-Optimized Available	Network Performance	IPv6 Support
General purpose	t2.nano	1	0.5	EBS only	-	Low to Moderate	Yes
General purpose	t2.micro <small>Free tier eligible</small>	1	1	EBS only	-	Low to Moderate	Yes
General purpose	t2.small	1	2	EBS only	-	Low to Moderate	Yes
General purpose	t2.medium	2	4	EBS only	-	Low to Moderate	Yes
General purpose	t2.large	2	8	EBS only	-	Low to Moderate	Yes
General purpose	t2.xlarge	4	16	EBS only	-	Moderate	Yes
General purpose	t2.2xlarge	8	32	EBS only	-	Moderate	Yes
General purpose	t3a.nano	2	0.5	EBS only	Yes	Up to 5 Gigabit	Yes
General purpose	t3a.micro	2	1	EBS only	Yes	Up to 5 Gigabit	Yes
General purpose	t3a.small	2	2	EBS only	Yes	Up to 5 Gigabit	Yes
General purpose	t3a.medium	2	4	EBS only	Yes	Up to 5 Gigabit	Yes
General purpose	t3a.large	2	8	EBS only	Yes	Up to 5 Gigabit	Yes

Figura 11.6 Scelta della tipologia di istanza.

Figura 11.7 Avvio dell'istanza.

Verrà visualizzata una finestra di dialogo per creare o selezionare una coppia di chiavi, necessaria per l'autorizzazione all'avvio dell'istanza. Per creare una nuova coppia di chiavi, selezioniamo l'opzione *Create a new key pair*, visibile nella Figura 11.8.

Specifichiamo un nome per la coppia di chiavi e facciamo clic su *Download Key Pair* (Figura 11.9). La coppia di chiavi viene così creata e scaricata. La coppia di chiavi per un'istanza Amazon EC2 durante la creazione dell'istanza è necessaria per la connessione all'istanza; si tratta di un file in formato .pem che dovremo mettere da parte, così poterla utilizzare in uno dei prossimi passaggi.

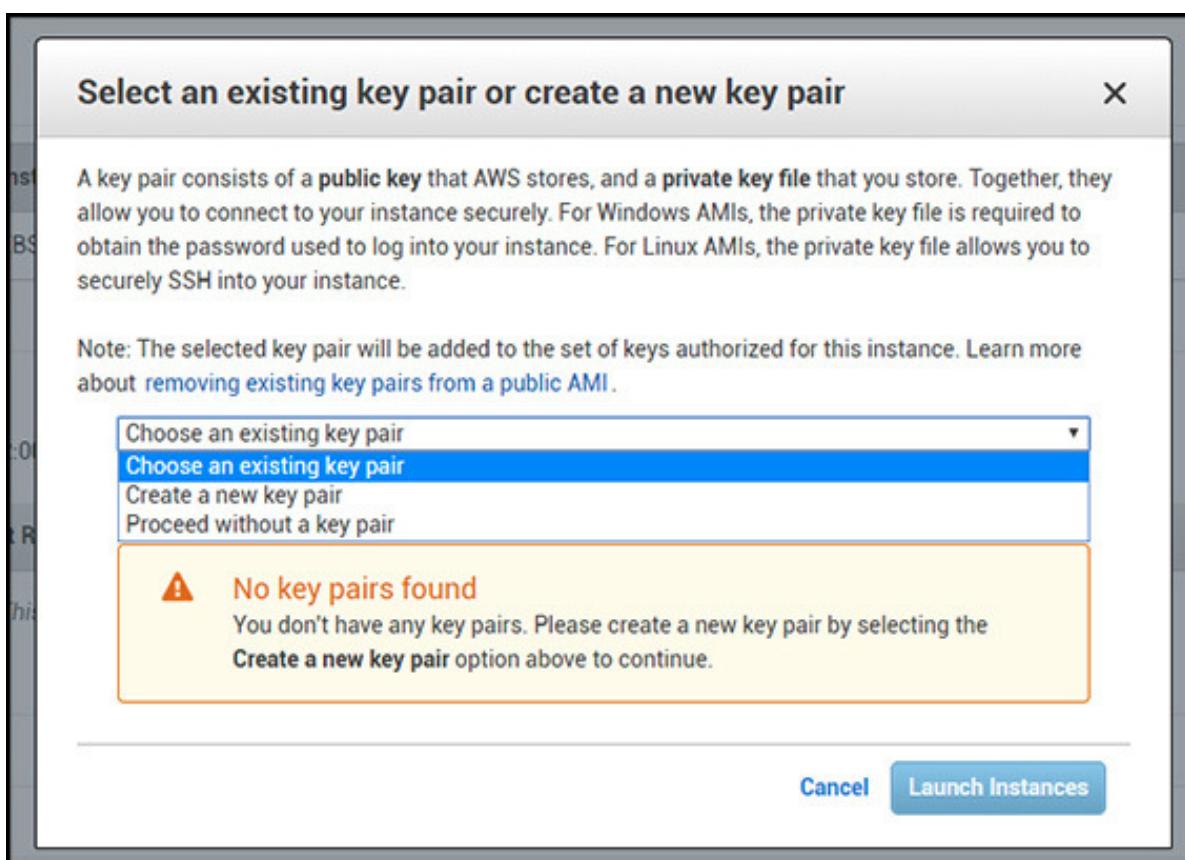


Figura 11.8 Creazione della coppia di chiavi.

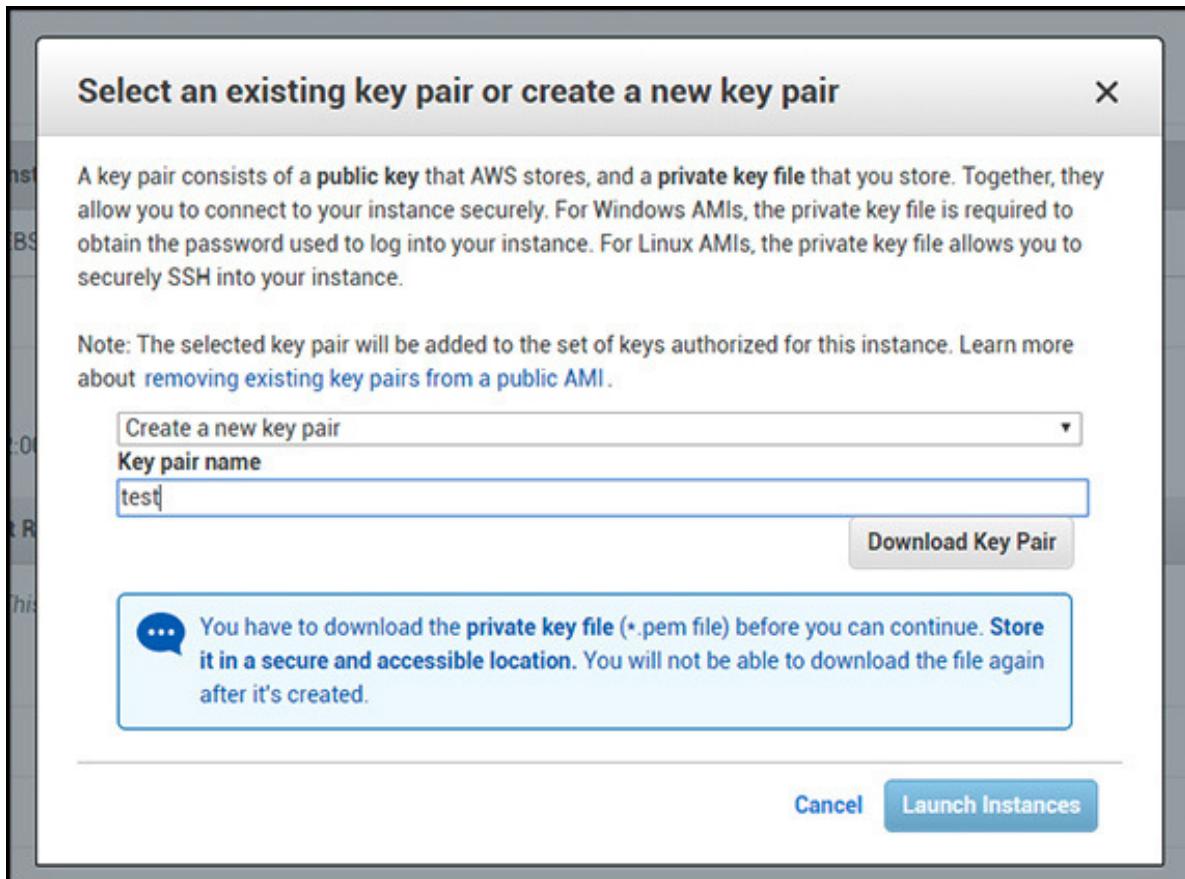


Figura 11.9 Download della coppia di chiavi.

Una volta scaricata la chiave, sarà possibile fare clic su *Launch Instances*; verrà mostrata una schermata che conferma il corretto avvio dell’istanza e da cui possiamo visualizzare ulteriori dettagli.

Un’istanza che è stata avviata può essere raggiunta tramite un computer locale come fosse un’istanza Linux locale, senza però dover disporre della stessa quantità di RAM o della stessa distribuzione Linux rispetto all’istanza alla quale è collegata. Si può avviare una *sessione ssh* da utilizzare per connettersi a un’istanza in esecuzione selezionando una delle istanze avviate e facendo clic sulla voce *Connect*, che possiamo vedere nella Figura 11.11.

Nella finestra di dialogo che viene visualizzata, vengono fornite le istruzioni per avviare un terminale tramite ssh. Per prima cosa, è necessario disporre di un client come PuTTY, che permette la connessione in remoto; inoltre è necessario recuperare la chiave in formato `.pem` che abbiamo scaricato prima, durante la fase di creazione

dell’istanza; questo file deve avere i soli permessi di lettura e scrittura per l’utente proprietario del file, e questi possono essere impostati in modo diverso a seconda del sistema operativo utilizzato.

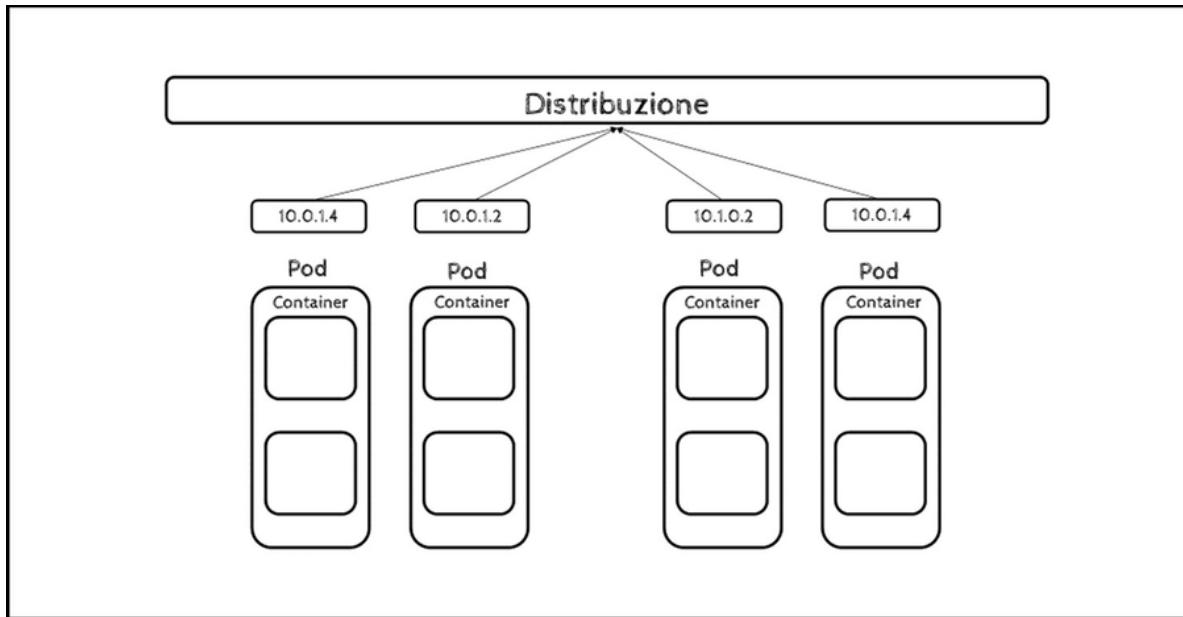


Figura 11.10 Corretto avvio dell’istanza.

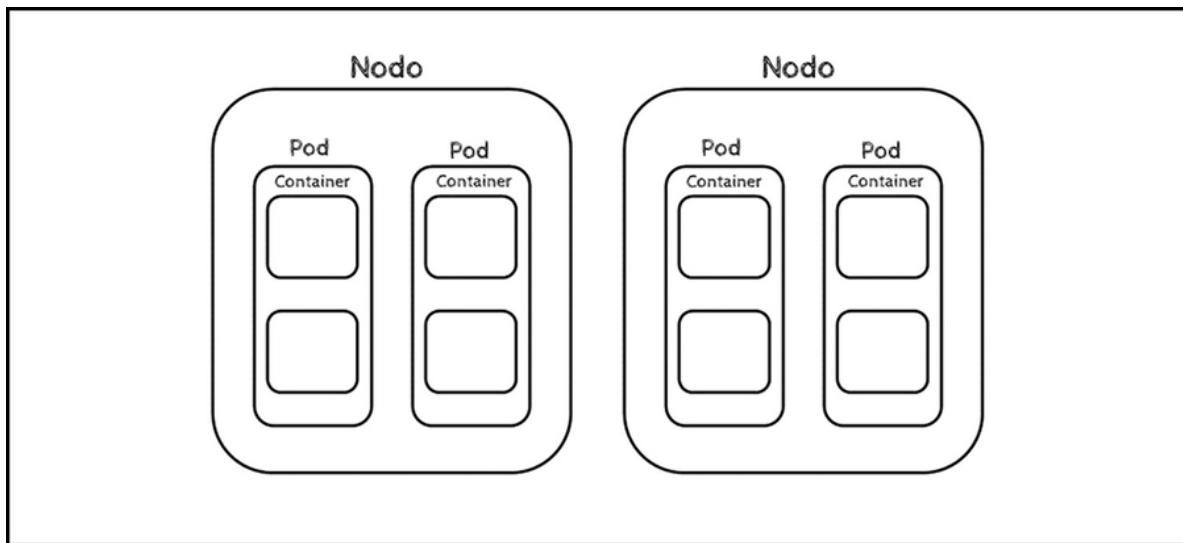


Figura 11.11 Riepilogo delle istanze.

A questo punto, possiamo accedere all’istanza utilizzando, per esempio, PuttyGen: questo converte le chiavi nel formato richiesto per

PuTTY. È infatti necessario convertire la chiave privata (il file `.pem`) nel formato supportato da PuTTY (un file `.ppk`) come descritto di seguito.

Dopo aver aperto il programma PuTTYGen, facciamo clic su *Load* e carichiamo il file `.pem` scaricato in precedenza; a questo punto, il pulsante *Save private key* sarà selezionabile e potremo salvare la chiave in formato `.ppk`. È importante salvare il file con lo stesso nome utilizzato per la coppia chiave-valore: per esempio, avendo usato la parola `test`, salveremo il file come `test.ppk`, considerando che l'estensione del file viene aggiunta automaticamente.

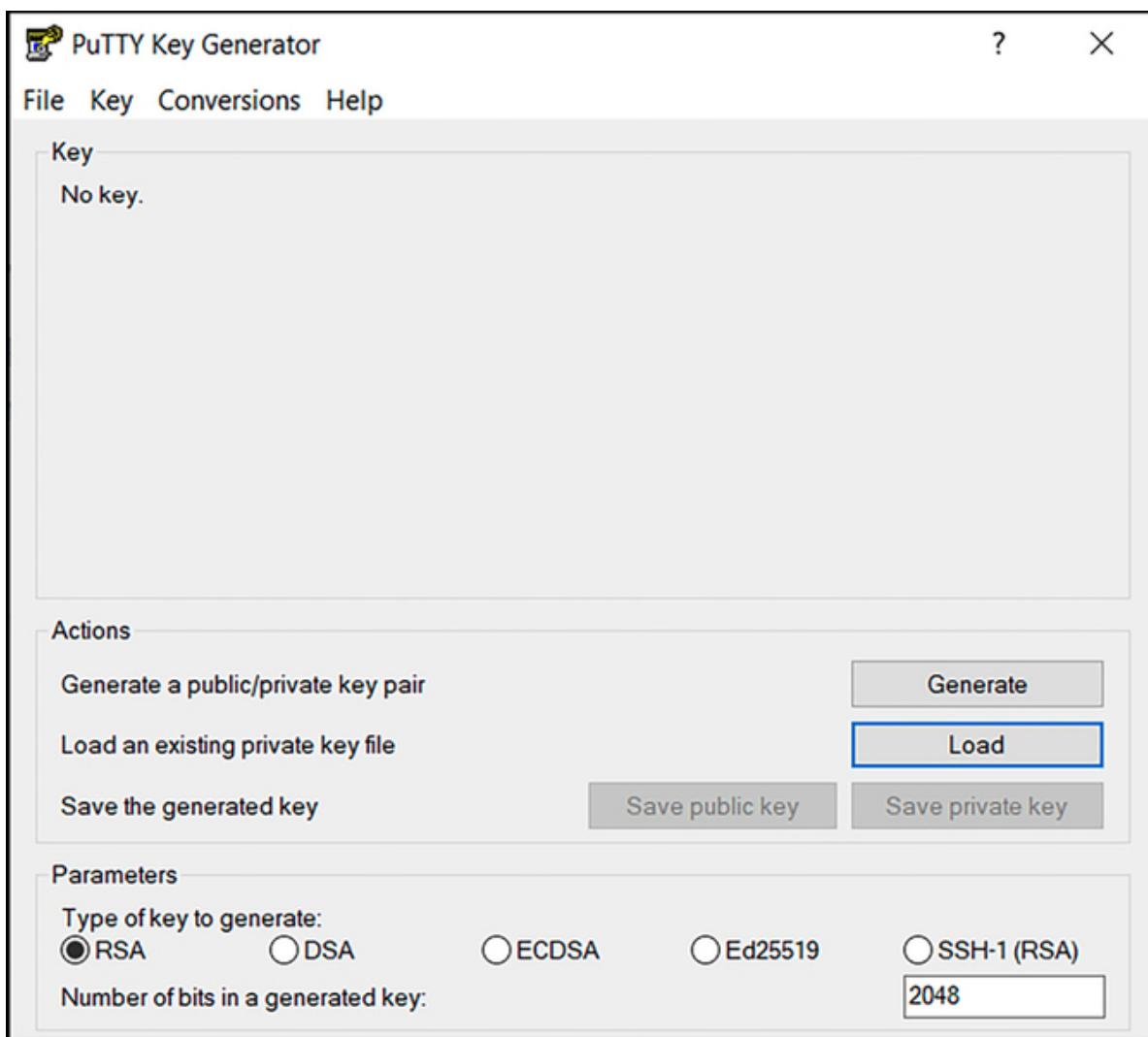


Figura 11.12 Schermata principale di PuTTYGen.

Una volta completate le istruzioni richieste, possiamo connetterci all'istanza usando PuTTY. Nella finestra principale, inseriamo come

*****ebook converter DEMO

Watermarks*****

hostname il nome utente, mostrato nella finestra della console AWS, nel seguente formato: `nome_utente@dns_pubblico` (se usiamo il DNS pubblico). Altrimenti usiamo `nome_utente@indirizzo_ipv6`.

Per il parametro `nome_utente` assicuriamoci di specificare il nome utente appropriato per l'AMI utilizzata. Per esempio:

- *per l'AMI Amazon Linux 2 o Amazon Linux*, il nome utente è `ec2-user`;
- *per un'AMI CentOS*, il nome utente è `centos`;
- *per un'AMI Debian*, il nome utente è `admin` o `root`;
- *per un'AMI Ubuntu*, il nome utente è `ubuntu`;
- *per un'AMI RHEL*, il nome utente è `ec2-user` o `root`;
- *per un'AMI SUSE*, il nome utente è `ec2-user` o `root`;
- *per un'AMI Fedora*, il nome utente è `ec2-user` o `fedora`.

Nel caso dell'istanza usata a scopo di esempio, andremo a usare `ubuntu` e quindi avremo come *hostname* `ubuntu@public-dns`; lasciamo le altre impostazioni invariate; facciamo clic su *Connection > SSH > Auth*.

Selezioniamo il file `.ppk` creato in precedenza e facciamo clic su *Open*. Se è la prima volta che ci connettiamo a questa istanza, PuTTY visualizzerà una finestra di dialogo di avviso di sicurezza, che chiede se ci fidiamo dell'host al quale ci stiamo connettendo. Dopo aver fatto clic su *Yes*, verrà mostrata la finestra del terminale dell'istanza.

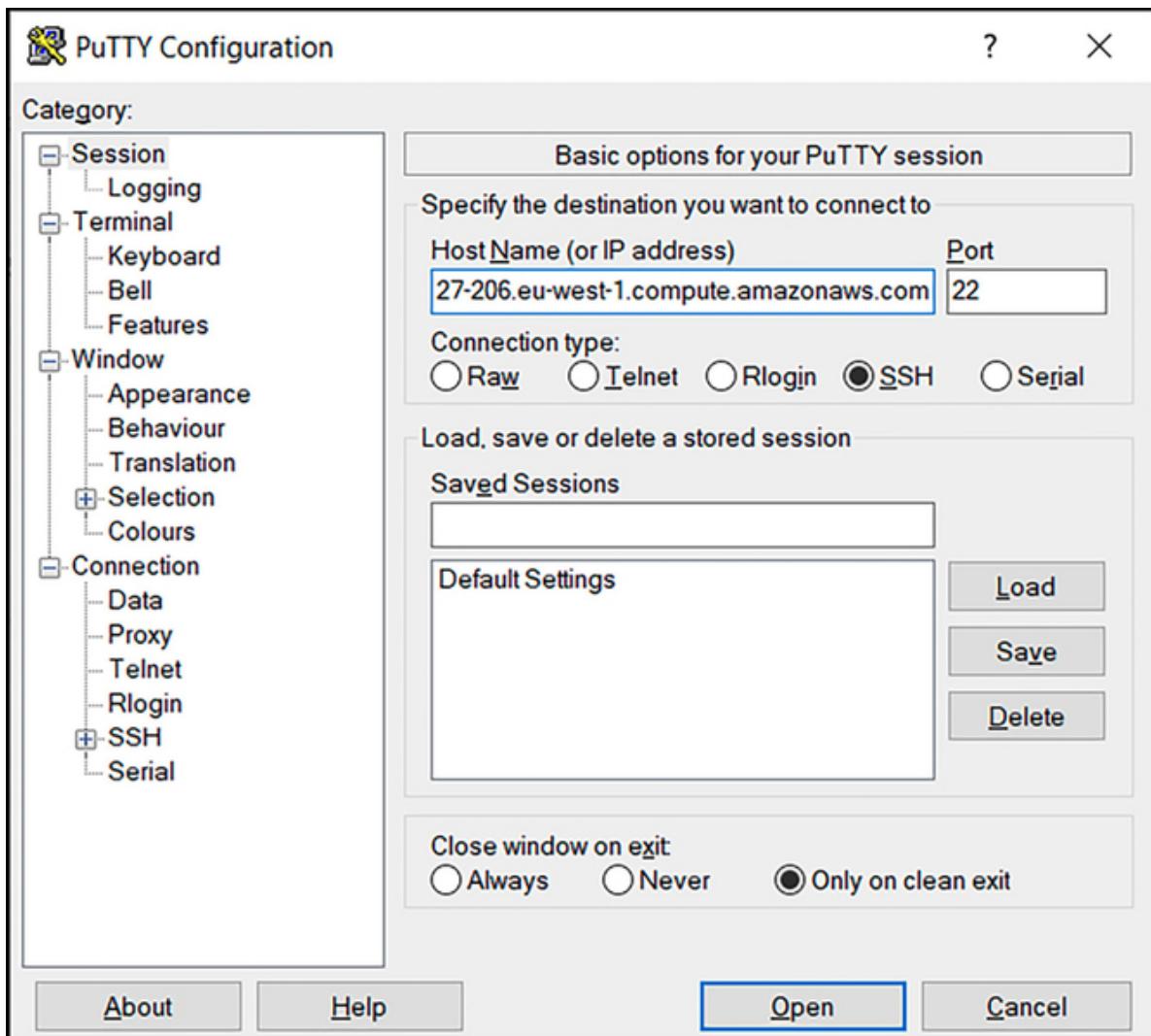


Figura 11.13 Schermata principale di PuTTY.

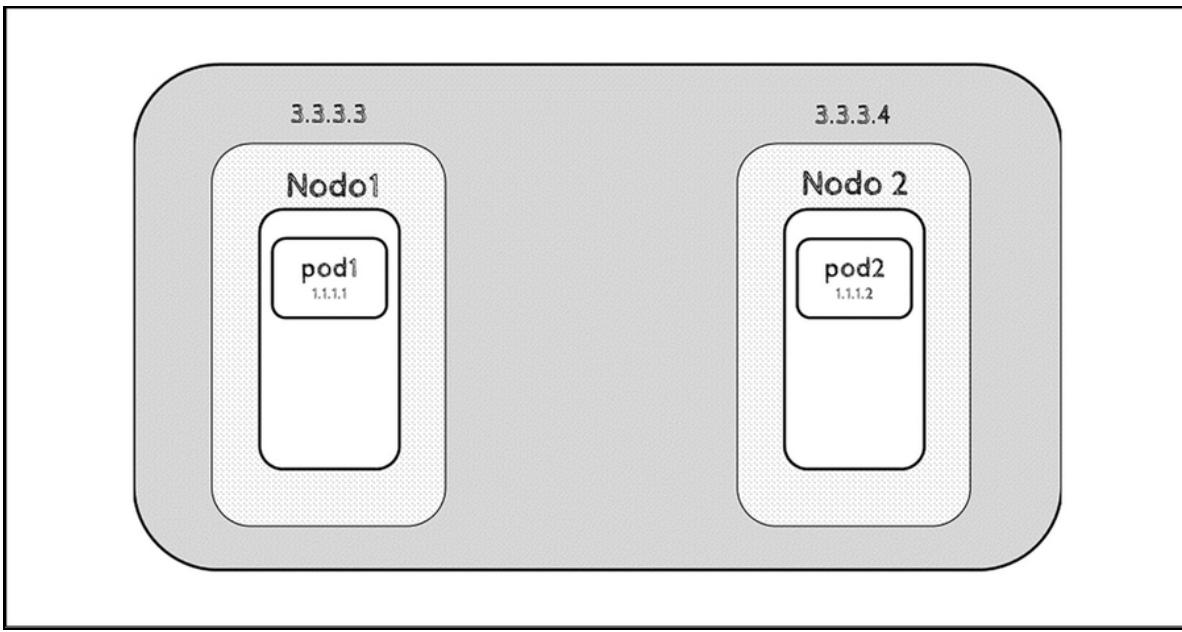


Figura 11.14 Sessione SSH dell'istanza attiva.

A questo punto, così come abbiamo fatto creando una macchina virtuale piuttosto che installando Docker in locale, possiamo sfruttare l'istanza EC2 per configurare Docker, scaricare o creare immagini ed eseguire container.

Google GCE

Google Compute Engine (abbreviato in GCE) consente di creare ed eseguire macchine virtuali sfruttando l'infrastruttura di Google. GCE offre dimensioni, prestazioni ed efficienza che consentono di avviare facilmente cluster con diverse potenze di calcolo e di grandi dimensioni senza affrontare investimenti ed eseguendo migliaia di CPU virtuali su un sistema progettato per essere veloce e offrire il massimo delle prestazioni.

Questa installazione presuppone la disponibilità di un account Google Cloud Platform. Se non disponi di un account, puoi andare su <https://console.cloud.google.com/getting-started> e crearne uno. Se invece hai un account, ma stai provando Google Cloud Platform per la prima volta, verifica bene i limiti di quota delle risorse di *Google Compute Engine* che sono gratis o inclusi nel pacchetto scelto. Per impostazione predefinita, GCP alloca un massimo di 8 CPU e nessuna GPU; è comunque possibile

modificare queste impostazioni in modo che corrispondano alle esigenze oppure richiedere più risorse. In questo caso, fai riferimento alle indicazioni riportate su <https://cloud.google.com/compute/quotas> per ulteriori informazioni, anche su come controllare le spese o come richiedere una quota aggiuntiva.

Nel browser accediamo alla console di *Google Compute Engine* all'indirizzo <https://console.cloud.google.com>, che avrà un aspetto simile a quello rappresentato nella Figura 11.15.

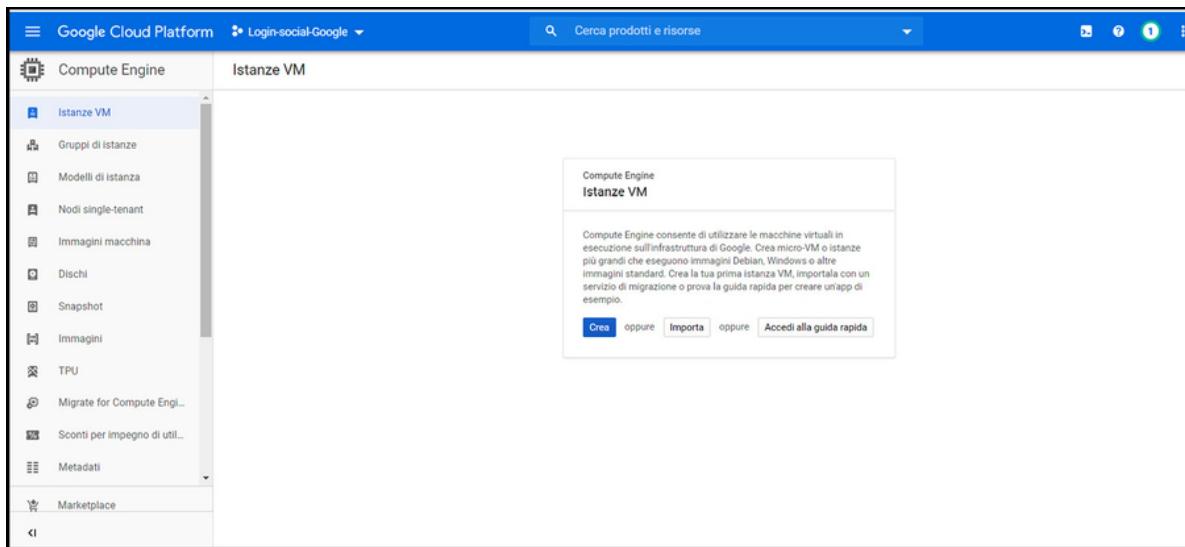


Figura 11.15 Schermata principale di GCE.

Nel pannello di navigazione a sinistra, selezioniamo *Compute Engine* > *Istanze VM*, come indicato nella Figura 11.16.

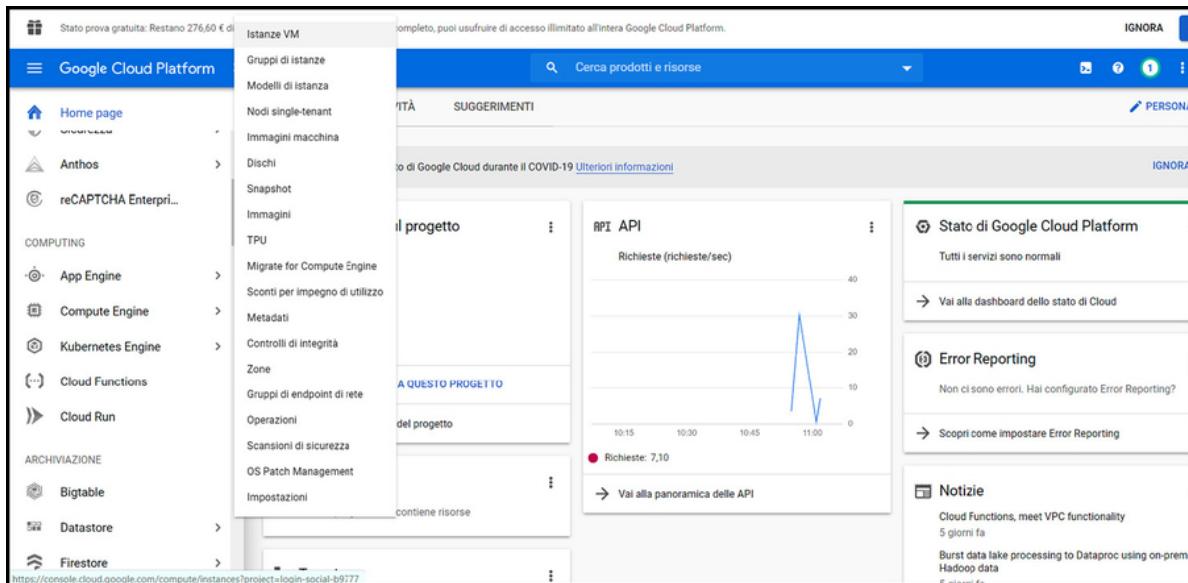


Figura 11.16 Selezione delle istanze VM.

Nella pagina delle istanze, facciamo clic su *Crea* e compiliamo le informazioni riguardanti le specifiche dell’istanza.

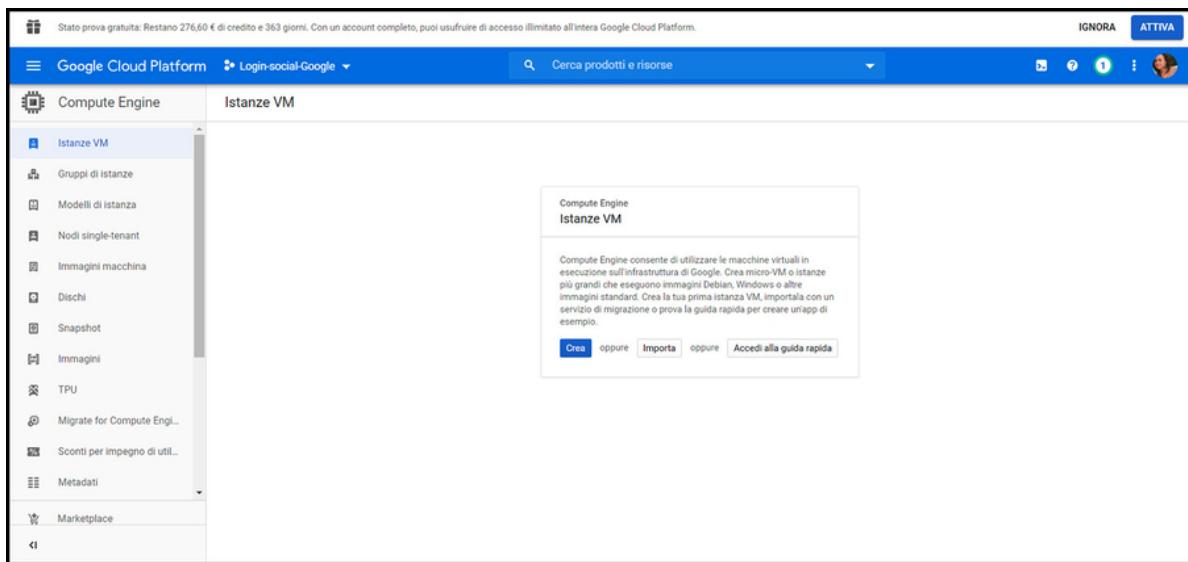


Figura 11.17 Crea istanza VM.

Occorre specificare almeno quanto segue.

- Un nome univoco e “parlante” per l’istanza.

- L'area geografica desiderata. Non tutte le zone e non tutti gli account utente possono selezionare delle aree con istanze GPU.
- Un sistema operativo che verrà utilizzato per creare l'istanza, per esempio *Ubuntu 20.04*.
- Le dimensioni del disco sono preimpostate a 10 GB di default, ma è bene cambiarle e aumentarle almeno a 50 GB.

Facciamo clic su *Crea* nella parte inferiore del modulo; quest'operazione creerà la nuova istanza della VM.

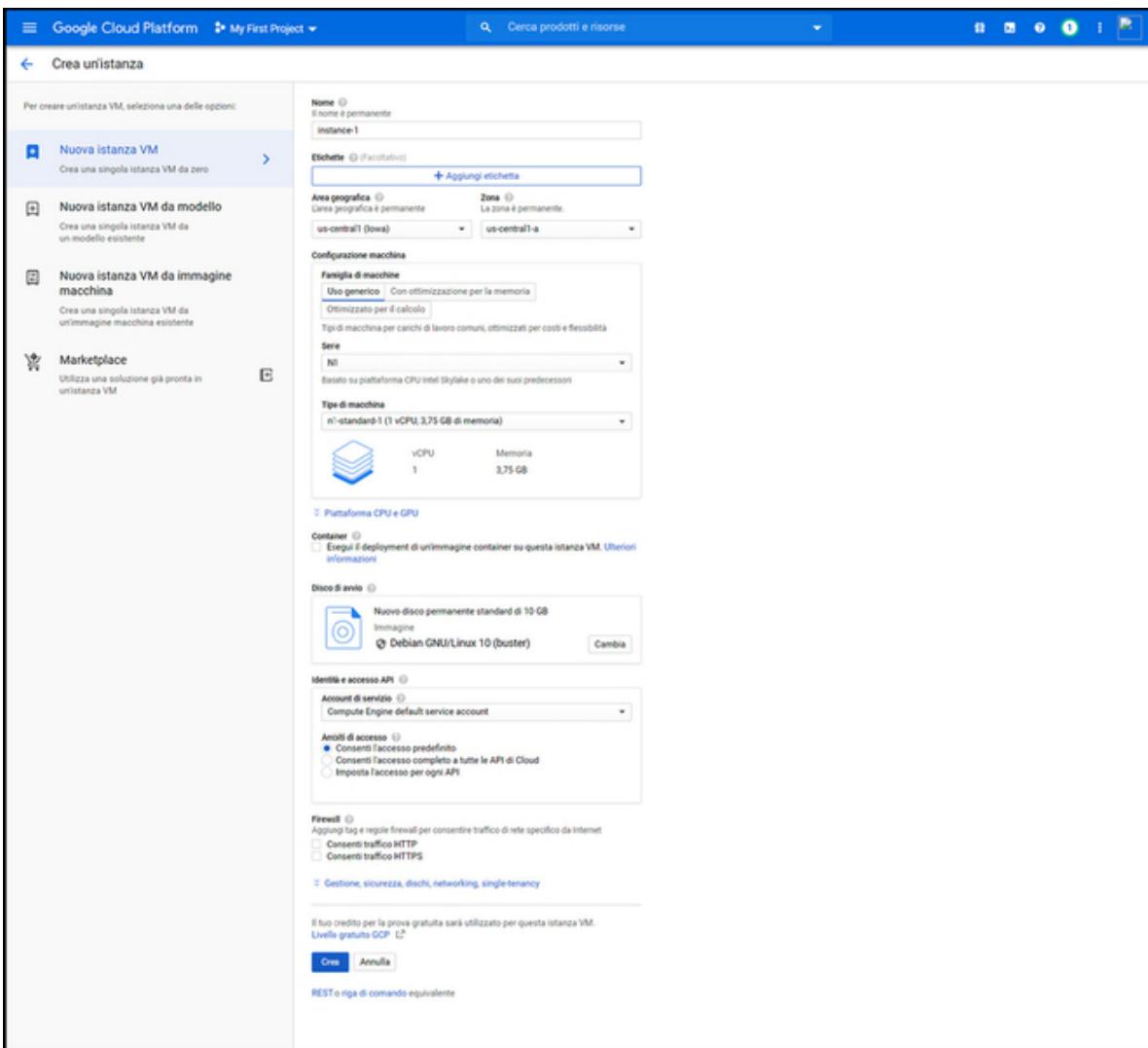


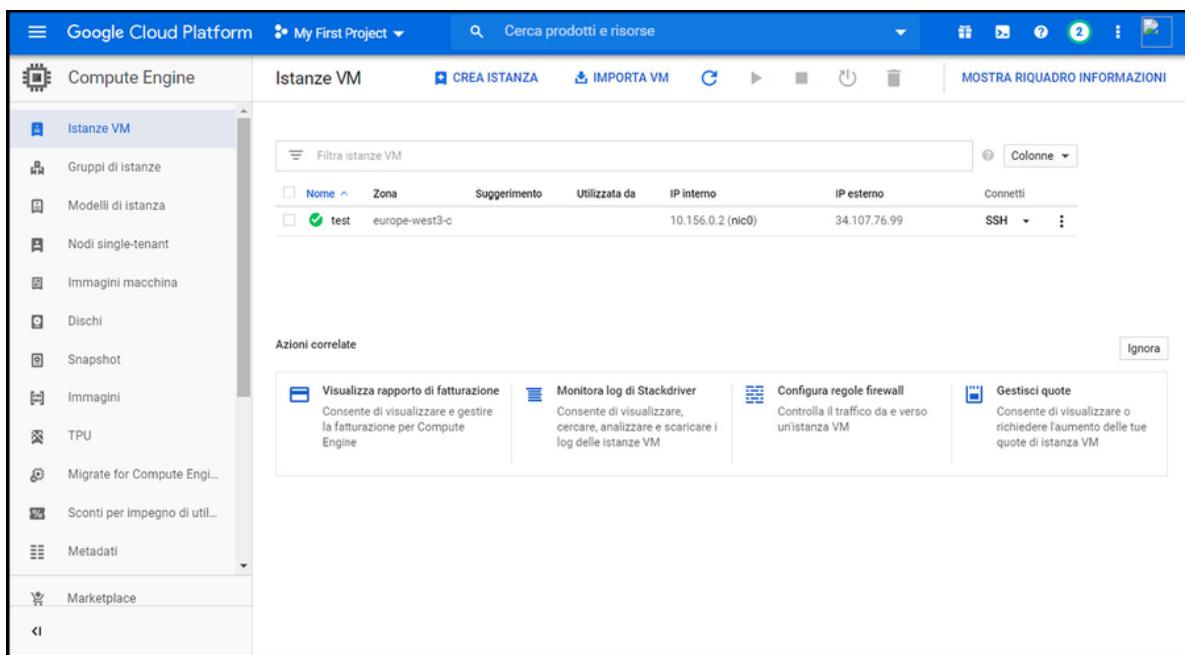
Figura 11.18 Configurazione dell'istanza della VM.

Una volta creata e avviata l'istanza, verrà visualizzato l'elenco di quelle presenti e le relative informazioni per l'accesso.

A questo punto, definiamo le regole del firewall tramite la gestione della rete VPC. Nel riquadro di navigazione a sinistra della pagina principale di *Google Cloud Platform*, selezioniamo *Rete VPC > Regole firewall* e inseriamo le seguenti impostazioni:

- specifichiamo un nome e una descrizione univoci per questa istanza;
- alla voce *Destinazioni* selezioniamo *Tutte le istanze nella rete* (a ora ne abbiamo solo una);
- alla voce *Intervallo IP di origine*, inseriamo il valore `0.0.0.0/0`.
- nella sezione *Protocolli e porte*, selezioniamo la voce *Protocolli e porte specificati* e immettiamo il valore `12345` alla voce *tcp*.

Una volta terminato, facciamo clic su *Crea* nella parte inferiore della pagina per salvare le modifiche e applicarle.



The screenshot shows the Google Cloud Platform interface for Compute Engine instances. The left sidebar lists various Compute Engine services: Istanze VM, Gruppi di istanze, Modelli di istanza, Nodi single-tenant, Immagini macchina, Dischi, Snapshot, Immagini, TPU, Migrate for Compute Engi..., Sconti per impegno di util..., Metadati, and Marketplace. The main panel is titled 'Istanze VM' and shows a table with one row. The table columns are: Nome (Nome), Zona (Zona), Suggerimento (Utilizzata da), IP interno (IP interno), IP esterno (IP esterno), and Connetti (Connetti). The row for the instance 'test' shows: Nome (test), Zona (europe-west3-c), Utilizzata da (10.156.0.2 (nic0)), IP interno (34.107.76.99), Connetti (SSH), and an 'Ignora' button. Below the table, there are four 'Azioni correlate' (Correlated Actions) cards: 'Visualizza rapporto di fatturazione' (View billing report), 'Monitora log di Stackdriver' (Monitor Stackdriver logs), 'Configura regole firewall' (Configure firewall rules), and 'Gestisci quote' (Manage quotas).

Figura 11.19 Riepilogo delle istanze presenti.

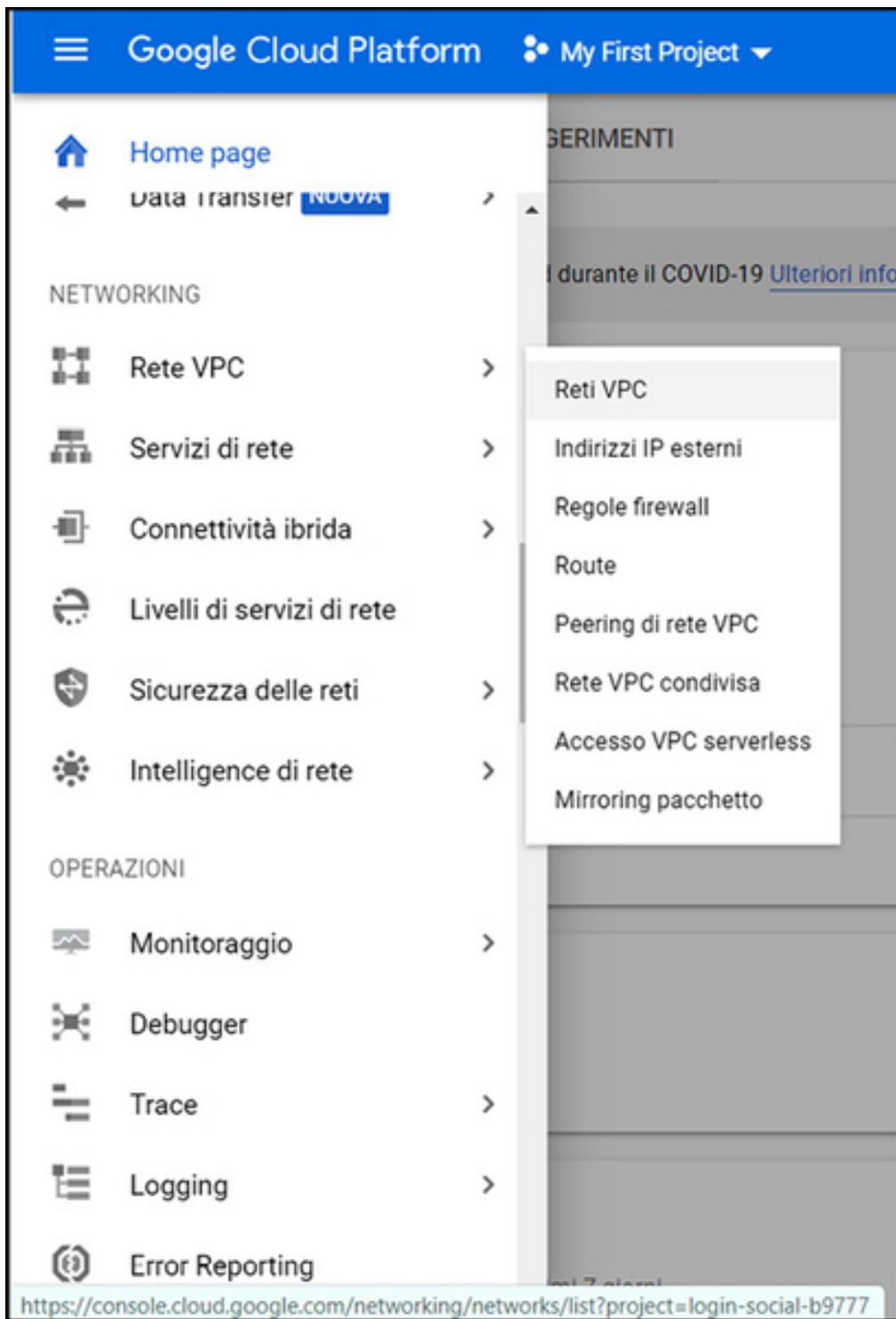


Figura 11.20 Configurazione delle reti VPC.

Torniamo nella pagina delle istanze VM e avviamo una sessione SSH per la macchina che abbiamo creato, selezionando una delle voci *Apri*

tramite il menu a tendina del pulsante *SSH*, visibile nell'elenco delle istanze.

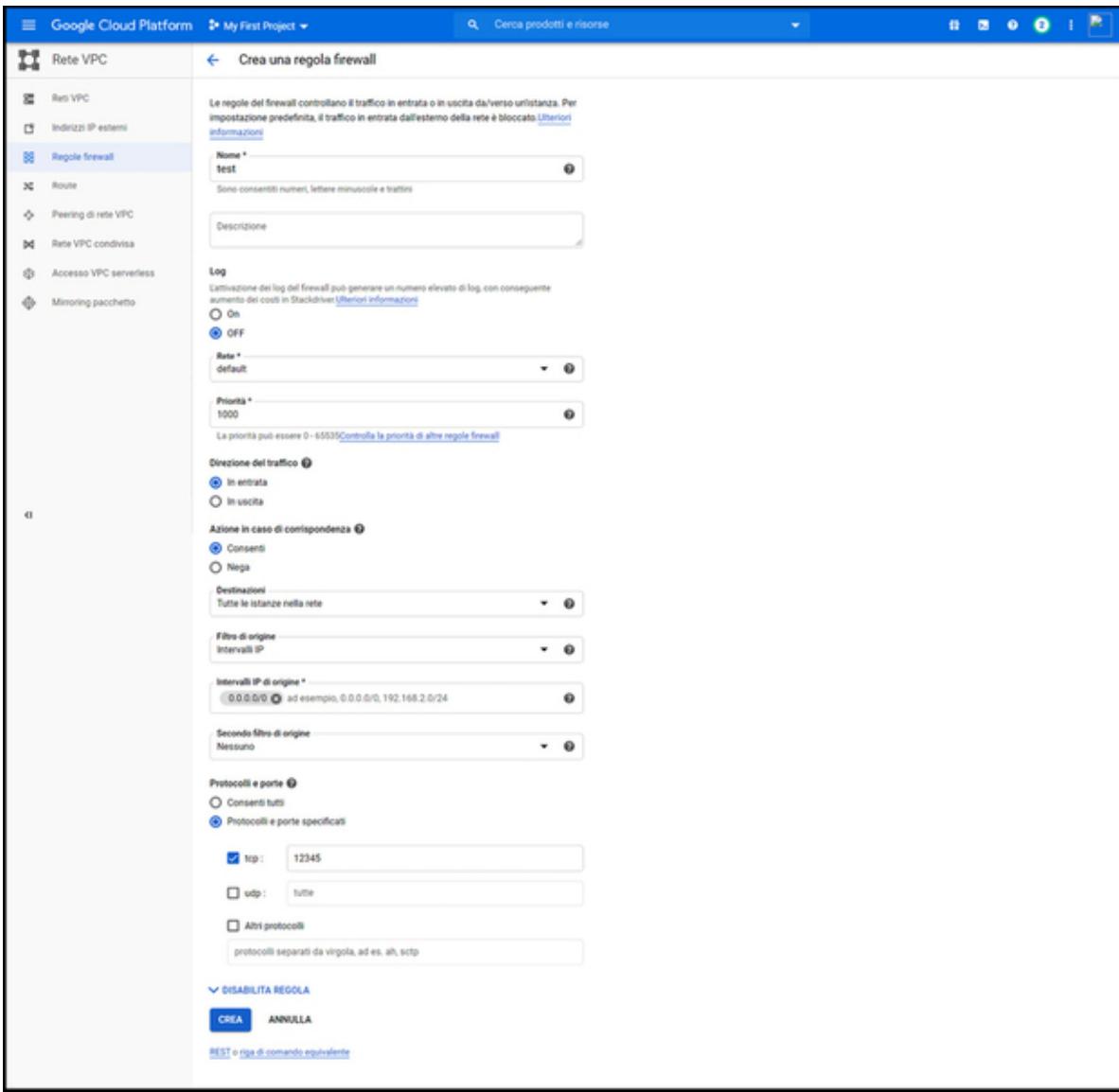


Figura 11.21 Configurazione delle regole del firewall.

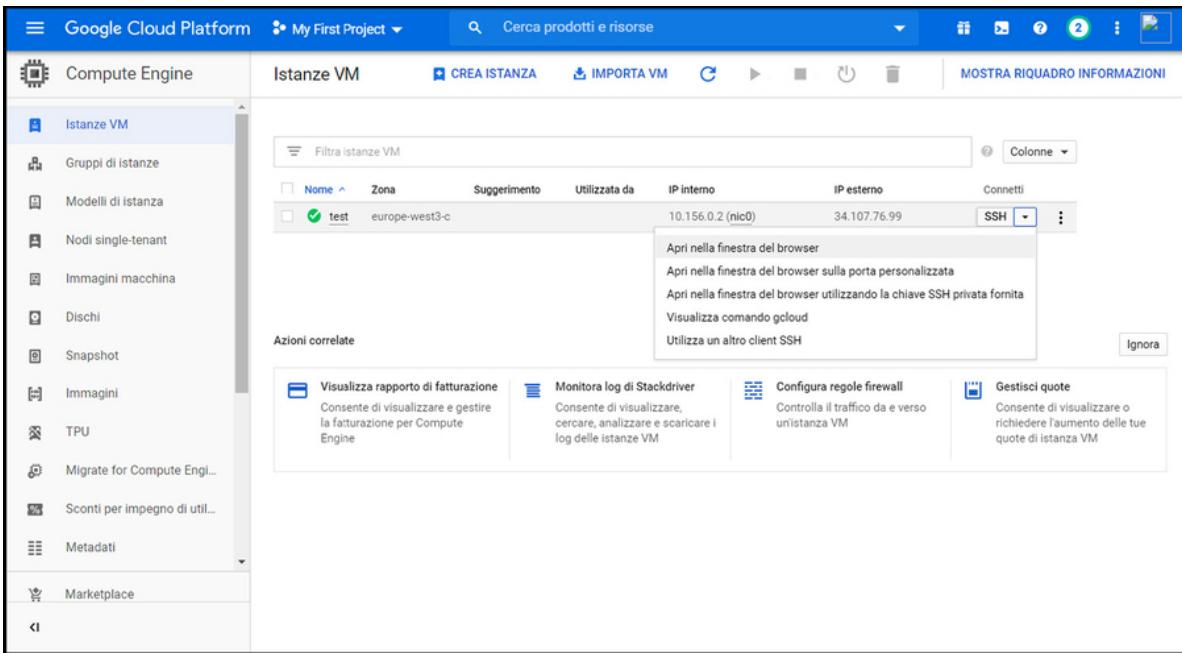


Figura 11.22 Collegamento tramite SSH all’istanza.

Selezionando *Apri nella finestra del browser*, verrà aperta una nuova finestra che fungerà da sessione SSH e da cui sarà possibile accedere alla macchina; a questo punto, potremo installare Docker Host e utilizzare la macchina come una macchina fisica, scaricando immagini e avviando container, così come abbiamo visto nei capitoli precedenti.

È possibile modificare le configurazioni della macchina in qualsiasi momento: possiamo aggiornare un’immagine Docker e le opzioni di configurazione per eseguire il container su un’istanza di VM utilizzando Google Cloud Console o lo strumento da riga di comando fornito da Google noto come `gcloud`.

Quando si aggiorna una macchina virtuale che esegue un container, Compute Engine svolge due passaggi.

- *Aggiorna le informazioni del container sull’istanza.* Compute Engine memorizza i dati aggiornati tra i metadati della macchina.
- *Arresta e riavvia l’istanza per attivare la configurazione aggiornata,* se è in esecuzione. Se l’istanza viene arrestata, aggiorna le informazioni del container e mantiene l’istanza in arresto. L’istanza della VM andrà a scaricare la nuova immagine e ad avviare il container all’avvio della VM.

Google è una scelta eccellente per ospitare degli host Docker, soprattutto se stiamo già utilizzando altri servizi Google Cloud. GCE è considerato un altro grande attore nel cloud computing e le loro API rendono molto semplice l'avvio di nuovi host sull'infrastruttura ad alta potenza di Google. È una scelta eccellente per ospitare i nostri host Docker, soprattutto se stiamo già utilizzando altri servizi Google Cloud.

Che cosa abbiamo imparato

- Come funzionano i cloud provider e quali sono quelli più utilizzati.
- Come creare degli account di accesso per Google Compute Engine e AWS.
- Come funziona il servizio EC2 di AWS e come configurare un'istanza su cui ospitare Docker, avviando inoltre una sessione in SSH per accedervi da remoto.
- Come funzionano le istanze di GCE, come crearne una che possa essere utilizzata per installare e ospitare Docker, e come creare delle regole di *firewalling* che ne gestiscano l'accesso, accedendo anche in SSH da remoto.

Docker: istruzioni

Ciclo di vita dei container

In questo paragrafo trovi i comandi più importanti relativi al ciclo di vita dei container Docker.

Creare un container (senza avviarlo)

```
docker create <IMMAGINE>
```

Esempio:

```
root@vbox:~# docker create hello-world
```

Rinominare un container esistente

```
docker rename <NOME CONTAINER> <NUOVO NOME>
```

Esempio:

```
root@vbox:~# docker rename c1 c2
```

Eseguire un comando in un nuovo container

```
docker run <IMMAGINE> <COMANDO>
```

Esempi:

```
root@vbox:~# docker run myapp
docker run --rm <IMMAGINE>
root@vbox:~# docker run -rm myapp      # rimuove il container quando questo termina
l'esecuzione
```

```
root@vbox:~# docker run -td myapp      # avvia un container e lo mantiene in esecuzione.  
root@vbox:~# docker run -it myapp
```

NOTA

Avvia un container, alloca uno pseudo-TTY collegato allo stdin del container e crea una shell bash interattiva nel container.

```
root@vbox:~# docker run -it -rm myapp
```

NOTA

Crea, avvia ed esegue un comando all'interno del container. Una volta eseguito il comando, il container viene rimosso.

Eliminare un container

```
docker rm <CONTAINER>
```

Esempio:

```
root@vbox:~# docker rm c1
```

NOTA

Questo comando può essere eseguito solo se il container non è in esecuzione, altrimenti verrà restituito un errore.

Aggiornare la configurazione di uno o più container

```
docker update <CONTAINER>
```

Esempio:

```
root@vbox:~# docker update c1
```

Avvio e arresto dei container

I seguenti comandi mostrano come avviare e arrestare i processi in un particolare container.

Avviare un container

```
docker start <CONTAINER>
```

Esempio:

```
root@vbox:~# docker start c1
```

Interrompere un container in esecuzione

```
docker stop <CONTAINER>
```

Esempio:

```
root@vbox:~# docker stop c1
```

Arrestare un container in esecuzione e riavviarlo

```
docker restart <CONTAINER>
```

Esempio:

```
root@vbox:~# docker restart c1
```

Mettere in pausa i processi in un container in esecuzione

```
docker pause <CONTAINER>
```

Esempio:

```
root@vbox:~# docker pause c1
```

Interrompere i processi in un container in esecuzione

```
docker unpause <CONTAINER>
```

Esempio:

```
root@vbox:~# docker unpause c1
```

Bloccare un container

*****ebook converter DEMO

Watermarks*****

```
docker wait <CONTAINER>
```

Esempio:

```
root@vbox:~# docker wait c1
```

NOTA

Comando viene eseguito per bloccare il container finché gli altri non si fermano (dopodiché verranno stampati i loro codici di uscita).

“Uccidere” un container

```
docker kill <CONTAINER>
```

Esempio:

```
root@vbox:~# docker kill c1
```

NOTA

Il processo viene ucciso inviando un segnale di tipo SIGKILL a un container in esecuzione.

Collegare flussi di input, output ed errore standard locali a un container in esecuzione

```
docker attach <CONTAINER>
```

Esempio:

```
root@vbox:~# docker attach c1
```

Comandi immagine Docker

Di seguito trovi tutti i comandi necessari per lavorare con le immagini Docker.

Creare un’immagine da un file Docker

```
docker build [URL]
```

Esempi:

*****ebook converter DEMO

Watermarks*****

```
root@vbox:~# docker build c1
docker build -t:[TAG] .

root@vbox:~# docker build -t myapp:latest .
```

NOTA

Quest'ultimo comando crea un'immagine partendo da un Dockerfile nella directory corrente e contrassegna l'immagine con un tag.

Pubblicare un'immagine su un repository

```
docker push <IMMAGINE>
```

Esempio:

```
root@vbox:~# docker push myapp
```

Scaricare un'immagine da un repository

```
docker pull <IMMAGINE>
```

Esempio:

```
root@vbox:~# docker pull myapp
```

Inviare un'immagine a un registro privato

```
docker push <IMMAGINE>
```

Esempio:

```
root@vbox:~# docker push 192.168.0.3:5000/myapp
```

NOTA

Vedi Capitolo 6 per configurare un registro privato.

Creare un'immagine da un tarball

```
docker import <URL/FILE>
```

Esempio:

```
root@vbox:~# docker import www.mydomain.com/image.tar.gz
```

Creare un'immagine da un container

```
docker commit <CONTAINER> <NUOVO NOME IMMAGINE>
```

Esempio:

```
root@vbox:~# docker commit c1 image2
```

Rimuovere un'immagine

```
docker rmi <IMMAGINE>
```

Esempio:

```
root@vbox:~# docker rmi image1
```

Arrestare tutti i container

```
docker stop $(docker ps -a -q)
```

Esempio:

```
root@vbox:~# docker stop $(docker ps -a -q)
```

Rimuovere tutti i container

```
docker rm $(docker ps -a -q)
```

Esempio:

```
root@vbox:~# docker rm $(docker ps -a -q)
```

Rimuovere tutte le immagini locali

```
docker rmi -f $(docker images -q)
```

Esempio:

```
root@vbox:~# docker rmi -f $(docker images -q)
```

Caricare un'immagine da un archivio tar o tramite stdin

*****ebook converter DEMO

Watermarks*****

```
docker load [TAR_FILE / STDIN_FILE]
```

Esempio:

```
root@vbox:~# docker load image.tar
```

Salvare un'immagine

```
root@vbox:~# docker save <IMMAGINE>> [TAR_FILE]
```

Esempio:

```
root@vbox:~# docker save image > image.tar.gz
```

NOTA

Salva un'immagine in un archivio tar, trasmesso in streaming su STDOUT con tutti i livelli collegati, i tag e le versioni principali.

Informazioni su container e immagini

Una volta impostati i container, dovrà sapere come ottenere tutte le informazioni importanti per gestirli. I seguenti comandi forniscono dettagli su immagini e container sul sistema.

Elencare i container in esecuzione

```
docker ps
```

Esempio:

```
root@vbox:~# docker ps
```

NOTA

Con l'opzione -a viene prodotta una lista contenente sia i container in esecuzione sia quelli che sono arrestati.

Elencare i registri da un container in esecuzione

```
docker logs <CONTAINER>
```

*****ebook converter DEMO

Watermarks*****

Esempio:

```
root@vbox:~# docker logs
```

Ispezionare

```
docker inspect <OBJECT_NAME/ID>
```

Esempio:

```
root@vbox:~# docker inspect image1
```

Elencare gli eventi in tempo reale da un container

```
docker events <CONTAINER>
```

Esempio:

```
root@vbox:~# docker events image1
```

Mostrare mappatura porta (o specifica) per un container

```
docker port <CONTAINER>
```

Esempio:

```
root@vbox:~# docker port c1
```

Processi in esecuzione in un container

```
docker top <CONTAINER>
```

Esempio:

```
root@vbox:~# docker top c1
```

Statistiche sull'utilizzo delle risorse in tempo reale dei container

*****ebook converter DEMO

Watermarks*****

```
docker stats <CONTAINER>
```

Esempio:

```
root@vbox:~# docker stats c1
```

Mostrare le modifiche ai file (o directory) su un file system

```
docker diff <CONTAINER>
```

Esempio:

```
root@vbox:~# docker diff c1
```

Elencare tutte le immagini archiviate localmente

```
docker image ls
```

Esempio:

```
root@vbox:~# docker image ls
```

Mostrare la storia di un'immagine

```
docker history <IMMAGINE>
```

Esempio:

```
root@vbox:~# docker history image1
```

Reti

Una delle funzionalità più preziose di Docker è la capacità di connettere i container tra loro e ad altri carichi di lavoro non Docker. Questo paragrafo tratta i comandi relativi alla rete.

Elencare le reti presenti

```
docker network ls
```

*****ebook converter DEMO

Watermarks*****

Esempio:

```
root@vbox:~# docker network ls
```

Rimuovere una o più reti

```
docker network rm [RETE]
```

Esempio:

```
root@vbox:~# docker network rm custom-network
```

Mostrare informazioni su una o più reti

```
docker network inspect [RETE]
```

Esempio:

```
root@vbox:~# docker network inspect custom-network
```

Connettere un container a una rete

```
docker network connect [RETE] <CONTAINER>
```

Esempio:

```
root@vbox:~# docker network connect custom-network
```

Disconnettere un container da una rete

```
root@vbox:~# docker network disconnect [NETWORK] <CONTAINER>
```

Esempio:

```
root@vbox:~# docker network disconnect custom-network
```

Dockerfile: istruzioni

ADD

Copia i file dal percorso nel file system ospite o da un dato URL all'interno dell'immagine. Se viene utilizzato un file compresso, questo verrà automaticamente espanso. Poiché la gamma di funzionalità coperta da `ADD` è piuttosto ampia, in genere è preferibile usare il comando `COPY`, di norma più semplice per copiare file e directory nel contesto di build; si preferisce inoltre l'utilizzo dell'istruzione `RUN` tramite il comando `curl` o `wget` per scaricare risorse online (soprattutto perché questo garantisce la possibilità di elaborazione ed eliminazione del file di download con la stessa istruzione).

Esempi:

```
ADD file.txt /home/          #copia file.txt nella cartella /home/ del container  
ADD file1 file2 /home/      #copia i file specificati nella cartella /home/
```

CMD

Esegue le istruzioni fornite all'avvio del container; se è stato definito un `ENTRYPOINT`, l'istruzione verrà interpretata come un argomento dell'istruzione `ENTRYPOINT`. L'istruzione `CMD` viene sovrascritta da tutti gli argomenti che Docker esegue dopo il nome dell'immagine; per questo motivo, solo l'ultima istruzione `CMD` avrà effetto e tutte le precedenti istruzioni `CMD` verranno ignorate (comprese quelle nelle immagini di base). Il funzionamento di base è analogo ai comandi `RUN` ed `ENTRYPOINT` e per questa ragione spesso vengono confusi; verifica nel dettaglio l'uso appropriato.

Esempi:

```
CMD echo "Hello world"          #stampa "Hello World"
CMD ["echo", "Hello world"]      #stampa "Hello World"
```

COPY

Utilizzato per copiare file dal file system del sistema host nell’immagine. I caratteri jolly possono essere utilizzati e permettono di specificare più file o directory. Non è possibile specificare percorsi di origine al di fuori del contesto di compilazione.

Esempi:

```
COPY file.txt /home/           #copia file.txt nella cartella /home del container
COPY file1 file2 /home/         #copia i file specificati nella cartella /home
COPY file1.py dest.py          #copia il contenuto di file1.py nel file dest.py
```

ENTRYPOINT

Imposta un eseguibile (e degli argomenti predefiniti) da eseguire all’avvio del container. Qualsiasi istruzione o argomento `CMD` da eseguire o qualsiasi parametro passato al comando `docker run` verrà passato come parametro all’eseguibile. Le istruzioni `ENTRYPOINT` sono spesso utilizzate per fornire script “di avvio” che inizializzano variabili e servizi prima di interpretare qualsiasi altra informazione. Il funzionamento di base è analogo ai comandi `RUN` e `CMD` e per questa ragione spesso vengono confusi; verifica nel dettaglio l’uso appropriato.

Esempi:

```
ENTRYPOINT ls      #elenca file e directory presenti nella cartella di lavoro
ENTRYPOINT ["/bin/echo", "Hello"]    #stampa "Hello"
ENTRYPOINT ["/bin/echo", "Hello"]    #stampa "Hello world"
CMD ["world"]
docker run -it <immagine> John      #stampa "Hello John"
```

ENV

Imposta le variabili d’ambiente all’interno dell’immagine; queste possono essere indicate in qualsiasi momento all’interno del Dockerfile, ovviamente prima dell’uso effettivo. Le variabili saranno disponibili anche all’interno dell’immagine.

Esempio:

```
ENV VERSIONE 1.3  
RUN apt install -y pacchetto=$VERSIONE      #installa il pacchetto nella versione  
1.3
```

EXPOSE

Indica a Docker che il container avrà un processo in ascolto su una o più porte specificate. Queste informazioni vengono utilizzate da Docker durante il collegamento di container o la pubblicazione di porte fornendo l'argomento `-P` all'esecuzione tramite `docker run`.

Esempio:

```
EXPOSE 8080      #il container sarà in ascolto sulla porta 8080
```

FROM

Imposta l'immagine di base per il Dockerfile; le istruzioni successive si basano su questa immagine. L'immagine di base è specificata come `<immagine>:<tag>`. Se il tag viene omesso, si presume che sia il più recente, ma è vivamente consigliabile impostare sempre il tag su una versione specifica, per evitare sorprese. *Deve essere la prima istruzione in un Dockerfile.*

Esempi:

```
FROM python:3      #l'immagine di base utilizata sarà python nella versione 3  
FROM node         #l'immagine di base sarà l'ultima versione disponibile sul  
registro
```

LABEL

Aggiunge metadati a un'immagine; utilizza il formato coppia chiave-valore. Per includere spazi all'interno di un valore `LABEL`, utilizza virgolette e backslash come faresti con la riga di comando. Normalmente viene utilizzato per impostare il nome e i dettagli di contatto del manutentore dell'immagine (sostituisce in parte l'istruzione `MAINTAINER`, ormai “deprecata”).

Esempi:

*****ebook converter DEMO

Watermarks*****

```
LABEL "nome"="pippo"      #assegna alla chiave "nome" il valore "pippo"
LABEL versione="1.0"        #assegna alla chiave "versione" il valore "1.0"
LABEL descrizione="Lorem ipsum \
dolor sit amet, consectetur adipiscing elit."      #label multiriga
```

ONBUILD

Specifica un’istruzione da eseguire in seguito, quando l’immagine viene utilizzata come livello base per un’altra immagine; può essere utile per elaborare i dati che verranno aggiunti in un’immagine figlia (per esempio, l’istruzione può copiare nel codice da una directory scelta ed eseguire uno script di compilazione sui dati). L’istruzione `ONBUILD` aggiunge all’immagine un trigger da eseguire in un secondo momento, quando l’immagine viene utilizzata come base per un’altra build. Il trigger verrà eseguito *nel contesto della build downstream*, come se fosse stato inserito immediatamente dopo l’istruzione `FROM` nel Dockerfile. Quando viene incontrata un’istruzione `ONBUILD`, si aggiunge un trigger ai metadati dell’immagine in costruzione.

Alla fine della compilazione, viene archiviato un elenco di tutti i trigger; a questo punto l’immagine può essere utilizzata come base per una nuova build, utilizzando l’istruzione `FROM`. Come parte dell’elaborazione dell’istruzione `FROM`, la costruzione avviene in modo che i trigger vengano eseguiti nello stesso ordine in cui sono stati registrati. Se uno dei trigger fallisce, l’istruzione `FROM` viene interrotta e ciò comporta il fallimento dell’intera build. Se tutti i trigger hanno esito positivo, l’istruzione `FROM` viene completata e la generazione continua come al solito.

I trigger vengono cancellati dall’immagine finale dopo essere stati eseguiti; in altre parole, non sono riportati nella costruzione delle successive build. Qualsiasi istruzione di compilazione può essere registrata come trigger.

Esempio:

```
ONBUILD ADD . /app/src
```

RUN

*****ebook converter DEMO

Watermarks*****

Esegue qualsiasi comando in un nuovo livello sopra l'immagine corrente e poi esegue il commit dei risultati. L'immagine risultante verrà utilizzata per il passaggio successivo nel Dockerfile; il comando viene utilizzato principalmente per installare un nuovo pacchetto sopra la distribuzione principale del sistema operativo. Il funzionamento di base è analogo ai comandi `ENTRYPOINT` e `CMD` e per questa ragione spesso vengono confusi; verifica nel dettaglio l'uso appropriato.

Esempi:

```
RUN apt update      #ricerca degli aggiornamenti del sistema operativo  
RUN apt install -y nginx    #installa il pacchetto nginx  
RUN [ "npm", "start" ]      #esegue il comando "npm start"
```

USER

Imposta l'utente (tramite nome o tramite UID) da utilizzare nelle successive istruzioni `RUN`, `CMD` o `ENTRYPOINT`. Nota che gli *UID* sono gli stessi tra l'host e il container, ma i nomi utente possono essere assegnati a *UID* diversi, il che può rendere le cose difficili quando si impostano le autorizzazioni.

Esempio:

```
USER john      #le istruzioni successive saranno eseguite con l'utenza john
```

VOLUME

Dichiara il file o la directory specificati come volume. Se il file (o la directory) esiste già nell'immagine, verrà copiato nel volume all'avvio del container. Non è possibile specificare la directory host per un volume all'interno di un file Docker per motivi di portabilità e sicurezza. Per maggiori dettagli, vedi il Capitolo 4.

Esempio:

```
VOLUME /volume1      #crea un volume denominato volume1
```

WORKDIR

Imposta la directory di lavoro per le successive istruzioni `RUN`, `CMD`, `ENTRYPOINT`, `ADD` oppure `COPY`. Può essere usato più volte; possono essere utilizzati percorsi relativi, ma sempre rispetto al precedente `WORKDIR`. Se la cartella specificata non esiste, verrà creata e verrà automaticamente cambiato il percorso di lavoro (questo comando può essere paragonato all'esecuzione del comando `mkdir <cartella> && cd <cartella>` di un sistema Unix-based).

Esempi:

```
WORKDIR /c1      #la directory di lavoro ora è /c1  
WORKDIR c2      #la directory di lavoro ora è /c1/c2  
WORKDIR c3      #la directory di lavoro ora è /c1/c2/c3  
WORKDIR /app    #la directory di lavoro ora è /app
```

Docker Machine: istruzioni

create

Crea una nuova macchina secondo il driver e il nome specificato.

Esempio:

```
docker-machine create --driver virtualbox test      #macchina VBox di nome test
```

ls

Elenca tutte le macchine a disposizione

Esempio:

```
docker-machine ls
```

env

Restituisce le informazioni di dettaglio di una data macchina.

Esempio:

```
docker-machine env test
```

start

Avvia una macchina con il nome specificato.

Esempio:

```
docker-machine start test      #avvia la macchina "test"
```

stop

Arresta una macchina con il nome specificato.

Esempio:

```
docker-machine stop test      #arresta la macchina "test"
```

restart

Riavvia una macchina con il nome specificato.

Esempio:

```
docker-machine restart test    #riavvia la macchina "test"
```

ip

Restituisce le informazioni sull'indirizzo IP della macchina.

Esempio:

```
docker-machine ip test       #indirizzo IP della macchina "test"
```

rm

Rimuove la macchina di cui viene specificato il nome.

Esempio:

```
docker-machine rm test       #elimina la macchina "test"
```

ssh

Avvia una sessione SSH con la macchina specificata.

Esempio:

```
docker-machine ssh test      #avvio SSH con la macchina "test"
```

Docker Compose: istruzioni

Esempio di file docker-compose.yml

```
version: '2'
services:
  # nome del servizio
  web:
    build: .
    # build a partire da un Dockerfile
    context: ./path/to/Dockerfile
    # nome del file
    dockerfile: Dockerfile
    # traffico instradato dalla porta 5000 del container sulla 5000 dell'host
    ports:
      - "5000:5000"
    volumes:
      - .:/mydata
  # nome del servizio
  redis:
    # immagine che viene scaricata dal registro ufficiale
    image: redis
```

up

Esegue la build del progetto e avvia i servizi specificati nel file `docker-compose.yml`.

Esempio:

```
$ docker-compose up
```

down

Arresta i servizi e rimuove i container, le reti, i volumi e le immagini create tramite comando `docker-compose up`. Per impostazione predefinita,

vengono rimossi i soli container che sono stati creati tramite file `docker-compose.yml`.

Esempio:

```
$ docker-compose down
```

pause

Mette in pausa i container in esecuzione di un servizio. Possono riprendere l'attività lanciando il comando `docker-compose unpause`.

Esempio:

```
$ docker-compose pause
```

stop

Arresta i servizi.

Esempio:

```
$ docker-compose stop
```

ps

Elenca tutti i container e le relative informazioni rispetto ai servizi avviati tramite il file `docker-compose.yml`.

Esempio:

```
$ docker-compose ps
```

scale

Aggiunge o rimuove container per scalare orizzontalmente l'applicazione secondo la dimensione specificata.

Esempio:

```
$ docker-compose scale test=6      #crea 6 repliche del container "test"
```

rm

Arresta ed elimina i servizi.
Esempio:

```
$ docker-compose rm
```

K8s: istruzioni

Pod

Esempio di file di configurazione per un pod (pod.yaml)

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
spec:
  containers:
  - image: busybox:1.28.4
    command:
    - sleep
    - "3600"
    name: busybox
  restartPolicy: Always
```

create

Crea l'oggetto specificato sfruttando il file formato YAML specificato.

Esempio:

```
$ kubectl create -f pod.yaml
```

get

Fornisce informazioni sui pod presenti.

Esempi:

```
$ kubectl get pods  
$ kubectl get pods pod1 -o yaml      # stampa il file .yaml del pod
```

describe

Fornisce informazioni su uno specifico pod.

Esempio:

```
$ kubectl describe pods pod1
```

Distribuzione

Esempio di file di configurazione per una distribuzione (deployment.yaml)

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: web-deployment  
  labels:  
    app: web  
spec:  
  replicas: 2  
  selector:  
    matchLabels:  
      app: web  
  template:  
    metadata:  
      labels:  
        app: web  
    spec:  
      containers:  
        - name: front-end  
          image: nginx  
          ports:  
            - containerPort: 80
```

create

Crea l'oggetto specificato sfruttando il file formato YAML specificato.

Esempio:

```
$ kubectl create -f deployment.yaml
```

run

Crea la distribuzione specificata sfruttando un'immagine.

Esempio:

```
$ kubectl run nginx --image=nginx
```

expose

Crea un servizio dalla distribuzione.

Esempio:

```
$ kubectl expose deployment nginx --port=80 --type=NodePort
$ kubectl run nginx --image=nginx
```

get

Fornisce informazioni sulle distribuzioni presenti.

Esempi:

```
$ kubectl get deployments
$ kubectl get deployments web-deployment -o yaml      # stampa il file .yaml della
distribuzione
```

describe

Fornisce informazioni su una specifica distribuzione.

Esempio:

```
$ kubectl describe deploy web-deployment
```

scale

Estende in scala il numero di repliche su una specifica distribuzione.

Esempio:

```
$ kubectl scale --replicas=4 deployments/web-deployment
```

Volume

*****ebook converter DEMO

Watermarks*****

Esempio di file di configurazione per un volume (volume.yaml)

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: data
  namespace: web
spec:
  storageClassName: local-storage
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /mnt/data
```

apply

Crea un volume persistente secondo il file in formato YAML specificato.

Esempio:

```
$ kubectl apply -f volume.yaml
```

get

Fornisce informazioni sui volumi presenti.

Esempio:

```
$ kubectl get pv
```

describe

Fornisce informazioni su uno specifico volume.

Esempio:

```
$ kubectl describe pv
```

Nodi

get

Fornisce informazioni sui nodi presenti.

Esempi:

```
$ kubectl get nodes  
$ kubectl get nodes node1 -o yaml      # stampa il file .yaml del nodo
```

describe

Fornisce informazioni su uno specifico nodo.

Esempio:

```
$ kubectl describe nodes node1
```

Introduzione

[A chi è rivolto questo manuale?](#)

Capitolo 1 - Introduzione a Docker

[Le origini di Docker](#)

[Chi utilizza Docker?](#)

[Quando non dovresti usare Docker](#)

[Architetture software](#)

[Pattern](#)

[Conclusioni](#)

Capitolo 2 - Installazione di Docker

[Installare Docker su Ubuntu/Debian](#)

[Installare Docker su Windows](#)

[Installare Docker su macOS](#)

[Installare Docker su Raspberry Pi](#)

[Installare Docker su RHEL/CentOS/Fedora](#)

[Che cosa abbiamo imparato](#)

Capitolo 3 - Concetti preliminari

[Docker vs VM](#)

[Architettura di Docker](#)

[Immagini](#)

[Dockerfile](#)

[Container](#)

[“Hello world!”](#)

[Avviare e arrestare i container](#)

[Rimozione di container e immagini](#)

[Analizzare un container](#)

[Le origini di Docker](#)

[Gestione dei processi](#)

[Che cosa abbiamo imparato](#)

Capitolo 4 - Gestire i dati

[Gestione dei volumi](#)
[Condivisione di dati tra container](#)
[Copiare i dati](#)
[Che cosa abbiamo imparato](#)

Capitolo 5 - Gestire la rete

[Controllo delle interfacce](#)
[Esporre in rete un container](#)
[Usare il bridge di default](#)
[Definire una rete custom](#)
[Creare una rete multi-host](#)
[Che cosa abbiamo imparato](#)

Capitolo 6 - Docker Hub

[Registri pubblici](#)
[Alternative disponibili](#)
[Registri privati](#)
[Creare un account](#)
[Creare un repository](#)
[Push di un'immagine](#)
[Namespacing](#)
[Creare un registro privato](#)
[Che cosa abbiamo imparato](#)

Capitolo 7 - Best practice

[Ottimizzare il Dockerfile](#)
[I dodici fattori](#)
[Scegliere l'immagine di partenza](#)
[Sviluppare con Docker](#)
[Docker API](#)
[I log di Docker](#)
[Sicurezza](#)
[Che cosa abbiamo imparato](#)

Capitolo 8 - Strumenti

[Docker Swarm](#)
[Docker Compose](#)
[Che cosa abbiamo imparato](#)

Capitolo 9 - Esempi pratici

[Applicazione Laravel, Nginx e MySQL](#)
[Applicazione Express.js con MongoDB](#)
[Applicazione Flask con SQLite](#)
[Installazione di Drupal con Docker Compose](#)
[Gestione di un sito Wordpress](#)
[Creazione di un'applicazione MEAN](#)
[Uso di Apache Kafka](#)
[Configurazione di un cluster Hadoop](#)
[Che cosa abbiamo imparato](#)

Capitolo 10 - Kubernetes

[Architettura](#)
[Componenti di base](#)
[Installazione](#)
[Creare un cluster](#)
[Avviare un pod](#)
[Modalità imperativa vs dichiarativa](#)
[Gestire un servizio](#)
[Gestione di un pod con container multipli](#)
[Utilizzo dei volumi](#)
[Che cosa abbiamo imparato](#)

Capitolo 11 - Docker su Cloud

[Accedere ai cloud provider](#)
[AWS](#)
[Google GCE](#)
[Che cosa abbiamo imparato](#)

Appendice A - Docker: istruzioni

[Ciclo di vita dei container](#)
[Avvio e arresto dei container](#)
[Comandi immagine Docker](#)
[Informazioni su container e immagini](#)
[Reti](#)

Appendice B - Dockerfile: istruzioni

[ADD](#)
[CMD](#)

COPY
ENTRYPOINT
ENV
EXPOSE
FROM
LABEL
ONBUILD
RUN
USER
VOLUME
WORKDIR

Appendice C - Docker Machine: istruzioni

create
ls
env
start
stop
restart
ip
rm
ssh

Appendice D - Docker Compose: istruzioni

Esempio di file docker-compose.yml
up
down
pause
stop
ps
scale
rm

Appendice E - K8s: istruzioni

Pod
Distribuzione
Volume
Nodi