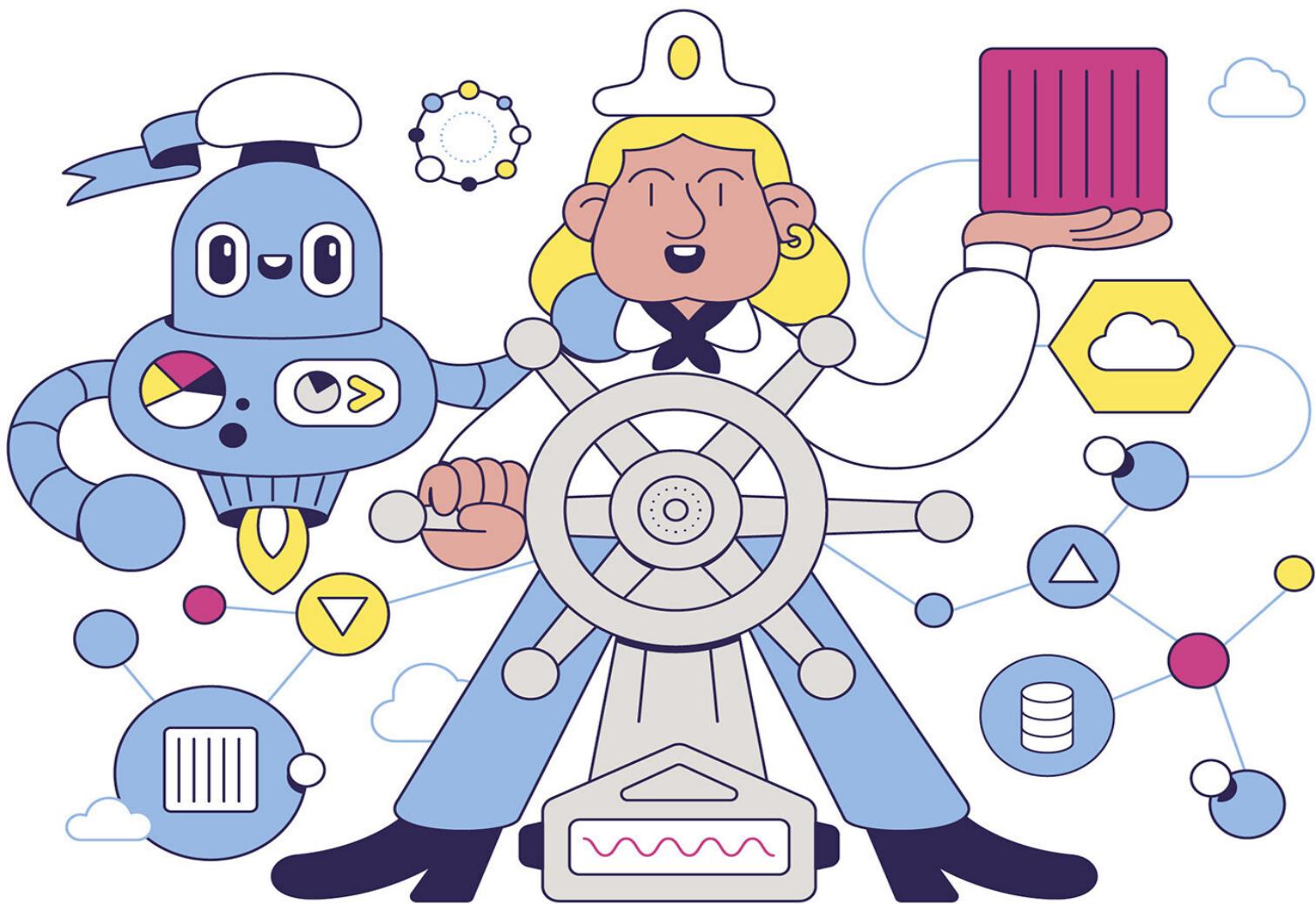


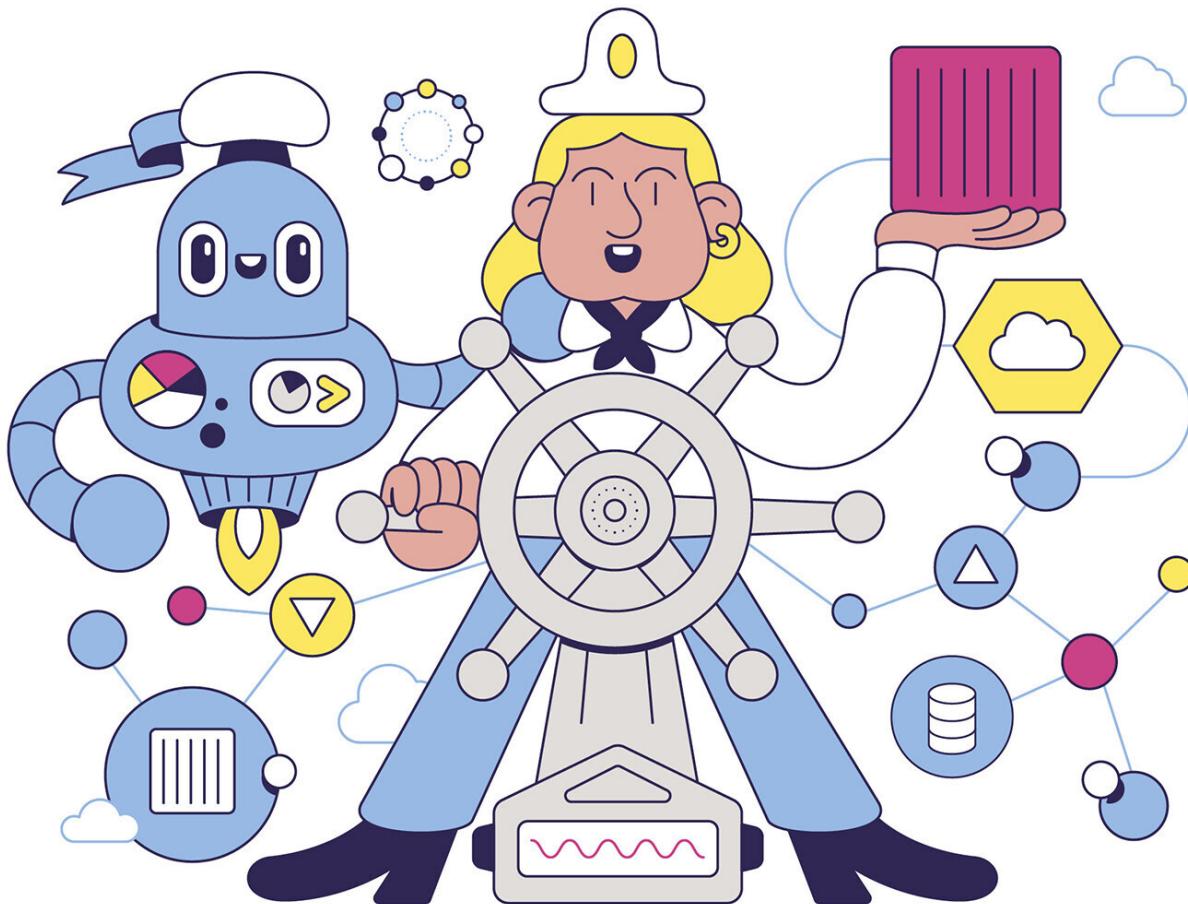
Kubernetes



**Guida per gestire
e orchestrare i container**

Licenza edgt-401-15357 ws_sonder_wHhBq0Xawbngn
SERENA SENSIO
rilasciata il 09 giugno 2023 a Giorgio Tarozzi

Kubernetes



**Guida per gestire
e orchestrare i container**

KUBERNETES
GUIDA PER GESTIRE E ORCHESTRARE I CONTAINER

Serena Sensini



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

ISBN ebook: 9788850319794

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.](#)

Se vi chiedessero di associare una sensazione a un cielo azzurro popolato di soffici nuvole bianche, che cosa rispondereste?

Leggerezza, molto probabilmente.

Ed è forse anche per questo motivo che si è deciso di adottare il termine *cloud computing* in ambito informatico: liberarsi dal peso di dover gestire l'infrastruttura fisica su cui girano le nostre applicazioni, delegando tutte le problematiche ai fornitori di servizi.

Non a caso i più critici affermano che il cloud è semplicemente il computer di qualcun altro che ci limitiamo a utilizzare senza averne alcun controllo diretto.

È indubbio che oggi questo termine faccia parte del nostro lessico quotidiano e che non possiamo più farne a meno.

Nonostante l'immagine di leggerezza da cui siamo partiti, non si può negare che dietro le quinte si celino tante tecnologie e strumenti complessi, che occorre padroneggiare se si opera in questo mondo.

E al primo posto troviamo sicuramente Kubernetes che, se da una parte sembra compiere magie facendo girare applicazioni capaci di gestire innumerevoli utenti e business milionari, dall'altra può trasformarsi in un cannone con cui provare a colpire una mosca, rappresentando più un costo che un effettivo beneficio.

Diventa necessario conoscerlo adeguatamente e comprendere quando e come utilizzarlo in maniera appropriata.

E fidatevi non è uno scherzo.

Bisogna districarsi tra una fitta rete di oggetti, termini tecnici, opzioni più o meno numerose, file di configurazione e molto altro.

Proprio in risposta a tali esigenze nasce questo manuale che, con un linguaggio rigoroso ma accessibile a tutti, guida il lettore alla scoperta del noto orchestratore di container, affrontando con gradualità tantissimi argomenti, a partire dai più semplici (pod, deployment, service, ingress ecc.) per giungere via via a quelli più ostici (volumi, RBAC, namespace, job ecc.).

L'elemento caratteristico di questa, come delle altre pubblicazioni dell'autrice Serena Sensini, è il giusto mix tra teoria e pratica: i concetti sono illustrati con dovizia di particolari in modo da comprendere che cosa si nasconde sotto al cofano, senza limitarsi a un puro utilizzo meccanico degli strumenti.

Ma allo stesso tempo non si tratta di semplice nozionismo privo di alcun valore.

Infatti, ogni argomento è corredata da una ricca serie di esempi pratici di difficoltà crescente, che portano il lettore a sporcarsi le mani per apprendere sul campo quanto trattato a livello teorico.

Il tutto in ossequio a un antico adagio che recita: "se ascolto dimentico, se vedo ricordo, se faccio capisco".

Aggiungerei: "se leggo un buon libro (come questo) e metto in pratica gli esempi, migliori le mie competenze".

A tale scopo, un intero capitolo illustra casi d'uso reali, ovvero situazioni che sicuramente si incontreranno nella propria vita lavorativa e professionale.

E ancora un'altra sezione è interamente dedicata alle buone pratiche (*best practices* per gli anglosassoni) in cui è evidente l'esperienza maturata dall'autrice lavorando su progetti concreti.

Solo confrontandosi quotidianamente con problemi reali alla ricerca delle soluzioni migliori si riesce a padroneggiare realmente una tecnologia e si imparano tanti trucchetti che non sono scritti nei libri (almeno non in tutti).

E affidarsi a chi ne sa più di noi è davvero un enorme vantaggio.

Non da ultimo, il libro fa una carrellata dei principali provider presenti sul mercato, illustrando per ciascuno di essi come essere subito operativi, anche sfruttando i piani gratuiti messi a disposizione dagli stessi.

Quindi non vi resta che intraprendere questo lungo percorso con una guida d'eccezione, Serena, che vi accompagnerà verso la meta e vi farà scoprire questo mondo affascinante.

Buona lettura e buon apprendimento!

Mauro Cicolella

Ringraziamenti

Quando questo volume sarà nelle librerie, non potrò che guardarla e ricordarmi che in ogni singola pagina c'è l'esperienza professionale maturata finora (e ancora in crescere), ma anche il s(o)upporto di tutte quelle persone che ci sono state dal giorno zero.

Gli ultimi anni sono stati densi di eventi personali e non, come una pandemia che mi ha fatto cambiare lavoro per atterrare in uno dei team più belli e forti che mi potessero capitare: un doveroso grazie va sempre a Luciano e Daniel, che sostengono ogni mia iniziativa e attività con una fiducia pressoché illimitata.

Un grazie va anche ai miei colleghi, con cui condivido gran parte delle mie giornate e che mi hanno insegnato (e faranno sicuramente ancora) tantissimo: le mie paperelle Paolo e Davide, con cui posso confrontarmi senza alcun timore, ma anche Claudio, Gabriele e Marcello, grazie ai quali ho avuto una crescita a dir poco esponenziale e che mi hanno aiutato a colmare una serie di lacune che temevo più del compito di matematica a sorpresa al liceo. Ringrazio anche Andrea per avermi aperto al mondo Azure (e non solo), ma soprattutto ringrazio Mauro, professionista e persona di inestimabile valore la cui prefazione è lusinghiera oltremodo, e che è riuscito nell'impossibile missione di leggere, e correggere, questo volume, in meno di una settimana, con una dovizia di particolari e cura senza paragoni.

Ringrazio tutte le community che mi hanno dato fiducia in questi anni e che mi hanno permesso di portare sul palco di eventi del calibro di Codemotion, KCD e PyCon il mio lavoro, così come successi e fallimenti, per rendere al mondo IT quello che ho ricevuto con estremo amore finora.

E, parlando di amore, non posso non menzionare la nascita di una nipote che mi ha stravolto la vita, di altri due nipoti in arrivo, che mi hanno reso ancora più l'orgogliosa sorella e zia che sono; inoltre, posso dire di avere la grandissima fortuna di essere circondata da una famiglia sempre pronta a farsi forza per me e a sorreggere qualunque peso, nonostante tutto, grazie a una mamma incredibile dalla forza straordinaria, a un papà che c'è sempre stato, anche con il suo essere "burbero", ai miei suoceri e a tutta la famiglia che mi ha scelto, e che mi sono scelta.

Non solo: un grazie va a tutte le persone che giorno dopo giorno mi accompagnano nelle mie follie professionali, come quella di aprire un blog a tema #tech durante la pandemia, ma anche di partecipare come speaker a diversi eventi nella stessa settimana per mettere alla prova la mia esperienza e la mia capacità di reggere lo stress.

Un grazie infinito va a chi continua a guardarmi con gli occhi pieni di amore e che, anche se non dice niente, riesce a riempirmi il cuore da più di 5 anni. Un grazie va alla vita che ho, e che verrà al mondo tra pochi mesi, e che già è circondata di tutto l'amore possibile.

Kubernetes non è solo un orchestratore. Kubernetes è una rivoluzione vera e propria nel mondo enterprise, che è permeata negli ultimi 10 anni del settore informatico in Italia attraverso un diverso approccio alle infrastrutture, al design, ma anche allo sviluppo delle applicazioni. Questa tecnologia non è soltanto lo strumento di punta di tantissime realtà grandi e piccole, ma è anche il collante di diverse community: come mi piace ricordare, dietro a ognuno di questi progetti c'è un team fatto di persone che crede fortissimamente nel proprio lavoro e che riesce, in qualche modo, a trascinare tantissime persone nella sua sperimentazione e nel suo utilizzo, fino alla completa e devota adozione. Proprio come altre tecnologie figlie di quest'epoca, Kubernetes nasce in un "piccolo" laboratorio ed esplode grazie a quelle persone che, duramente, lavorano anni per renderla un must: non a caso, dietro a molti degli strumenti odierni che sono disponibili tramite dei provider cloud c'è sempre questo strumento. Si tratta, però, di un oggetto piuttosto complesso, che richiede una conoscenza a 360 gradi di ciò che significa poter mettere un'applicazione o un servizio a disposizione di un utente finale: quando inizieremo a interagire con i primi concetti, potremo subito provare la sensazione di avere a che fare con una risorsa che non è solo per persone che sviluppano, o per persone che si occupano di amministrazione di rete, ma è molto di più. Kubernetes *guida* tutto il flusso di lavoro di un'applicazione dal suo sviluppo nel portatile personale, fino a rendere fruibile il sito Web che chiunque può raggiungere tramite il proprio browser.

In questo contesto nasce questo manuale, che vuole essere un percorso di introduzione a questa tecnologia, ma anche un modo per mettersi alla prova con dei casi d'uso reali: grazie all'esperienza maturata vestendo ognuno dei cappelli necessari alla comprensione di questa tecnologia, è stato possibile creare capitoli *ad hoc* per trattare i tanti volti dietro a Kubernetes. Un esempio è il capitolo dedicato alla sua architettura: per chi normalmente si occupa di sviluppo software, il modo in cui l'infrastruttura è stata predisposta e la configurazione su cui installeremo potrebbero essere sconosciuti e anche di scarso interesse. In realtà, per poterci astrarre dal concetto di "applicazione", dovremo aprire il cofano di questo strumento e dare un'occhiata al suo motore: ogni sezione successiva e ogni esempio sono stati curati per rendere la sua comprensione più semplice possibile, soprattutto per tutte le persone che, come una giovane me, non hanno un contesto così forte su una serie di argomenti. Questo manuale non è pensato, però, solo per chi sviluppa, ma anche per chi "costruisce": progettare un'applicazione è un'arte che parte dai dati (o da un'idea), e arriva sempre molto velocemente negli ambienti di produzione, quindi di tempo per testarla non se ne ha mai abbastanza. Avere a disposizione una serie di casi d'uso reali, insieme a *best practices* che ci permettano di affinare le tecniche che utilizziamo, ci renderà il lavoro sempre più agevole. Questo manuale, infatti, vuole essere un timone sicuro per il vostro viaggio all'interno di Kubernetes: che possa guidarvi con il vento in poppa, ma anche durante la tempesta e nella marea, facendovi sentire sempre forti delle vostre competenze.

Struttura

Ecco perché questo volume è stato così pensato: dopo una breve introduzione sulla nascita di Kubernetes e sul perché questo strumento abbia fatto da pilota per una serie di progetti successivi (Capitolo 1 e 2), vediamo con gli occhi di un *architect* quali sono i suoi “mattoncini” di base e come installare un ambiente che ci permetta di sporcarci le mani (Capitolo 3 e 4). Cambiamo cappello, e torniamo nel mondo dello sviluppo, analizzando quali sono le risorse che abbiamo a disposizione per poter avviare un’applicazione (Capitolo 5), parametrizzare la sua configurazione (Capitolo 6) e anche renderla disponibile per chi al cluster non sempre vi accede direttamente (Capitolo 7). All’interno del dominio Kubernetes avremo sicuramente bisogno della persistenza, attraverso volumi (Capitolo 8) o attraverso risorse con funzionalità specifiche (Capitolo 9). Torniamo poi nei panni di architect ed esploriamo il mondo degli utenti e dei ruoli che ci permettono di rendere più sicuro il nostro cluster (Capitolo 10), anche attraverso strumenti utili a “isolare” gli ambienti e le risorse, nonché a strumenti che ci permettono di adattare il cluster al carico di lavoro e agli imprevisti (Capitolo 14). Kubernetes è stato il motore di una serie di progetti strettamente dipendenti, come Helm e Kustomize, che a oggi sono integrati completamente in questa tecnologia e che vale la pena analizzare con casi d’uso reali (Capitolo 11); questo strumento lascia che le persone che lo utilizzano estendano le sue funzionalità di base con risorse create *ad hoc* dai suoi utenti, che poi possono essere distribuite liberamente o non (Capitolo 12). In ultimo, ma non per importanza, il Capitolo 13 è dedicato all’utilizzo di Kubernetes nel contesto cloud: per avere un’occhio sempre verso il futuro, analizziamo quali sono le diverse opzioni presenti tramite i maggiori *vendor*, insieme a una “chicca” tutta italiana.

Le appendici, infine, sono studiate per essere uno strumento utile a completare la visione di insieme sulla community e gli eventi legati al settore, sulle certificazioni, ma anche per avere a disposizione un prontuario di comandi che può tornare utile in un momento di difficoltà.

A chi è rivolto questo libro

Come anticipato, questo manuale è dedicato a chi vuole approcciarsi a Kubernetes e lavora nello sviluppo del software, a chi si è sempre occupato di compiti associati alla professione di “sistemista”, ma anche a chi viene dal mondo di Docker e vuole capire come questa tecnologia ne inglobi le infinite potenzialità e le porti in mari ancor più sicuri. Tutti gli esempi che vengono fatti sono dettagliati e sempre indipendenti dalla singola tecnologia: se presente un esempio per un’applicazione scritta in Node.js, le stesse considerazioni varranno anche se questa utilizza un linguaggio di programmazione diverso, o anche una porta differente.

Inoltre, tutto il codice presente è riportato sul repository relativo a questo manuale e disponibile sul sito <https://github.com/serenasensini/Kubernetes>, di modo che sia più semplice riuscire a rispettare la formattazione degli esempi o che sia anche possibile recuperare parte del codice che, per brevità, viene riportato in maniera parziale. Potrebbe darsi che alcuni degli esempi riportino un’indentazione non precisa: fare affidamento al codice presente online!

Terminologia

Così come avviene di frequente in questo settore, la maggior parte dei termini con cui avremo a che fare sono in lingua inglese e non facilmente traducibili. In ogni caso, ove possibile, si è preferito l’uso della lingua italiana anche in caso di termini inglezzati, per favorire chi non vi sia avvezzo: un esempio è la parola “rilascio” al posto di *deployment*, o anche il termine “programmazione” al posto di *schedule*. Il motivo di questa scelta è sì riconoscere l’utilizzo ormai dominante della forma inglese in questo contesto, ma anche far sì che la lettura sia comprensibile a chi è alle prime armi, con la

possibilità di accedere sempre alla documentazione relativa a Kubernetes cercando maggiori informazioni tramite i consueti motori di ricerca o nei siti dedicati.

Errori e feedback

Potete segnalare errori, frasi non chiare o altri riferimenti scrivendo nella sezione *Issues* del repository pubblico indicato in precedenza. Per altre informazioni, è possibile contattare direttamente l'indirizzo info@theredcode.it.

Primi passi

Il perfezionismo ci impedisce di fare doppi passi nella nostra carriera. Pensiamo di dover essere perfetti, ma non è così.

– Reshma Saujani, avvocato e fondatrice di Girls Who Code

Con l'avvento dei container Docker, questa tecnologia “dimenticata” appartenente al mondo Linux torna in auge e si fa spazio in un mondo dove la portabilità è la chiave del successo di ogni applicazione: eliminare dal proprio frasario “sul mio PC funziona”, per poter affermare con certezza che l'applicazione funzionerà su qualsiasi ambiente, indipendentemente dall'infrastruttura o dal sistema operativo, è stata una vera e propria rivoluzione nel settore IT.

Con il passare degli anni, è chiaro che le esigenze sono cambiate e che i limiti appartenenti al prodotto della Docker Inc. sono emersi piuttosto velocemente: in un ambiente cosiddetto di “produzione”, ossia l'ambiente utilizzato direttamente dagli utenti finali, è necessario prevedere dei meccanismi di alta disponibilità, piuttosto che di recupero in caso di problematiche all'hardware o al software installato.

Senza però fare ulteriori “spoiler” sui contenuti di questo e dei prossimi capitoli, cominciamo dalle basi: questa parte del manuale sarà infatti dedicata alla comprensione del perché Docker (o altri strumenti di containerizzazione) non è sufficiente per i nostri scopi e come mai tecnologie come Kubernetes si sono fatte largo nel settore grazie alle numerose features introdotte fin dai primi rilasci.

SUGGERIMENTO

Questo capitolo affronterà solo una piccola parte dei concetti di base relativi al mondo dei container: la maggior parte della terminologia specifica è stata ampiamente discussa nel volume intitolato *Docker: Sviluppare e rilasciare software tramite container*, edito sempre da Apogeo. Se si parte da zero, si consiglia quindi di dare prima un'occhiata a questo manuale, per arrivare più preparati/e ai successivi capitoli!

Dal container all'orchestratore

I *container* nascono come tecnologia che consentiva di impacchettare e isolare le applicazioni con il loro intero ambiente di *runtime*, compresi tutti i file necessari per l'esecuzione. Ciò semplificava notevolmente lo spostamento dell'applicazione tra differenti ambienti durante le varie fasi di realizzazione del prodotto (per esempio sviluppo, test, produzione e così via) pur mantenendo la piena funzionalità. Inoltre, i container nascono come parte fondamentale della sicurezza: adottando le opportune *best practice*, quali l'utilizzo di tali oggetti come ambiente di esecuzione, si ottiene una forma di segregazione per definizione e ciò garantisce che la tua applicazione sia affidabile, scalabile e sicura.

Tuttavia i container non nascono con Docker, ma fanno parte del mondo Linux®, in quanto rappresentano un insieme di uno o più processi isolati dal resto del sistema. Tutti i file necessari per metterli in esecuzione vengono forniti da un'immagine distinta, il che significa che i *container Linux* sono portatili e coerenti mentre passano dallo sviluppo, al test e infine alla produzione. Ciò li rende molto più veloci da utilizzare rispetto a un flusso di lavoro che si basa sulla replica di ambienti di test tradizionali.

Immagina di sviluppare un'applicazione: lavori su un laptop e il tuo ambiente ha una configurazione specifica, con file `.properties`, librerie e dipendenze, ma anche programmi che hai dovuto installare per far funzionare tutto l'ambiente. Altre persone che sviluppano con te potrebbero avere configurazioni leggermente diverse: una differente versione della macchina virtuale utilizzata per interpretare e compilare il linguaggio di programmazione adottato oppure errori o sviste nei file di configurazione e così via. L'applicazione che stai sviluppando si basa però su tale configurazione e dipende da librerie, dipendenze e file specifici. Allo stesso tempo, sai che all'interno della tua azienda ci sono degli ambienti di sviluppo e produzione standardizzati, ognuno con le proprie configurazioni e relativi file, per cui il tuo ambiente dovrà emulare quegli ambienti il più possibile in locale, anche se ci saranno delle impostazioni da modificare opportunamente. Inoltre, è chiaro che sul tuo PC sarà sufficiente eseguire l'applicazione con una sola istanza e testarla, magari tramite un browser, ma sugli ambienti utilizzati dagli utenti finali bisognerà accertarsi che non ci sia alcun disservizio, anche in caso di errore applicativo.

Come fai a far funzionare la tua applicazione in questi ambienti, a garantirne un certo livello di qualità tramite un flusso di CI/CD eseguito attraverso strumenti differenti, ma soprattutto a distribuire la tua app senza sbattere la testa su configurazioni e problemi vari da risolvere? La risposta: tramite i *container*.

Il container che ospiterà la tua applicazione conterrà le librerie, le dipendenze e i file necessari al funzionamento della stessa, così da poterla spostare in produzione senza spiacevoli effetti collaterali. In effetti, il contenuto di un'immagine, creata utilizzando uno strumento open source come Buildah o Podman, può essere considerato come l'installazione *ex novo* di una qualsiasi distribuzione Linux con l'aggiunta dell'applicazione sviluppata. In quanto corredata di pacchetti RPM, file di configurazione, ecc., rende tutta l'operazione molto più semplice rispetto all'installazione di nuove copie dei sistemi operativi. Crisi scongiurata: tutti sono felici.

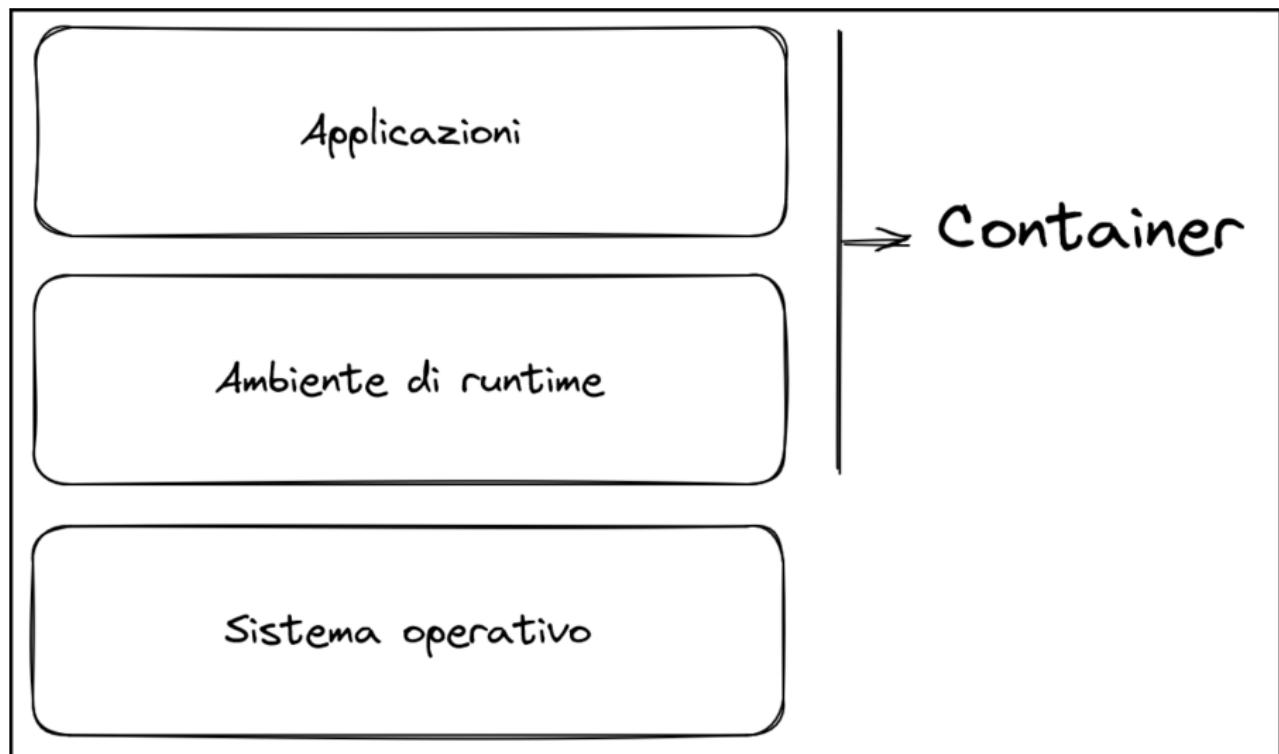


Figura 1.1 Le responsabilità di un container nel contesto di rilascio di un'applicazione.

Questo è un esempio abbastanza comune, ma il concetto di container può essere applicato a molti problemi diversi in cui sono necessarie portabilità, configurabilità e isolamento. Lo scopo dei *container Linux* (quindi già prima di Docker) è sempre stato quello di favorire un più rapido sviluppo da parte di chi si occupa di implementare le funzionalità e di soddisfare le esigenze aziendali man mano che si presentano, senza però lasciare che questo compito “consumi” del tempo necessario ad altre attività. Non solo: il termine “sicurezza” è venuto fuori diverse volte. Perché è così importante e perché dovremmo preoccuparcene? La sicurezza dei container comporta la definizione e l'adesione a procedure di creazione, distribuzione e runtime che proteggono un container, e riguarda anche le applicazioni che supportano all'infrastruttura su cui si basano. La sicurezza dei container deve essere un processo integrato e continuo a supporto della sicurezza complessiva di un'azienda. In generale non si tratta soltanto di garantire la protezione della sola applicazione, ma anche di tutto il flusso di lavoro e degli ambienti che costituiscono l'infrastruttura che serve a distribuire le applicazioni.

I container hanno quindi rappresentato uno strumento rivoluzionario per il mondo dello sviluppo e del rilascio del software, ma hanno richiesto molto lavoro e un'evoluzione significativa verso una gestione centralizzata e semplificata del loro ciclo di vita: è qui che strumenti come Kubernetes entrano in gioco e ci permettono di avere una cassetta degli attrezzi pronta all'uso estremamente potente per poter lavorare con continuità e in maniera efficiente nella fase successiva allo sviluppo dell'applicazione. Prima però di parlare dell'orchestrazione, è importante soffermarsi sull'aspetto della sicurezza di questi oggetti: non solo in Kubernetes, ma anche in tecnologie che sfruttano Kubernetes dietro le quinte; questo fattore è cruciale e rappresenta, per i clienti che adottano queste soluzioni, un requisito fondamentale per il rilascio dei propri prodotti. Vediamo quindi come integrare la sicurezza nel contesto dei container, ancor prima di “atterrare” nel mondo dell'orchestrazione, e quindi di Kubernetes.

Integrare la sicurezza

Non è un mistero che negli ultimi anni il termine *hacker* sia entrato nella quotidianità: solo nel 2022 sono stati riportati 236 milioni di attacchi causati da *ransomware* con conseguenze più o meno gravi. Questo ha fatto sì che per la maggior parte delle aziende la sicurezza informatica diventasse una priorità da non relegare in fondo alla catena di rilascio del software, ma al contrario da porre in cima alla lista. In tale contesto, come si integra il mondo dei container? I container sono composti da diversi strati di immagini e sono ormai uno standard diffuso per il rilascio delle applicazioni, sia che si parli di un ambiente “tradizionale” (on-premise in gergo) che di un ambiente cloud. Docker ha segnato un punto su una tecnologia che ha di fatto rivoluzionato il deploy, e ha rappresentato l'inizio della diffusione di altri motori come Buildah o Podman per la definizione e messa in esecuzione delle immagini. Le immagini hanno quindi creato uno standard per tutte le persone che sviluppano, lasciando un vuoto non trascurabile: come gestire flussi di lavoro che esulano da questi standard? Chiariamo questo aspetto: non sempre andremo a lavorare su un ambiente “chiuso” e (quasi) sempre il lavoro che facciamo non è pura e semplice sperimentazione, ma un qualcosa che verrà messo a disposizione degli utenti finali. Questo vuol dire *esporre* l'applicazione verso l'esterno e di conseguenza a potenziali rischi. Una riflessione importante riguarda l'introduzione della sicurezza a monte di questo flusso, quindi ancor prima che l'applicazione sia a disposizione degli utenti finali: questo ci permette di non dover *rimediare*, ma di *prevenire* rispetto al perimetro di vulnerabilità che ci troviamo a dover difendere.

Non a caso, si è passati da un approccio DevOps, dove sviluppo e rilascio (semplificando) sono ben integrati e connessi insieme, al movimento DevSecOps: il lavoro congiunto tra il team che sviluppa l'applicazione, quello che la rilascia e quello che si occupa della sicurezza è necessario e fortemente voluto dalle aziende che vedono nella gestione della sicurezza non solo una mitigazione di problemi futuri, ma un investimento redditizio. Ormai le immagini distribuite tramite container sono divenute il

formato standard di rilascio delle applicazioni negli ambienti nativi del cloud, ma anche in quelli basati su un approccio ibrido. Come indicano molte aziende leader nel settore, tra cui RedHat, bisogna prestare particolare attenzione alle immagini di base utilizzate: ai fini della sicurezza, questa è la fase più delicata perché questa immagine (in gergo tecnico è chiamata anche *base image* o *golden image*) viene utilizzata come punto di partenza da cui si creano tutte le immagini che ne derivano. Questo vuol dire che ogni pacchetto installato e ogni strumento utilizzato comporta un incremento della superficie esposta a potenziali vulnerabilità; per questo, la ricerca dovrebbe iniziare da fonti attendibili per le immagini di base: verifica sempre che l'immagine provenga da un'azienda nota o da un gruppo open source, sia ospitata su un registry affidabile e che il codice sorgente di tutti i componenti dell'immagine sia disponibile per poter verificare le risorse utilizzate. Tuttavia la sicurezza non riguarda solo questo aspetto: è infatti fondamentale giocare di anticipo e correggere le vulnerabilità, tenendo conto degli aggiornamenti forniti sia da chi distribuisce le immagini, sia dagli stessi sistemi operativi utilizzati; inoltre, è buona prassi utilizzare dei registry privati per le immagini che contengono informazioni sensibili o applicativi proprietari, che poi possono essere distribuite gestendo l'accesso al repository con delle utenze *ad hoc*.

Inoltre, integrare test di sicurezza e meccanismi di automazione per la loro distribuzione rende il lavoro più snello e sposa anche quanto descritto all'interno della filosofia DevOps; ogni manualismo deve lasciare il posto a un lavoro di sviluppo o di miglioramento delle funzionalità già presenti, e non deve togliere tempo ad attività ripetitive che possono essere trattate in maniera automatica. Ultima cosa, ma non meno importante, quando abbiamo a che fare con dei container e con tutte le risorse che ne derivano e di cui questo manuale offre un'ampia descrizione, è fondamentale avere una visione a 360 gradi: non basta utilizzare delle fonti attendibili per le proprie immagini, ma occorre proteggere in maniera opportuna tutta l'infrastruttura. Infatti, il sistema operativo che ospiterà il cluster viene abilitato utilizzando uno strumento di esecuzione dei container, idealmente gestito tramite un sistema di orchestrazione. Per rendere resiliente la piattaforma, soprattutto per ambienti complessi come quelli di produzione dedicati agli utenti finali, è fondamentale seguire alcune *best practice*, tra cui prevedere dei meccanismi di autenticazione e autorizzazione, isolamento dei progetti, e così via.

Da quanto descritto finora, sono emerse delle esigenze importanti, che Docker non sempre riesce a soddisfare: sappiamo che l'adozione di un pattern architettonale a microservizi implica la necessità di lavorare con più di un container, e quindi di monitorare più "ambienti". Avere uno strumento che ne possa centralizzare la gestione e ci fornisca un aiuto quando (e se) ci saranno errori, diventa fondamentale. In più, possiamo anche affermare che difficilmente ci capiterà di avere a che fare con una sola applicazione contenuta in un container che non abbia bisogno di comunicare con altri servizi e che quindi richieda una configurazione più complessa: non parliamo solo del modo in cui l'applicazione viene eseguita, ma anche di come si coordina con altri componenti applicativi. E se poi il nostro volume di traffico o di utenti crescesse? Come possiamo rendere l'applicazione adatta a gestire un flusso di lavoro che cambia nel tempo? L'orchestrazione rappresenta la configurazione, la gestione e il coordinamento automatizzato di sistemi informatici, applicazioni e servizi. L'orchestrazione, nel senso più generico del termine, aiuta l'IT a gestire più facilmente attività e flussi di lavoro complessi. I team IT devono gestire molti server e applicazioni, ma farlo manualmente non è una strategia scalabile: più complesso è un sistema informatico, più complessa può diventare la gestione di tutti gli attori. In questo senso, automazione e orchestrazione sono concetti diversi, ma correlati. L'automazione aiuta a rendere più efficiente la tua azienda, riducendo o sostituendo l'interazione umana con i sistemi IT e utilizzando invece il software per eseguire attività al fine di ridurre costi, complessità ed errori. In generale l'automazione si riferisce a una singola attività e questo è ben diverso dall'orchestrazione, che è il modo in cui è possibile automatizzare un processo o un flusso di lavoro che prevede molti passaggi su più sistemi disparati. L'orchestrazione in ambito IT aiuta anche a semplificare e ottimizzare i processi e

i flussi di lavoro che si verificano di frequente, soprattutto se supportano un approccio DevOps, con lo scopo di aiutare il team a distribuire le applicazioni più rapidamente.

Come si incassa Kubernetes in tutto ciò? In questo caso, parliamo di orchestrazione delle applicazioni o anche dei container, ossia quando si integrano insieme due o più applicazioni software distribuite tramite container e nel cui contesto è necessario avere dei meccanismi di automazione per la loro gestione complessiva. Questo processo consente di gestire e monitorare in maniera centralizzata le tue integrazioni e di aggiungere funzionalità per il routing delle applicazioni, per la sicurezza e per l'integrità delle stesse su cui Kubernetes è focalizzato. Ma che cos'è esattamente l'orchestrazione dei container? Tra le funzionalità che ci mette a disposizione uno strumento come questo c'è la pianificazione della distribuzione dei container nel cluster, nonché la gestione del ciclo di vita del container in base alle specifiche stabilite nella loro definizione. Ma perché abbiamo bisogno dell'orchestrazione dei container? E qual è lo scopo dell'automazione e dell'orchestrazione? Ebbene, queste funzionalità consentono di ridimensionare le applicazioni con un singolo comando, creare rapidamente nuove applicazioni containerizzate per gestire il traffico in crescita e semplificare il processo di installazione, migliorando anche la sicurezza. Kubernetes nasce proprio con questo intento: semplificare una gestione altrimenti complessa in cui soluzioni come Docker Swarm o Docker Machine avevano delle "mancanze" in un contesto enterprise.

Ma chi c'è dietro a Kubernetes?

Kubernetes: breve storia

Le grandi aziende di Internet come Google, Twitter ed eBay hanno utilizzato architetture basate su container per anni; mentre il container, secondo quanto visto finora, offre una soluzione elegante per la creazione di applicazioni basate su microservizi, Docker non include tutto il necessario per distribuire e gestire diversi container che devono lavorare insieme e scalare all'aumentare della domanda. Quando ognuna di queste aziende ha iniziato a creare i propri strumenti per lo sviluppo, a gestirne l'implementazione e il ridimensionamento, ci si è resi conto che il lavoro era eccessivo perché Docker potesse gestirne tutto il carico.

Senza voler rendere questo capitolo noioso, riportiamo solo alcuni dei punti fondamentali che ci permettono di comprendere al meglio la nascita e la crescita di questo ambizioso progetto che, in qualche modo, ci ha portato alla scrittura di questo manuale.

Nascita di Borg

Google presenta il sistema Borg (<https://research.google/pubs/pub43438/>) intorno al 2003-2004: è iniziato come un progetto su piccola scala, con circa 3-4 persone inizialmente in collaborazione per produrre una versione aggiornata del nuovo motore di ricerca di Google. Borg, invece, era un sistema interno di gestione dei cluster su larga scala, che eseguiva centinaia di migliaia di *task*, provenienti da molte migliaia di applicazioni diverse, su molti cluster, ciascuno con un massimo di decine di migliaia di macchine. Dall'esperienza di Borg, si passa a Omega (<https://research.google/pubs/pub41684/>), un sistema di gestione dei cluster con uno *scheduler* per pianificare il rilascio delle applicazioni flessibile e scalabile indirizzato a cluster di calcolo di grandi dimensioni. Questo strumento, quindi, si integra con Borg, introducendo uno strumento che permette la gestione del parallelismo delle applicazioni, di uno stato condiviso e anche della concorrenza, tutti aspetti fondamentali per il ciclo di vita di un cluster. Questo finché nel 2014 non annunciano il progetto Kubernetes: ormai Google esegue i suoi carichi di lavoro in produzione sfruttando Borg da più di un decennio e praticamente tutto in Google viene eseguito sotto forma di container. Kubernetes traccia una linea diretta con Borg, tanto che molte delle

persone che hanno lavorato originariamente al progetto sono le protagoniste della nascita di uno degli orchestratori più diffusi al mondo. Il 7 giugno del 2014 viene effettuato il primo commit su GitHub relativo al progetto dedicato a Kubernetes ed è subito un successo; appena un mese dopo, aziende del calibro di Red Hat, Microsoft e IBM si uniscono alla community che si crea intorno a Kubernetes.

NOTA

In omaggio a una delle serie più geek di sempre, la tecnologia Borg prende il nome dal gruppo di cyborg *Borg*, una specie immaginaria che fa parte dei personaggi più ricorrenti della saga di Star Trek: il fatto che i Borg siano organizzati come una sorta di "mente centralizzata" collettiva ben rappresenta il senso del progetto!

Arriva il KubeCon

Ben presto nasce anche la prima conferenza a tema: il 9-11 novembre del 2015 si tiene il primo KubeCon a San Francisco, dove si inaugura la creazione della community dedicata a questa tecnologia: il suo obiettivo era (ed è ancora) quello di tenere discorsi tecnici di esperti con l'obiettivo di stimolare la creatività e promuovere l'istruzione su Kubernetes. Nel 2016 diventa quindi un fenomeno, una vera e propria rivoluzione: questo anno segna moltissime novità per questo progetto, come il rilascio della prima versione di Helm, il gestore di pacchetti per Kubernetes, la prima KubeCon EU, ossia la prima conferenza europea di Kubernetes con quasi 500 partecipanti. Man mano che la risonanza aumenta, Kubernetes inizia a sfornare i primi oggetti che rappresentano una novità nel mondo dell'orchestrazione: nascono i PetSet, una risorsa che oggi assume un nome diverso (no spoiler, lo vedremo più in là!), così come lo sviluppo e l'implementazione di Minikube, uno strumento che semplifica l'esecuzione di Kubernetes in locale.

In questo contesto, bisogna parlare anche del caso Pokemon GO!

(<https://cloud.google.com/blog/products/containers-kubernetes/bringing-pokemon-go-to-life-on-google-cloud>): infatti, sebbene possa sembrare strano, questo caso d'uso rappresenta la più grande implementazione Kubernetes mai realizzata su Google Container Engine. I Pokemon sono stati nel cuore di ogni millennial fin dall'infanzia. Poiché Pokemon è stato creato dalla nota azienda Nintendo Co., la maggior parte dei giochi è sempre stata disponibile solo su piattaforme Nintendo Switch. Pokemon GO!, invece, è stato il primissimo gioco Nintendo a essere disponibile su dispositivi diversi dai Nintendo Switch, portandolo di fatto sui dispositivi mobili, come cellulari e tablet: questa novità ha generato un enorme scalpore nel mercato prima che il gioco uscisse e, dopo il rilascio, le stime fatte dall'azienda hanno superato di molto qualsiasi aspettativa. Entro 15 minuti dal lancio in Australia e Nuova Zelanda, il traffico di giocatori è aumentato di 6 volte, con un risultato migliore di qualsiasi aspettativa del team. Il clamore è stato tale da indurre i suoi creatori alla pubblicazione di un caso di studio per illustrare la dimensione e diffusione di questo fenomeno e il contributo fondamentale di Kubernetes alla scalabilità delle risorse necessarie per sostenere rapidamente le continue richieste di risorse da parte dei giocatori di Pokemon GO! L'aspetto ancora più interessante è stato poter dedicare a ogni cliente del gioco un "pezzo" del cluster dedicato alle sue attività, isolandolo dagli altri giocatori.

Kubernetes su cloud

Questa esperienza ha finalmente consolidato il ruolo di Kubernetes come giocatore di punta nel mercato degli orchestratori, rendendolo di fatto l'apripista per altri progetti che si integrano con questa tecnologia, come Prometheus, Fluentd e OpenTracing. Nel 2017 Kubernetes viene adottato in maniera diffusa dalle aziende, tanto che Google e IBM annunciano Istio, una tecnologia open che fornisce un modo per connettere, gestire e proteggere senza problemi reti costituite da diversi microservizi, indipendentemente dalla piattaforma e dall'origine.

Un esempio di grande azienda che adotta Kubernetes per le sue applicazioni è GitHub: tutte le richieste Web e le API vengono servite dai container in esecuzione nei cluster distribuiti on cloud. E, mentre la CNCF incrementa il numero dei suoi membri grazie alla presenza di grandi aziende come Oracle, Kubernetes vede le prime installazioni open source anche su sistemi Windows o Oracle systems e soprattutto le prime implementazioni di cluster completamente o parzialmente gestiti sui principali cloud provider. Amazon annuncia nel 2017 Elastic Container Service per Kubernetes, un servizio per distribuire, gestire e ridimensionare le applicazioni containerizzate direttamente tramite AWS. Nello stesso anno, abbiamo la presentazione di Kubeflow, uno stack di strumenti per attività di machine learning altamente componibile, portatile e scalabile, creato appositamente per Kubernetes.

Nel 2018, DigitalOcean si tuffa nel mondo Kubernetes e annuncia un nuovo prodotto che ospita questa tecnologia per fornire una piattaforma di gestione e orchestrazione dei container come servizio gratuito in aggiunta alle opzioni di archiviazione e cloud computing esistenti. Anche Amazon fa progressi e lancia EKS, un sistema gestito che semplifica il processo di creazione, protezione, funzionamento e manutenzione dei cluster Kubernetes offrendo i vantaggi dell'elaborazione basata su container alle organizzazioni che desiderano concentrarsi sulla creazione di applicazioni invece di configurare un cluster Kubernetes da zero. Anche Microsoft lavora in questo senso, e annuncia quasi contemporaneamente il servizio Azure Kubernetes (AKS), per distribuire e gestire le proprie app Kubernetes di produzione.

La squadra dietro Kubernetes

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, anche Alibaba e Amazon sono entrate a far parte della cerchia delle grandi aziende che adottano 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 loro codice a questa tecnologia dalla connotazione fortemente infrastrutturale.

Google ha rilasciato Kubernetes come progetto open source con l'obiettivo di standardizzare la gestione delle applicazioni containerizzate; 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).

Kubernetes = K8S?

Ti sei chiesto perché Kubernetes viene abbreviato in K8S? I cosiddetti "numeronomi" sono apparsi alla fine degli anni Ottanta. Ci sono diverse storie su come le persone abbiano effettivamente iniziato a usarli; tuttavia, l'idea condivisa è che, per semplificare la comunicazione, il settore IT abbia iniziato a utilizzare i numeronomi per abbreviare le parole: prendere la prima e l'ultima lettera di una parola e collocarvi nel mezzo un numero che ne desse il suono corretto riduceva di molto il lavoro. Per esempio, il termine "*i18n*" deriva dall'ortografia di "*internationalization*", che inizia con la lettera "i", continua con 18 lettere e finisce con la lettera "n". In modo simile, Kubernetes è composta dalla lettera "k" più 8 lettere e infine la lettera "s".

A oggi, Kubernetes è molto popolare. Secondo lo State of Cloud Native Development Report di CNCF (Cloud Native Computing Foundation) per il primo trimestre del 2021, ci sono 6,8 milioni di persone che sviluppano in ambito cloud in tutto il mondo, di cui 5,6 milioni utilizzano Kubernetes, con una crescita del 67% rispetto all'anno precedente, con 10,2 milioni di container in esecuzione. La CNCF ha inoltre stimato che ci sono 26,8 milioni di persone che sviluppano a livello globale, quindi circa 1 utente su 5 nel mondo utilizza attualmente Kubernetes.

Il timoniere di cui Docker aveva bisogno

Il nome "Kubernetes" deriva da un'antica parola greca che sta per "timoniere" (qualcuno che dirige una nave, come una nave per il trasporto di container) e ritroviamo questo nome nel logo del progetto che mostra il timone di una nave. Questo ha sette lati, in omaggio al nome originale del progetto; Beda, McLuckie e il suo team inizialmente intendevano chiamare la tecnologia Kubernetes *Project Seven*, che prende il nome da Seven of Nine, un simpatico personaggio di *Star Trek* appartenente al gruppo dei Borg.

Perché Kubernetes

Ormai è chiaro: i container sono un modo semplice e valido per raggruppare ed eseguire le tue applicazioni. In un ambiente di produzione, però, è necessario gestire i container che eseguono le applicazioni e assicurarsi che non vi siano tempi di inattività. Per esempio, se un container si arresta, deve essere avviato un altro container in sostituzione del primo. Per farlo, potresti trovarsi nelle condizioni di dover entrare tramite SSH sul server, avviare il container e poi verificare che tutto stia funzionando a regola d'arte. È evidente quanto sarebbe più semplice se questo comportamento fosse gestito da un opportuno sistema: Kubernetes si occupa della scalabilità e del *failover* per la tua applicazione, fornisce dei modelli per agevolare il rilascio delle applicazioni secondo delle policy predefinite e molto altro ancora. Tra le diverse funzionalità, abbiamo uno storage gestito per il cluster, *rollout* e *rollback* automatizzati per adattarne lo stato effettivo a quello desiderato a una velocità controllata, resilienza tramite dei test che controllano l'integrazione degli applicativi e molto altro.

Senza indulgere ulteriormente, voliamo verso il mondo dell'orchestrazione, dando prima un'occhiata alle soluzioni presenti sul mercato: non vogliamo che Kubernetes rappresenti la soluzione a tutti i nostri problemi, ma che sia una scelta consapevole e adatta al nostro caso d'uso!

Che cosa abbiamo imparato

- Abbiamo visto come avviene storicamente il passaggio da strumenti di containerizzazione al contesto di orchestrazione.
- Abbiamo ripercorso i punti salienti del successo di Kubernetes, attraverso le aziende che hanno sposato il progetto e i suoi casi d'uso.
- Abbiamo introdotto le principali funzionalità che lo rendono uno strumento perfetto per i nostri scenari di lavoro in determinati contesti, che verranno approfonditi a breve.

Orchestrazione

La vita non ti presenta sempre l'opportunità perfetta al momento giusto. Le opportunità arrivano quando meno te le aspetti, o quando non sei pronta... Le opportunità, quelle buone... sono difficili da riconoscere. Sono rischiose. E ti mettono alla prova.

– Susan Wojcicki, CEO di YouTube

Lo scopo di questo capitolo è di fare un po' di chiarezza sulle diverse tipologie di architetture a oggi disponibili, così da entrare un po' più in contatto con tutte le parole che d'ora in poi saranno sempre più frequenti. Non solo: è importante che questo manuale possa illustrare al meglio i vantaggi nell'utilizzare una tecnologia come Kubernetes, ma anche esporre le alternative presenti sul mercato e soprattutto perché scegliere di adottare l'una o l'altra, senza mai scordarsi che questi strumenti non sono una panacea, ma più che altro strumenti che ci faciliteranno un po' il lavoro su ordini di utenti sempre più grandi.

Modelli

Nel vasto mondo dell'offerta di servizi cloud c'è ancora una certa confusione riguardo alla differenza tra i termini *CaaS*, *PaaS*, *IaaS* e così via. Questo perché il termine *cloud computing* viene spesso associato a servizi che sono molto diversi tra loro, come per creare un unico gruppo di tecnologie che sfruttano il cloud anche se per caratteristiche e finalità assai diverse. Facciamo allora un passo indietro e cerchiamo di analizzare queste parole una per volta, analizzando le differenze nella loro implementazione e nel loro utilizzo, portando in evidenza quali sono le ragioni che hanno portato negli ultimi anni al loro successo.

Quando un'azienda dispone di un'infrastruttura on-premise (ossia un tipo di installazione ed esecuzione del software direttamente su macchina locale, sia essa aziendale o privata), questa si dovrà occupare dell'intera gestione del processo produttivo delle proprie applicazioni: significa provvedere alla manutenzione dell'infrastruttura fisica (quindi l'hardware, gestione di rete e persistenza), ma anche all'organizzazione di tutti gli aspetti di gestione dei sistemi e degli ambienti operativi necessari allo sviluppo, al test e alla distribuzione delle applicazioni agli utenti finali, oltre alla sicurezza fisica della piattaforma, la gestione degli eventuali guasti e via dicendo.

Con l'avvento del cloud computing, si sono superate una serie di problematiche come quelle elencate, potendo astrarre molti di questi aspetti, a partire dall'hardware: esistono infatti dei *provider*, ossia dei fornitori, che possono garantirci l'utilizzo di piattaforme già pronte all'uso che devono essere configurate secondo le nostre necessità. Possiamo anche disporre di macchine che siano già configurate con un sistema di base e manutenute da un fornitore esterno, lasciando alle persone che le utilizzano la sola installazione degli applicativi e/o dei software necessari al loro corretto funzionamento. Questi sono solo alcuni degli esempi delle architetture che esamineremo in breve, dove esiste un rapporto tra chi utilizza i sistemi (anche se in forme diverse), ossia il *cliente*, e il *fornitore*, che solleva il cliente da tutti questi aspetti.

Immaginiamo quindi di voler classificare questi modelli in base a una serie di caratteristiche dove, a seconda del modello, assegneremo a cliente e fornitore le proprie responsabilità, per cercare di

comprendere meglio quali sono le reali differenze tra quelli a oggi esistenti. All'interno di questo schema inseriamo la gestione delle applicazioni, dei dati, dell'ambiente di esecuzione, dei container, del sistema operativo, dei sistemi di virtualizzazione, della persistenza e della rete con relativa configurazione: tutti aspetti che in un sistema on-premise sono perlopiù assegnati all'azienda che gestisce le macchine, anche se è possibile avvalersi dell'aiuto di fornitori esterni o persone esperte del dominio.

Possiamo quindi riassumere le caratteristiche elencate in precedenza in questo schema, e utilizzarle man mano per poter delineare i diversi modelli che descriveremo nelle sezioni che seguono.

NOTA

Esistono moltissimi acronimi che servono a descrivere un'architettura specifica e le relative responsabilità di fornitori e utilizzatori. Questo capitolo non vuole però essere una carrellata di informazioni prolisse e facilmente fruibili online, motivo per cui ci concentreremo su quelli che rivestono una certa importanza in ambito *cloud*, tralasciando altri modelli, come FaaS o RaaS.

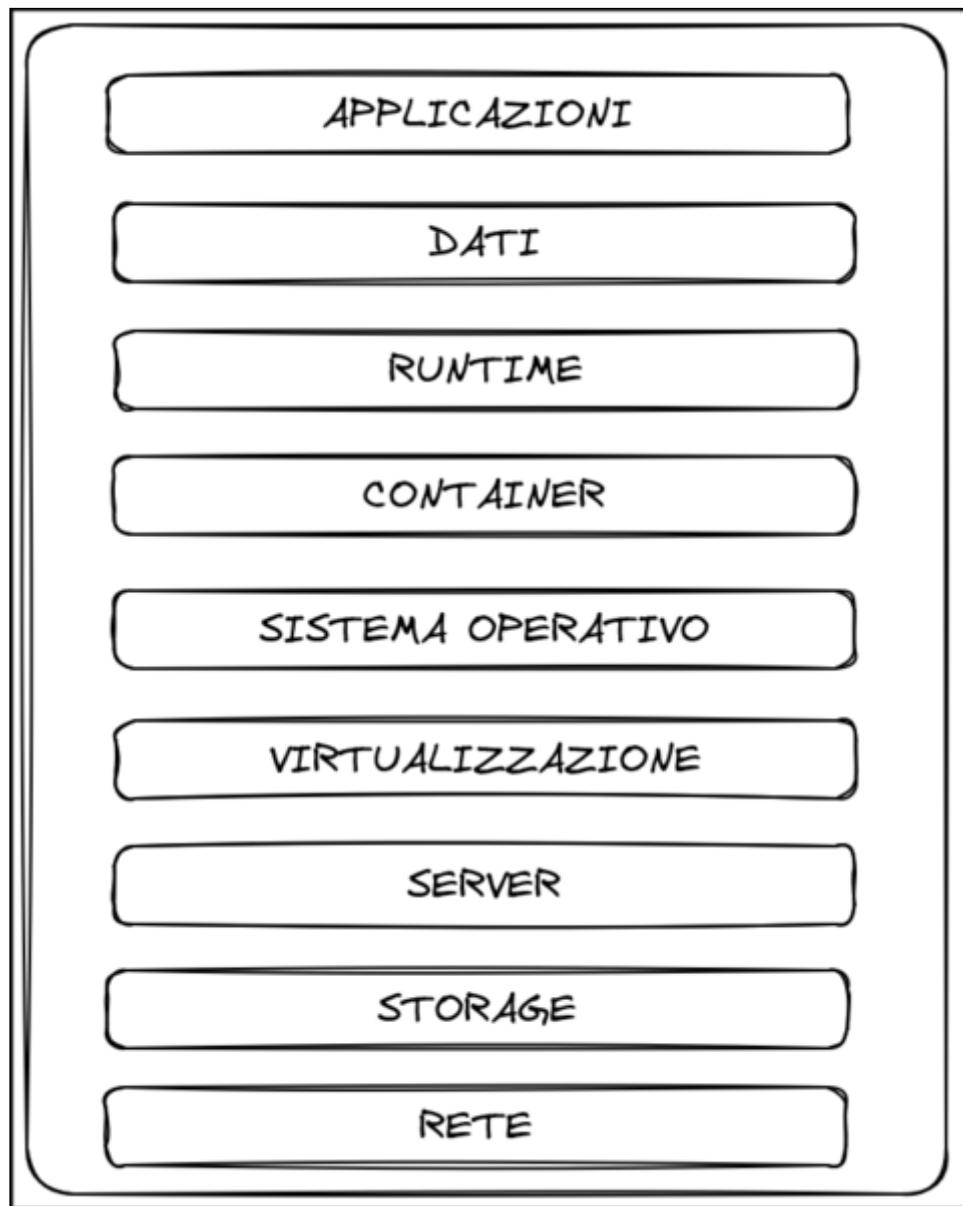


Figura 2.1 Struttura generica di un'infrastruttura.

Infrastructure as a Service

IaaS (acronimo di *Infrastructure as a Service*) è un tipo di servizio di cloud computing che offre risorse di calcolo, persistenza e connettività essenziali secondo necessità e con pagamento in base al consumo. Il provider del servizio IaaS fornisce la virtualizzazione, lo storage, la rete e i server, eliminando l'esigenza di un datacenter on-premise ed evitando all'utente tutte le attività fisiche di aggiornamento e manutenzione necessarie per tali componenti.

Nella maggior parte dei casi, l'utente che utilizza un servizio che segue questo modello può controllare completamente l'infrastruttura tramite un'API o una dashboard. Questo approccio è tra i più flessibili, perché consente di adattare il sistema alle proprie esigenze, effettuare l'upgrade e aggiungere le risorse, come lo storage su cloud, quando necessario, senza costringere l'utente a prevedere necessità future e relativi costi.

IaaS ti permette di evitare il costo e la complessità dell'acquisto e della gestione di server fisici e dell'infrastruttura del data center. Un provider di servizi di cloud computing, per esempio *Azure*, gestisce l'infrastruttura, la sua disponibilità fisica e la relativa manutenzione, mentre tu, in qualità di cliente, installi, configuri e gestisci il software, tra cui sistemi operativi, middleware e applicazioni.

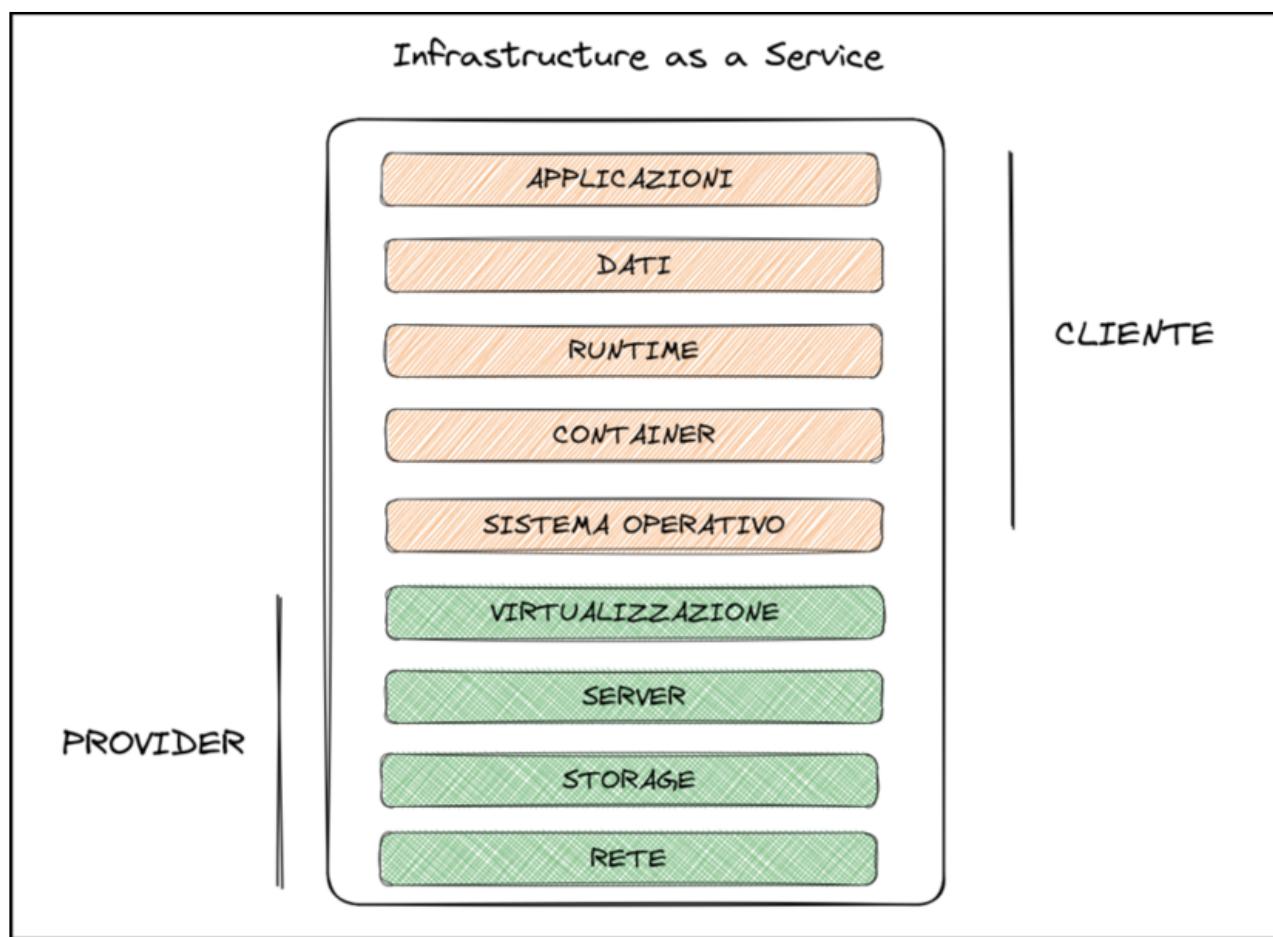


Figura 2.2 Responsabilità all'interno di una struttura di un IaaS.

Il principale vantaggio di un approccio come questo è la grande libertà che ne deriva in termini di personalizzazione: non c'è limite a ciò che possiamo installare come sistema operativo, piuttosto che come software e aggiornamenti. Come visibile anche in Figura 2.2, il provider fornisce il sistema di base, la configurazione di rete tra le macchine e lo spazio di archiviazione e il cliente può utilizzarle secondo esigenza. Ma, come ci insegna Ben Parker in Spider-Man, "da un grande potere derivano

grandi responsabilità": in questo caso stiamo infatti godendo di un'infrastruttura che non è fisicamente di nostra responsabilità, ma di cui gestiamo gran parte delle necessità, con tutte le relative conseguenze. Un problema tecnico, oppure una momentanea indisponibilità del sistema, è gestito dal provider, mentre aggiornamenti, problemi di sicurezza nei pacchetti di sistema e molto altro, è a carico di chi ne fruisce.

Container as a Service

Il modello CaaS (o *Container as a Service*) si differenzia dallo IaaS principalmente per il suo approccio alla virtualizzazione: risulta anche abbastanza esplicativo tramite l'acronimo l'utilizzo della tecnologia dei container per garantire al cliente un servizio che permetta l'utilizzo di tali oggetti per il rilascio del proprio software. Viene da sé che non sia più necessario, all'interno di un'infrastruttura di questo tipo, installare e configurare il software che ospiterà i container, né manutenere il sistema operativo.

Il vantaggio dei container è proprio quello di encapsulare le applicazioni (filesystem incluso) e offrire quindi completa libertà di scelta sui linguaggi e sui framework di programmazione, per poter garantire alle persone che sviluppano la piena portabilità delle applicazioni, indipendentemente dall'infrastruttura sottostante. In questo caso, il compito del provider è quello di fornire un servizio che ci permetta di istanziare le immagini che abbiamo predisposto all'interno di un ambiente pronto a ospitarle.

In pratica, nel CaaS si incontrano la flessibilità del modello IaaS e il livello di astrazione del modello PaaS, con l'aggiunta di un livello di controllo delle risorse (e quindi anche dei costi) in più: possiamo parlare quindi di un'infrastruttura elastica, modulabile sulle esigenze del cliente, alla quale è possibile aggiungere risorse *on demand*. Un ulteriore vantaggio del CaaS è dato dalla possibilità di organizzare più container in architetture complesse attraverso l'utilizzo del tool di orchestrazione, proprio come Kubernetes. In tal modo, è possibile sviluppare applicazioni multi-container gestite in maniera automatizzata.

Il provider offre quindi il framework, o la piattaforma di orchestrazione, su cui i container vengono distribuiti e gestiti, ed è attraverso questa orchestrazione che le funzioni IT chiave vengono automatizzate, mentre le persone che la utilizzano possono acquistare le risorse che desiderano (bilanciamento del carico, dimensionamento delle applicazioni ecc.), aumentando l'efficienza e garantendo un risparmio economico.

Kubernetes è di per sé un CaaS, dal momento che tratta i diversi container come singoli servizi che può gestire, istanziare e ridimensionare in autonomia; se poi guardiamo ai cloud provider, AWS Fargate, per esempio, è un servizio che adotta il modello CaaS, in quanto permette istanziare dei container senza la necessità di provvedere all'infrastruttura sottostante o alla sua gestione.

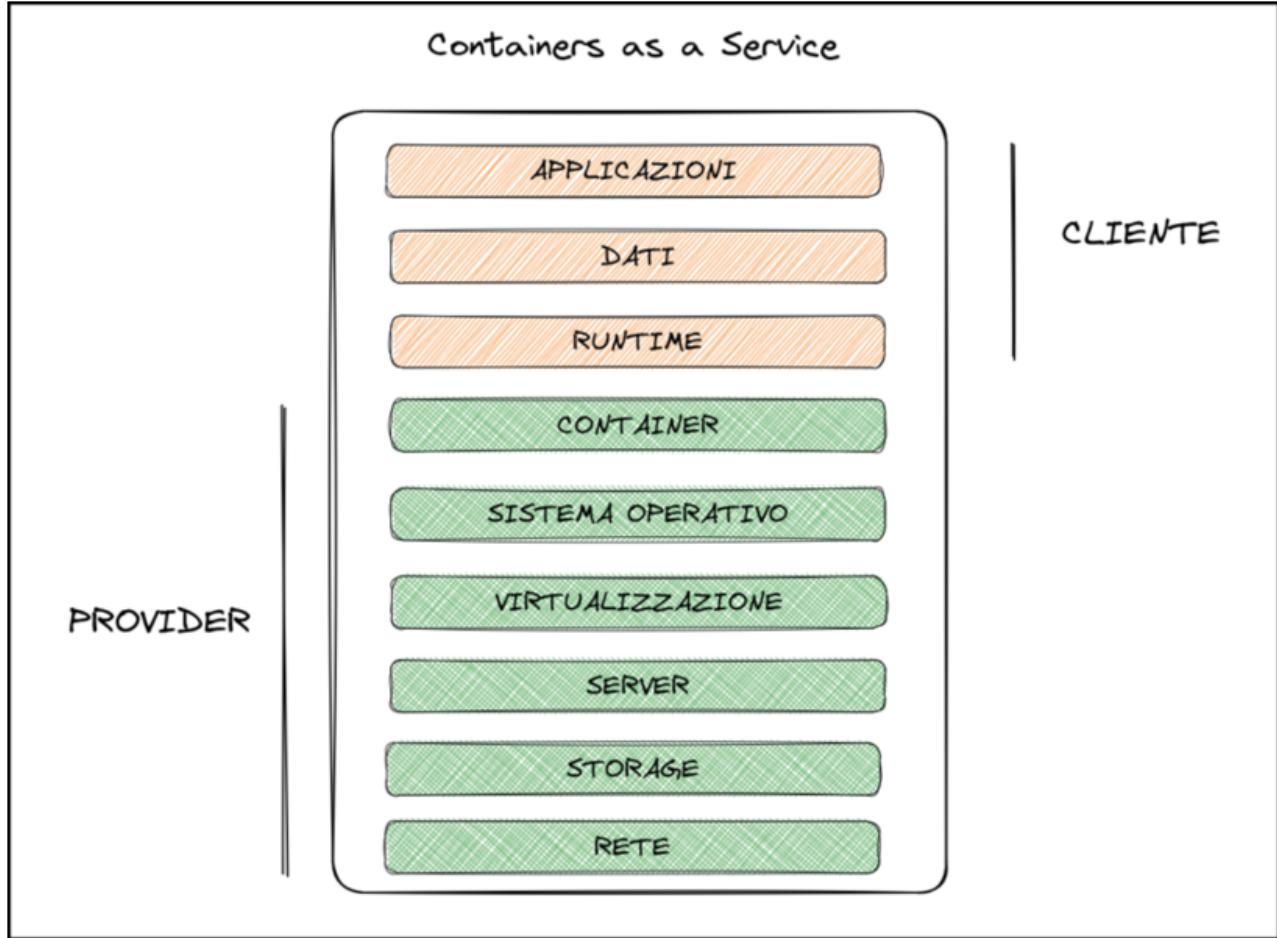


Figura 2.3 Responsabilità all'interno di una struttura di un CaaS.

Platform as a Service

Il modello *Platform-as-a-Service* (abbreviato in PaaS) è un servizio in cui la piattaforma software per il rilascio delle applicazioni viene fornita da un provider ed è pensato principalmente per gli sviluppatori, di modo che possano realizzare, eseguire e gestire le proprie soluzioni senza dover creare e manutenere l'infrastruttura o la piattaforma normalmente associata a tali processi.

Le piattaforme PaaS possono essere eseguite nel cloud o anche su infrastrutture on-premise; nel caso di un cloud provider, questo modello prevede che sia hardware che software siano ospitati nell'infrastruttura del provider, il quale mette a disposizione la piattaforma all'utente come soluzione integrata e accessibile, per esempio, tramite una connessione Internet.

Un esempio di servizio PaaS basato su Kubernetes che può fornire una piattaforma per l'esecuzione, la scalabilità e la gestione delle containerizzate è OpenShift. Le stesse piattaforme cloud sono un tipo di servizio PaaS, come quello fornito da Alibaba Cloud, Microsoft Azure, Google Cloud, Amazon Web Services (AWS) e IBM Cloud. Il modello PaaS funziona bene per le piccole e grandi imprese per due motivi fondamentali: in primo luogo, è conveniente, consentendo alle organizzazioni più piccole di accedere a risorse all'avanguardia senza le competenze infrastrutturali o costi particolarmente elevati, offrendo un percorso che acceleri lo sviluppo del software. Altri vantaggi includono la sicurezza, in quanto i fornitori di PaaS investono molto nella tecnologia e nelle competenze in termini di sicurezza informatica, nonché una scalabilità pressoché infinita, che garantisce grande flessibilità, per permettere

a chi lavora con questa tipologia di infrastruttura di accedere e lavorare alle proprie applicazioni da qualsiasi luogo. Sicuramente, tra i problemi principali va considerato il rischio di lock-in, ovvero che i clienti possano essere vincolati al provider scelto a causa degli investimenti fatti, avendo cucito la soluzione proposta agli utenti finali proprio sui servizi specifici di tale fornitore, creando di fatto delle possibili incompatibilità con altre soluzioni esistenti.

Se utilizzata in modo efficace, l'architettura PaaS può ridurre il carico di lavoro che grava sul personale tecnico, così come i tempi dovuti alla messa sul mercato di una soluzione: la pressione a consegnare sempre più rapide e l'aumento dei debiti tecnici può essere in parte risolta grazie a una piattaforma pronta all'uso che permette alle persone che sviluppano di concentrarsi sul prodotto finale, sulla sua qualità, senza dover temere una configurazione o un'installazione complessa. Questo tipo di modello infatti, tra le altre cose, incoraggia l'apprendimento e i comportamenti innovativi, come l'adozione di un flusso di lavoro che segua la filosofia DevOps, attraverso le diverse funzionalità che vengono messe a disposizione.

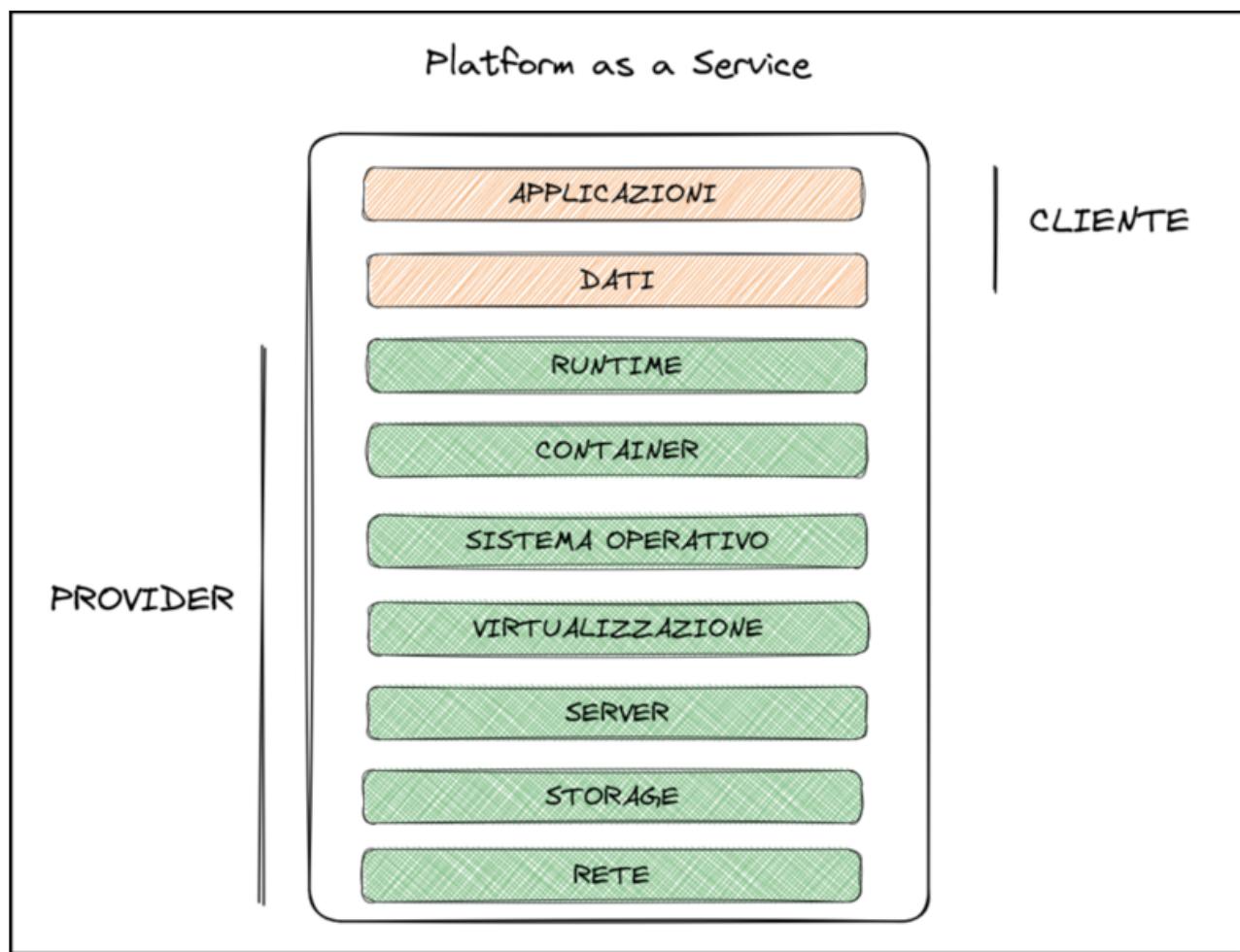


Figura 2.4 Responsabilità all'interno di una struttura di un PaaS.

Software as a Service

L'ultimo modello che tratteremo in questo capitolo è il modello SaaS: l'acronimo sta per *Software as a service* e si tratta del tipo di architettura che ha reso il cloud computing così diffuso e utilizzato. In pratica, il provider mette un servizio (o software) a disposizione degli utenti, in forma gratuita o a pagamento.

Un esempio comune di applicazione SaaS è la posta elettronica che utilizziamo tramite il Web come Gmail o AOL, potendo inviare e ricevere e-mail senza dover gestire eventuali nuove funzionalità al prodotto di posta elettronica o mantenere i server e i sistemi operativi su cui viene eseguito il programma di posta elettronica. Un altro esempio sono i software per la contabilità, o anche i software per la gestione delle risorse umane o quelli per la sicurezza, o anche Netflix, Slack, Zoom e molto altro: tutti servizi di cui usufruiamo, a cui paghiamo un abbonamento, ma la cui infrastruttura sottostante rimane un mistero (o quasi).

Questo modello si sposa perfettamente con il cloud computing, in quanto i fornitori di servizi in SaaS di solito ospitano le applicazioni e i dati sui propri server, oppure utilizzano i server di un cloud provider di terze parti per poi distribuirli all'utente finale tramite, per esempio, una licenza. Una volta adottato un servizio o una soluzione SaaS, il provider concede al cliente l'accesso all'applicazione attraverso la registrazione e il login nel browser Web, o tramite altri meccanismi di autenticazione.

Tra i vantaggi di questo modello c'è sicuramente la riduzione dei costi iniziali del software tradizionale, come le licenze, l'installazione o la gestione dell'infrastruttura. Inoltre, non è necessario investire in risorse informatiche aggiuntive per l'esecuzione del software, poiché il fornitore gestisce tutto sui propri server. Altro vantaggio è la scalabilità, in quanto l'utilizzo in genere è gestito tramite un numero che indica gli utenti o le licenze per mese, in modo che un'azienda utilizzi solo ciò di cui ha bisogno. L'aggiunta di più risorse e l'estensione delle applicazioni con l'espandersi delle esigenze significano anche convenienza e scalabilità.

Solitamente, poi, il provider fornisce aggiornamenti automatici e frequenti per garantire un vantaggio al cliente in termini di competizione sul mercato, e sono tutti a carico del fornitore. Non è necessario eseguire la manutenzione o supportare versioni legacy del software, ma tutti gli ultimi aggiornamenti vengono erogati direttamente all'utente finale. Ultimo aspetto, non scontato, è relativo alla sicurezza: questa viene gestita dal fornitore ed è fondamentale assicurarsi che il provider disponga di tutte le certificazioni di sicurezza necessarie a garantire un servizio altamente affidabile.

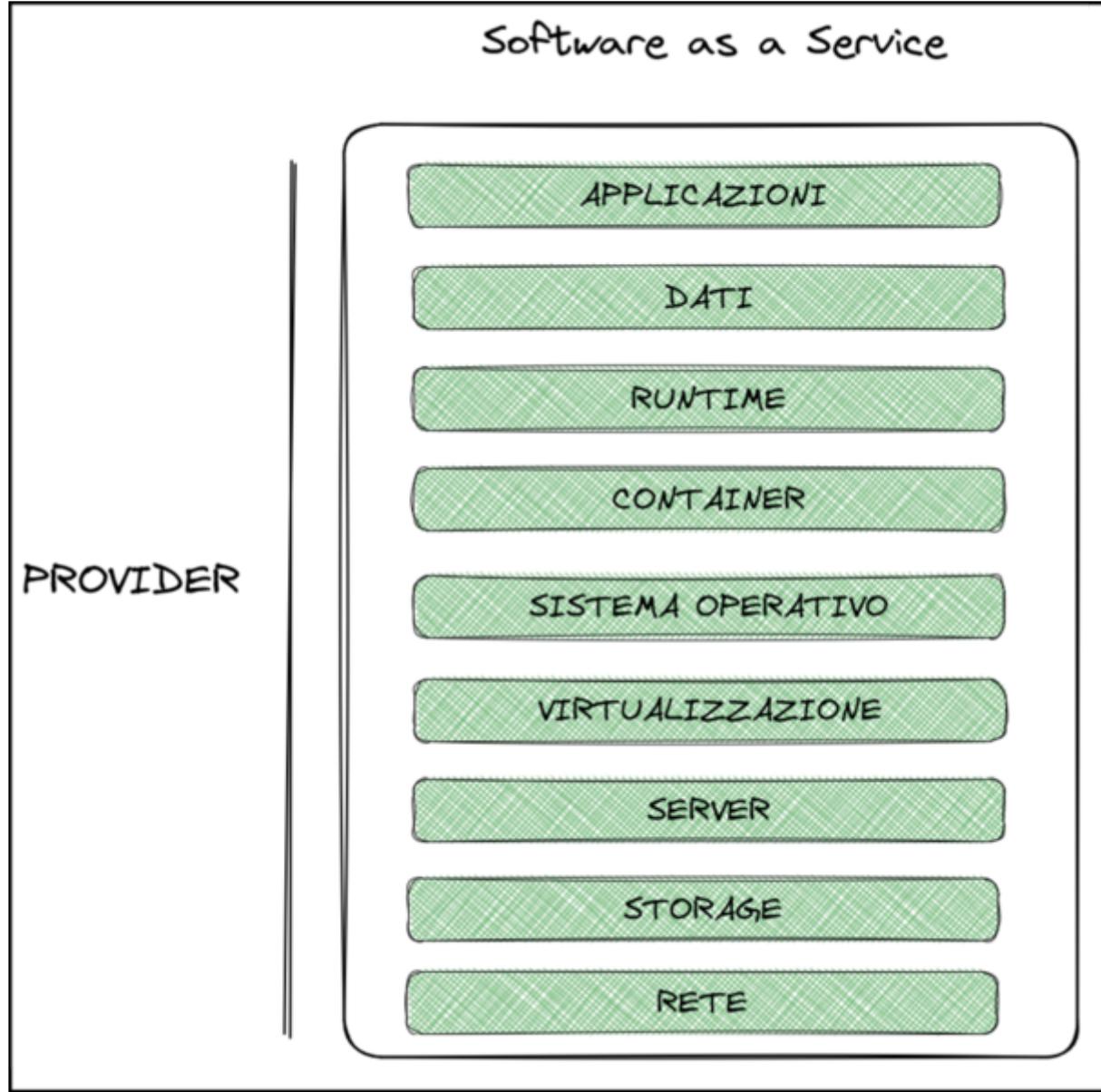


Figura 2.5 Responsabilità all'interno di una struttura di un SaaS.

Soluzioni

Finora abbiamo descritto i modelli più comuni di software per il cloud: questo ci ha aiutato a capire un po' di più quale tipologia vogliamo approcciare quando decidiamo di lavorare con tecnologie come Kubernetes e di quali responsabilità ci vogliamo far carico. Sul mercato esistono però tantissime soluzioni che permettono di gestire i container, oltre a Docker: quale di queste utilizzare e perché? Proviamo a dare un'occhiata rapida a quelle più comuni e a capire nel dettaglio quali sono le soluzioni più adatte a una serie di esigenze che incontreremo durante il nostro percorso professionale.

Potresti chiederti: “Docker o Docker Compose non sono irrilevanti a questo punto?”. Data l'incredibile popolarità di Kubernetes, potresti essere incline a pensarla, ma Docker è ancora in circolazione ed è importante parlarne, per comprendere che Kubernetes non è la soluzione a tutti i problemi, ma piuttosto una soluzione a uno o più problemi specifici.

Docker Swarm e Docker Compose

Docker (o in particolare il comando `docker`) viene utilizzato per gestire i singoli container, così come Docker Compose viene utilizzato per gestire applicazioni composte da più servizi, mentre Kubernetes è uno strumento di orchestrazione dei container. Docker è (in molti casi) la tecnologia di base utilizzata per i container e può distribuire singole applicazioni containerizzate durante le fasi di sviluppo, ma il suo utilizzo è fondamentale anche quando abbiamo bisogno di lavorare su un singolo container; Docker Compose viene utilizzato per configurare e avviare più container Docker sullo stesso host, quindi non è necessario avviare ogni container separatamente. Docker Swarm è uno strumento di orchestrazione che nasce all'interno della stessa Docker Inc. che consente di eseguire e connettere i container su più host. Kubernetes è uno strumento di orchestrazione dei container simile a Docker Swarm, ma ha una serie di funzionalità che volgono alla facilità di automazione e alla capacità di gestire un carico di lavoro più elevato.

Va anche detto che la più grande differenza tra Docker Swarm e Kubernetes è la facilità d'uso: niente vale più dell'esempio pratico di come ognuno di questi due gestisce il networking. Quando costruisci un cluster di container con Docker Swarm, questi sono in genere disponibili all'interno della tua rete perché utilizzi una porta esterna che punta a una porta interna del tuo host tramite il comando stesso. In altre parole, non devi configurare un livello di rete separato all'interno dei tuoi file YAML.

Tuttavia, con Kubernetes, devi configurare un livello di rete (utilizzando dichiarazioni come `hostNetwork: true` di cui parleremo successivamente) e quindi, senza l'aggiunta di un livello di rete accessibile tramite LAN all'interno dei tuoi file YAML, non sarai in grado di accedere ai tuoi container da nessuna parte se non dal cluster Kubernetes stesso. Questo potrebbe scoraggiarti e darti l'idea che questa non sia la strada giusta per te: Kubernetes porta con sé una maggiore scalabilità e automazione del cluster e delle applicazioni che ospita che Docker Swarm non può in alcun modo garantire.

Quindi, rispondiamo alla domanda: quando dovrresti usare Docker, Docker Swarm o Kubernetes?

- *Docker*: quando si desidera distribuire un singolo container (accessibile dalla rete) o sviluppare in locale le proprie applicazioni;
- *Docker Compose*: quando si desidera distribuire più container su un singolo host da un singolo file YAML;
- *Docker Swarm*: quando si desidera distribuire un cluster di nodi (più host) per un'applicazione semplice e scalabile
- *Kubernetes*: quando dobbiamo gestire un'ampia implementazione di container scalabili e automatizzati, di cui vogliamo gestire anche lo stato di integrità, o quando lavoriamo in ambienti di produzione particolarmente complessi.

E questo (almeno dal mio punto di vista) dovrebbe aiutarti a capire le differenze, le somiglianze e i casi d'uso tra queste tecnologie.

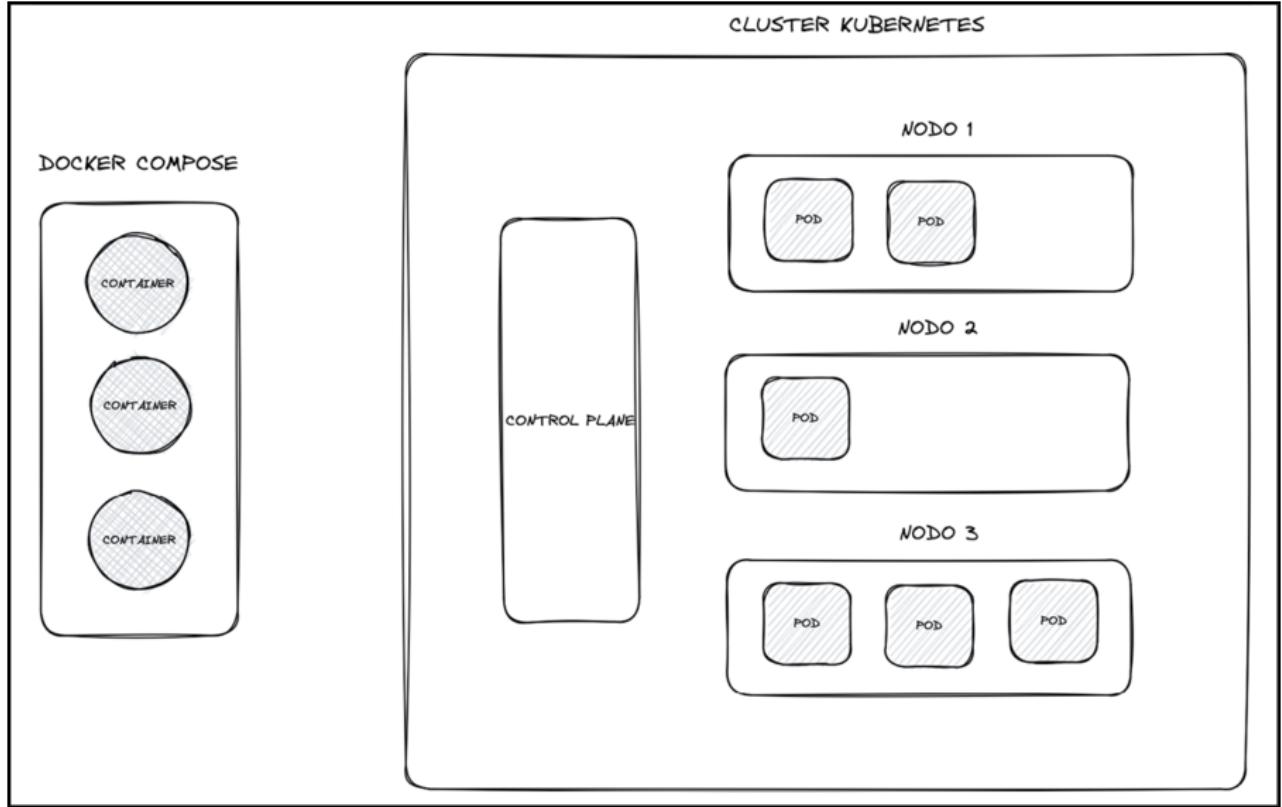


Figura 2.6 Rappresentazione delle architetture di Docker Compose, Docker Swarm e Kubernetes.

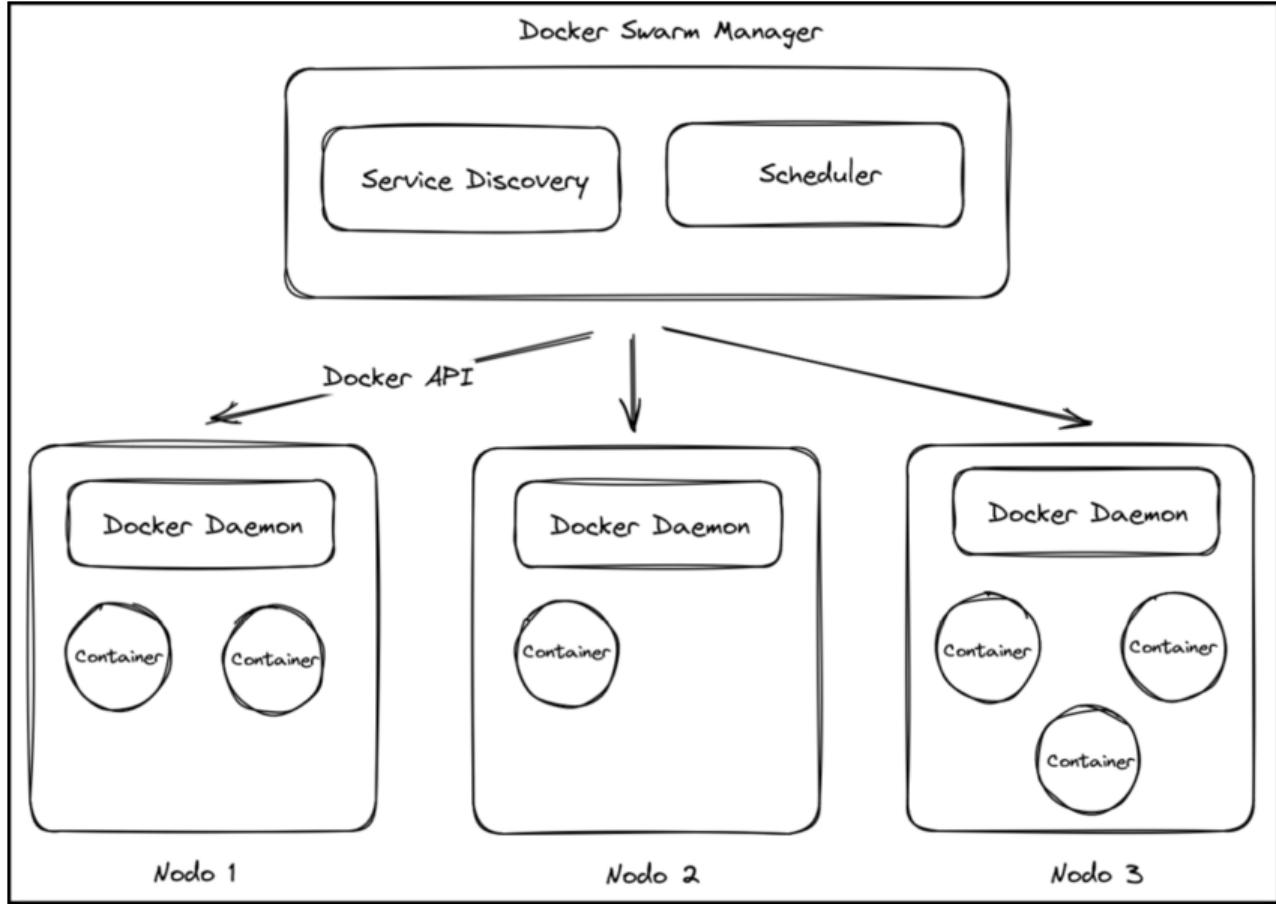


Figura 2.7 Rappresentazione del funzionamento interno di Docker Swarm.

Docker Machine

Docker Machine è una macchina virtuale molto piccola che esegue un'istanza del motore Docker. Ciò significa che possiamo eseguire i container Docker al suo interno e quindi può essere inteso come un container di container.

In altre parole, Docker Machine è uno strumento per il provisioning e la gestione degli host "dockerizzati", ossia host con installato Docker Engine. In genere, installi Docker Machine sul tuo sistema locale e utilizzi il comando `docker-machine` insieme al comando `docker` per gestire le diverse istanze delle macchine e i container ospitati in esse. Questi sistemi virtuali possono essere considerati, e talvolta vengono definiti, "macchine" gestite.

Ci permette quindi la creazione automatica di host, l'installazione di Docker Engine su di essi e la configurazione di Docker. Questa soluzione supporta anche vari fornitori di servizi cloud come AWS, Azure, DigitalOcean, Google Compute Engine, OpenStack ecc.

Come visto in precedenza, Docker Swarm è invece un gruppo di host dotati dell'engine Docker collegati tra loro in un cluster. Il cluster *swarm* è gestito da uno *swarm manager* e da un insieme di *worker*. Con Docker Swarm, puoi distribuire e ridimensionare le tue applicazioni su più host. Swarm aiuta con la gestione, il ridimensionamento, il networking, l'individuazione dei servizi e il bilanciamento del carico tra i nodi del cluster.

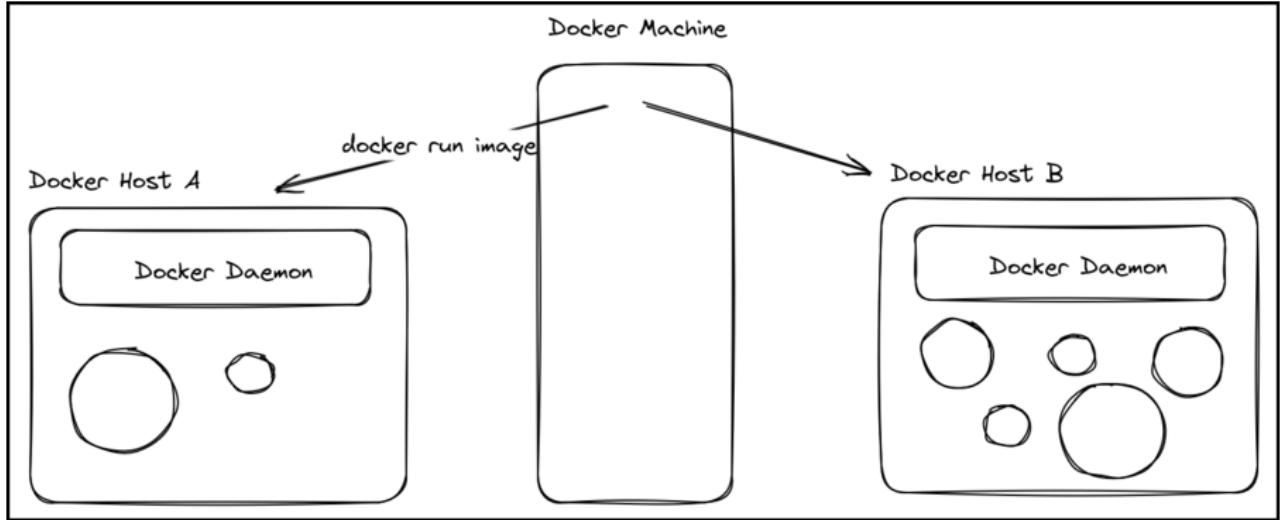


Figura 2.8 Rappresentazione del funzionamento di Docker Machine.

Kubernetes

Abbiamo già detto che Kubernetes è un framework CaaS open source creato dagli sviluppatori di Google più di dieci anni fa. Fondamentalmente, Kubernetes è un sistema di containerizzazione portatile e open source che consente agli sviluppatori di gestire servizi e carichi di lavoro. Questo sistema automatizza il rilascio, il ridimensionamento e le operazioni delle applicazioni.

Ora parte della *Cloud Native Computing Foundation*, Kubernetes consente agli sviluppatori di applicazioni di sfruttare funzionalità come il monitoraggio, l'automazione dei processi, il bilanciamento dei container, l'orchestrazione dello storage e altro ancora. Kubernetes può essere configurato e installato su soluzioni on-premise o sfruttando uno dei provider già citati in precedenza: se, per esempio, scegliessi di gestire tramite AWS in autonomia la configurazione e la gestione di Kubernetes, potresti creare delle istanze EC2 (simili alle macchine virtuali) dove installare tutto ciò che ti occorre per eseguirlo. Potresti anche decidere di lasciare la responsabilità di configurare Kubernetes al provider e occuparti solo di avviarlo ed eseguirlo, tramite un servizio chiamato “EKS” (acronimo di *Amazon Elastic Kubernetes Service*). Le soluzioni sono potenzialmente infinite!

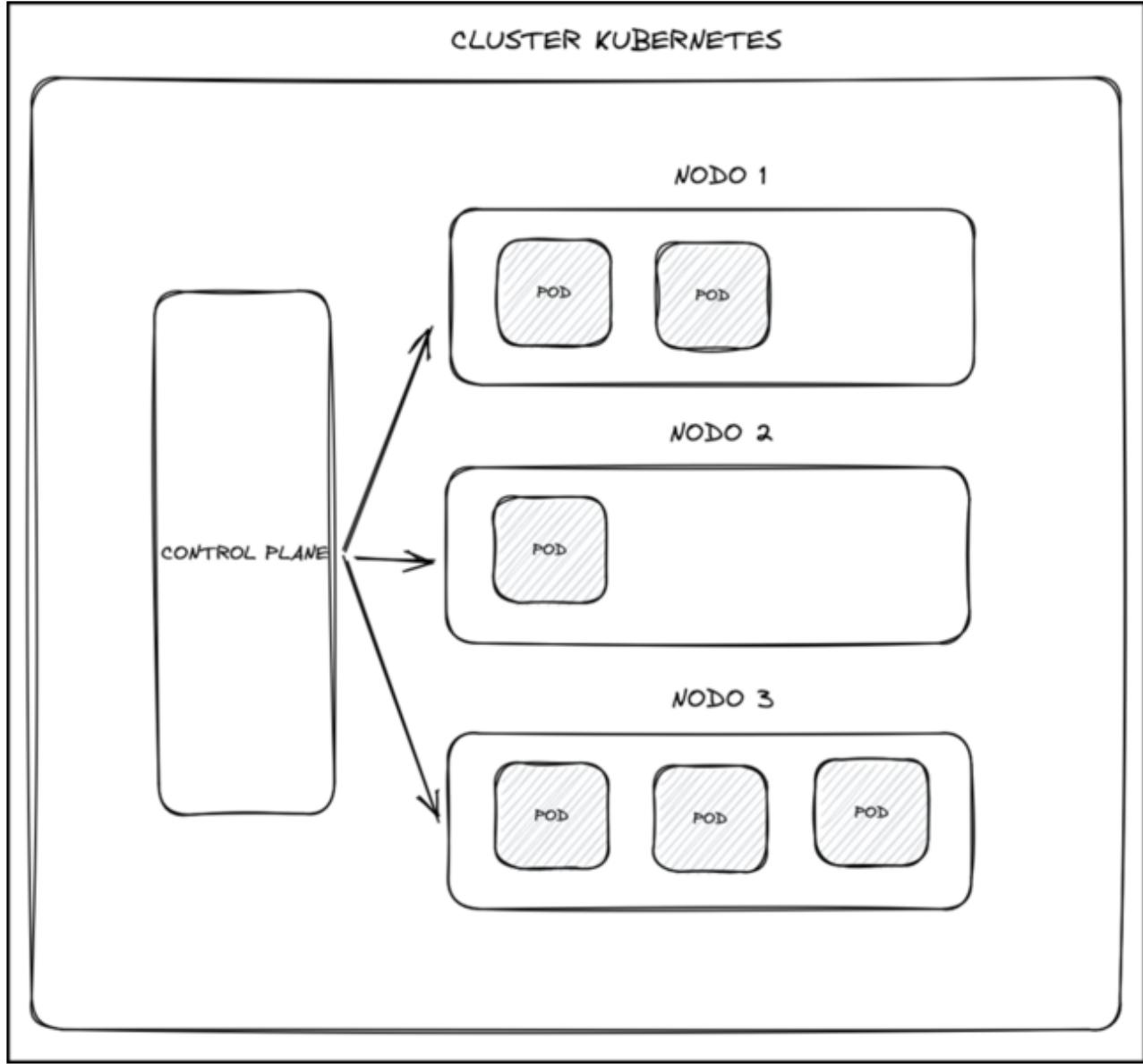


Figura 2.9 Architettura generica di esempio di un cluster Kubernetes.

OpenShift

OpenShift è una piattaforma basata su container (e su Kubernetes) che permette la gestione delle distribuzioni gli applicativi su cloud ibridi e multi-cloud. Rispetto ad altri prodotti che servono a gestire i container come orchestratori, OpenShift permette di definire un workflow complesso che, a partire dal codice sorgente dell'applicazione, può rendere automatizzato e più semplice il processo di deploy e di gestione delle applicazioni sulla propria piattaforma. OpenShift è stato costruito a partire da Kubernetes, ma sfruttando altri progetti oltre che tecnologie open source come il *Source-to-image*, ossia una tecnologia che permette di passare il codice sorgente di un'applicazione al sistema e generare una nuova immagine, o anche Jenkins, ossia una tecnologia che si occupa di *continuous integration* e *continuous delivery*.

CI/CD: di che parliamo?

Se vuoi addentrarti un po' di più nel mondo DevOps e scoprire che cosa rappresentano nel dettaglio tecniche come l'integrazione continua, puoi leggere l'ottimo volume scritto da Fabio Mora chiamato *DevOps. Guida per integrare Development e Operations e produrre software di qualità*, edito da Apogeo Editore.

Kubernetes e OpenShift sono due soluzioni molto simili nel loro utilizzo, anche se differiscono per una serie di motivi. Il primo può essere adatto per quasi tutti i progetti; il software è infatti open source e abbiamo visto che è possibile sfruttare moltissime distribuzioni diverse anche tramite provider esterni. Al contrario, OpenShift rappresenta una soluzione *enterprise* e si tratta di un prodotto disponibile sia in versione community (con le dovute limitazioni e supporto), sia a pagamento. Oltre a questo, essendo un prodotto autonomo e configurato dagli esperti di Red Hat, si ha come vantaggio un elevato standard di sicurezza, marchio di qualità di tutti i prodotti Red Hat.

Tra i vantaggi nell'uso di questo sistema c'è la possibilità di una completa automazione di una serie di operazioni che normalmente verrebbero gestite da un team, come la gestione del CI/CD e del registry; la gestione dei progetti e di tutti gli elementi che lo compongono diventa infatti più semplice grazie a una dashboard che evita per la maggior parte delle operazioni la necessità di ricorrere alla riga di comando, garantendo all'utente la possibilità di leggere i log dei Pod, piuttosto che gestirne le risorse in termini di memoria e CPU tramite una semplice console Web. Fondamentalmente, OpenShift è una piattaforma container Kubernetes basata su cloud che è considerata sia un software di containerizzazione che una piattaforma come servizio (PaaS).

OpenShift offre *security-by-design*, un sistema di monitoraggio integrato, gestione centralizzata delle policy di autenticazione e autorizzazione e compatibilità con i flussi e le risorse di lavoro Kubernetes. È veloce, consente il provisioning self-service e si integra con una varietà di strumenti tra i più utilizzati dalle grandi aziende per il monitoraggio e l'automazione dei flussi di lavoro in ambienti di produzione. Precedentemente nota come Origin, questa piattaforma open source consente agli sviluppatori di creare, testare e distribuire applicazioni sul cloud e supporta anche diversi linguaggi di programmazione, inclusi Go, Node.js, Ruby, Python, PHP, Perl e Java.

Kubernetes vs OpenShift

Le somiglianze tra questi due sistemi sono nel motore che gira dentro OpenShift. Vediamo infatti alcune delle differenze tra queste due soluzioni. A partire dall'installazione. Kubernetes offre una grande flessibilità come framework e può essere installato su quasi tutte le piattaforme, come Microsoft Azure e AWS, nonché su qualsiasi distribuzione Linux, inclusi Ubuntu e Debian. OpenShift, d'altra parte, richiede Red Hat Enterprise Linux Atomic Host, Fedora o CentOS. Ciò restringe le opzioni per molte aziende, soprattutto se non stanno già utilizzando queste piattaforme. Attenzione, però: se hai voglia di provare OpenShift sul tuo PC, puoi sempre utilizzare OpenShift Local!

In termini di sicurezza, OpenShift ha politiche più rigorose. Per esempio, è vietato eseguire un container come utente *root*, e questo per aumentare la sicurezza delle applicazioni rilasciate al suo interno. Kubernetes non viene fornito con funzionalità di autenticazione o autorizzazione integrate, quindi le persone che sviluppano devono gestire le procedure di autenticazione manualmente.

A livello di gestione della rete, Kubernetes non dispone di una soluzione di rete *by design*, ma consente agli utenti di utilizzare plug-in di rete di terze parti mentre OpenShift, d'altra parte, ha la sua soluzione di rete pronta all'uso chiamata Open vSwitch, uno strumento che estende i rudimentali strumenti di rete tradizionali presenti in Linux che sono stati usati per anni per lavorare in ambienti virtualizzati e offre un modello molto più semplice e centralizzato per implementare VLAN, tunnel GRE, VXLAN, QoS traffic shaping di base, IPsec e molte altre funzionalità.

Kubernetes non ha un registry delle immagini integrato, sebbene ti consenta di estrarre immagini da un registry privato in modo da poter creare i tuoi Pod. OpenShift, d'altra parte, ha un registry di

immagini integrato e si sposa perfettamente con Docker Hub o altri servizi.

Ultimo, ma non ultimo, l'esperienza utente: gli utenti che desiderano sfruttare un'interfaccia grafica per lavorare con Kubernetes devono installare separatamente una dashboard che gli permetta di eseguire anche i comandi più semplici, mentre OpenShift, al contrario, dispone di una console Web intuitiva che include una pagina di accesso *one-touch*. La console offre un'interfaccia semplice basata su moduli, che consente agli utenti di aggiungere, eliminare e modificare risorse. Con questa sola funzionalità, OpenShift ha un netto vantaggio per l'utente che deve approcciarsi a questo tool e non conosce la riga di comando a fondo.

Come sempre, con queste righe non si vuole porre una soluzione al di sopra dell'altra. Le due sono adatte a esigenze diverse: dipende infatti dal livello di competenze, dall'investimento che si vuole fare in termini infrastrutturali, dalle esigenze di personalizzazione e gestione dell'infrastruttura e così via. Scegliere una soluzione o l'altra non è banale!

Che cosa abbiamo imparato

- All'interno del mondo delle infrastrutture, abbiamo visto quali modelli abbiamo a disposizione e come questi lavorino nel contesto dei container.
- Attraverso una breve panoramica, abbiamo visto quali sono le differenze tra i diversi strumenti che hanno alla base il concetto di container, come Docker Swarm o Kubernetes, e perché una soluzione può adattarsi a un'esigenza più che un'altra, essendo poi in grado di scegliere il prodotto che più si addice alla situazione.

Architettura

Penso che sia molto importante coinvolgere più donne nell'informatica. Il mio slogan è: l'informatica è troppo importante per essere lasciata agli uomini.

– Karen Spärck Jones, ingegnere informatico e creatrice della funzione tf-idf, una tecnologia che è alla base dei più moderni motori di ricerca

Per poter comprendere un prodotto così pieno di sfaccettature, è necessario partire dalle basi con cui è stato costruito: la sua infrastruttura, benché possa sembrare complessa e articolata, è invece un meccanismo perfetto all'interno del quale si incassa ogni componente con la relativa funzionalità, che permette all'utente di raggiungere il proprio scopo: orchestrare i propri container in maniera trasparente, rendendo anche più semplice la loro permanenza nel cluster.

Definizione generale

In questo capitolo vedremo molti concetti che ci aiuteranno a muoverci nel mondo di Kubernetes, alcuni dei quali saranno fondamentali per le sezioni successive. Per esempio, i concetti di seguito riportati rappresentano i mattoni su cui si basa Kubernetes e che ne costituiscono l'architettura di base. Per capire come Kubernetes è in grado di fornire le sue funzionalità, è utile avere un'idea di come è organizzato ad alto livello e di quali componenti è costituito. Il suo disegno può essere composto in diversi modi, quindi cercheremo di trovare quello più funzionale, per poterne analizzare ogni singolo livello, e partiamo da un primo concetto: Kubernetes è un sistema *distribuito*. Questo significa che ci sono diversi componenti che sono “sparsi” su diversi server o macchine all'interno di una o più reti. Questi server possono essere macchine virtuali o anche server fisici, e insieme compongono un *cluster*.

Il cluster, in questo dominio, consiste di diversi sistemi, chiamati nodi *control-plane* e nodi di lavoro, o *worker*. Questi nodi sono di due tipologie e si chiamavano in passato rispettivamente *master* e *worker* o *slave* ma, per una questione legata all'utilizzo di un linguaggio più inclusivo (issue aperta sul repository di Kubernetes sul tema: <https://github.com/kubernetes-sigs/kubespray/issues/7157>), questi nomi non sono più in uso. Questi variano in numero in base al tipo di disponibilità che vogliamo dare all'infrastruttura, ma di base c'è sempre almeno un nodo *control-plane* e un nodo *worker*, dove il primo ha un ruolo primario; nella Figura 3.1 vediamo in che modo questi sono correlati. A grandi linee, il *control-plane* è infatti un tipo di nodo che ha la responsabilità di orchestrare i container e mantenerli nello stato desiderato all'interno del cluster.

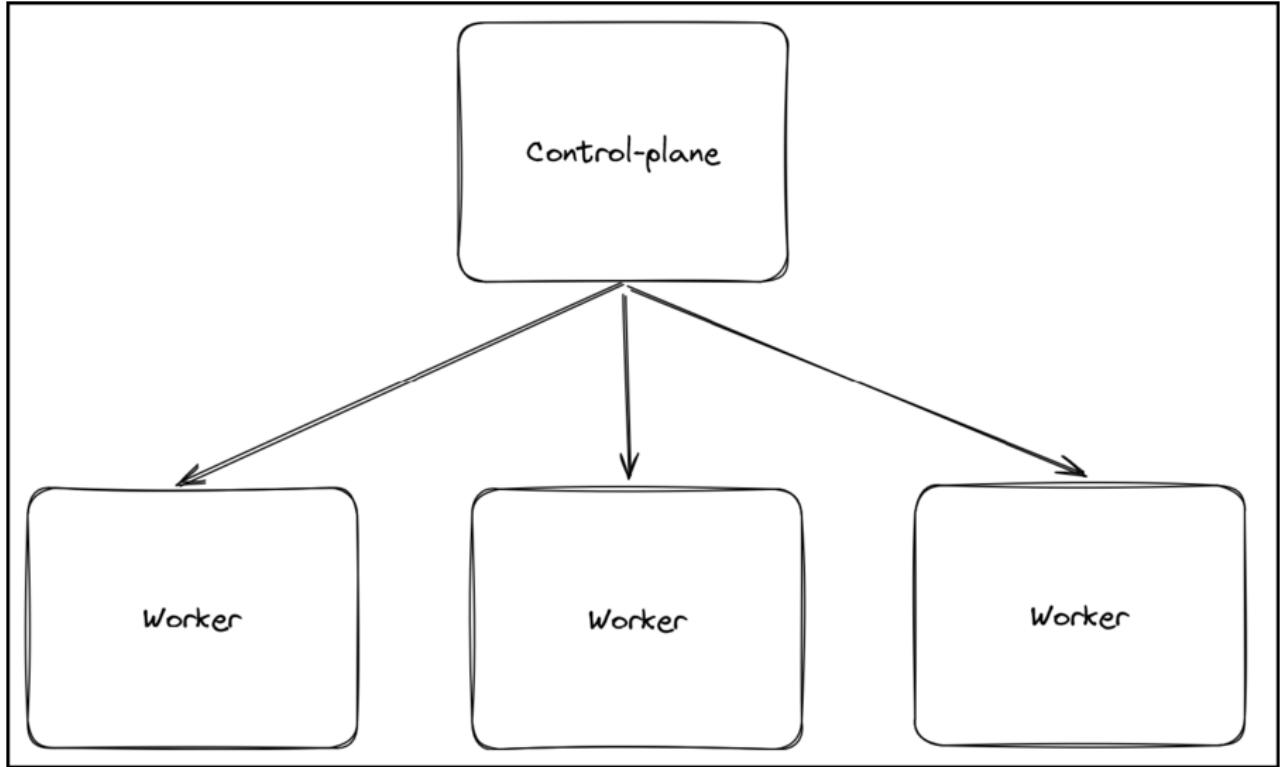


Figura 3.1 Esempio di cluster in Kubernetes, composto da un nodo control-plane e 3 worker.

Al suo interno, sono presenti diversi componenti, come il `kube-apiserver`, `etcd`, `kube-scheduler`, `kube-controller-manager` e `cloud-controller-manager`, e ognuno di essi ha un ruolo ben preciso (che vedremo a breve). Un nodo di tipo worker è invece considerato più applicativo, dal momento che è responsabile dell'esecuzione delle applicazioni containerizzate; ognuno di essi ospita `kubelet`, `kube-proxy` e l'ambiente di esecuzione dei container.

Ma che cosa fa ognuno di questi componenti? Vediamoli nel dettaglio.

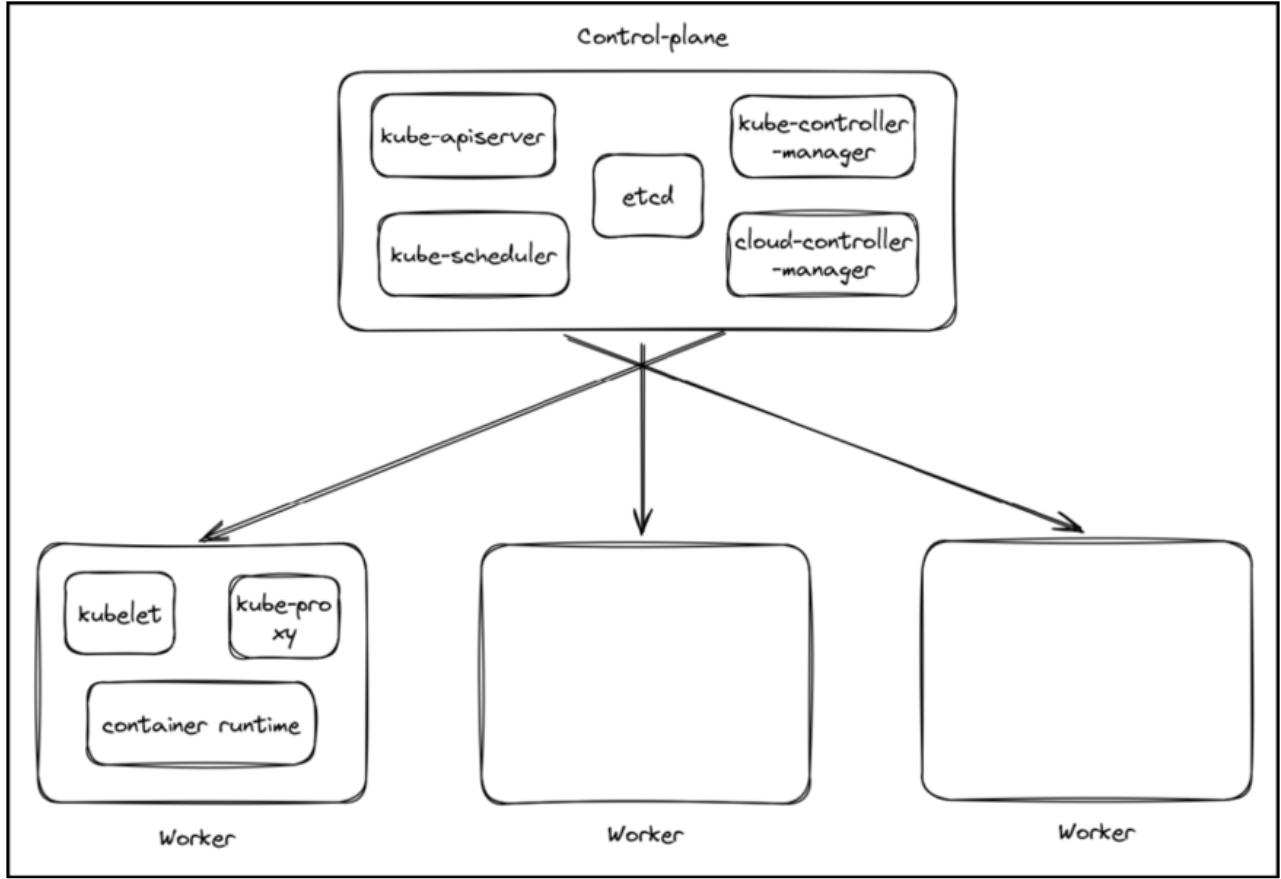


Figura 3.2 Architettura del cluster di esempio precedente, ma con i componenti presenti nei nodi.

Componenti del control-plane

Nel complesso, i componenti che girano su questi nodi 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 worker, configurare la rete a livello di cluster e gestire il ridimensionamento del sistema.

Etcdb

Uno dei componenti fondamentali che Kubernetes ha a disposizione è la gestione della propria configurazione in modo che sia accessibile a livello globale. Il progetto etcd, sviluppato dal team di CoreOS, è un sistema open source che lavora come 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 a cui è possibile accedere da ciascuno dei nodi nel cluster; dal momento che abbiamo parlato di sistema distribuito, era fondamentale avere un database distribuito che supportasse la persistenza di una configurazione condivisa, garantendo ai diversi componenti di mantenere traccia delle informazioni relative al loro stato e agli eventuali aggiornamenti delle informazioni a disposizione. In altre parole, lavora come fosse il “cervello” di Kubernetes, e anche come la sua esperienza.

Quando parliamo di “persistere lo stato della configurazione”, facciamo riferimento a ciò che il cluster distribuisce, come se potessimo fare una fotografia della situazione attuale memorizzandola in un database. Se viene eseguito un aggiornamento su un nodo, come per esempio una nuova applicazione

che viene eseguita, questo cambio di configurazione nell'architettura viene immediatamente riportata nel database di etcd, ancor prima che questo avvenga; in effetti, è proprio etcd a dare il via a qualsiasi cambiamento nel cluster, dal momento che si occupa di confermarne lo stato desiderato e l'avvio ai lavori soltanto quando le informazioni saranno persistite al suo interno. Oltre a questo, etcd aiuta anche a mantenere lo stato del cluster tramite funzionalità come l'elezione del leader, che sfrutta l'algoritmo di consenso Raft; questo consente di mantenere alta la disponibilità del cluster anche nel caso uno o più nodi falliscano. Quindi, come funziona etcd con Kubernetes? Per farla semplice, quando useremo il comando `kubectl` per ottenere i dettagli di una delle tante risorse che questa tecnologia ci mette a disposizione, questi dati vengono recuperati proprio da etcd. Questo memorizza tutte le configurazioni, gli stati e i metadati degli oggetti Kubernetes: che siano essi Pod, Secret, Daemonset o anche Deployment, ogni informazione relativa al loro stato viene riportata su questo database, così da poter "ricostruire" la situazione se ci sono guasti nei nodi worker. etcd memorizza tutti gli oggetti in una directory chiamata `/registry` tramite il formato valore-chiave. Per esempio, le informazioni su un Pod denominato Nginx all'interno nel namespace di default, saranno disponibili sotto la cartella `/registry/Pods/default/nginx`. Non c'è da preoccuparsi se termini come "Pod" o "namespace" al momento sono sconosciuti, ci arriveremo!

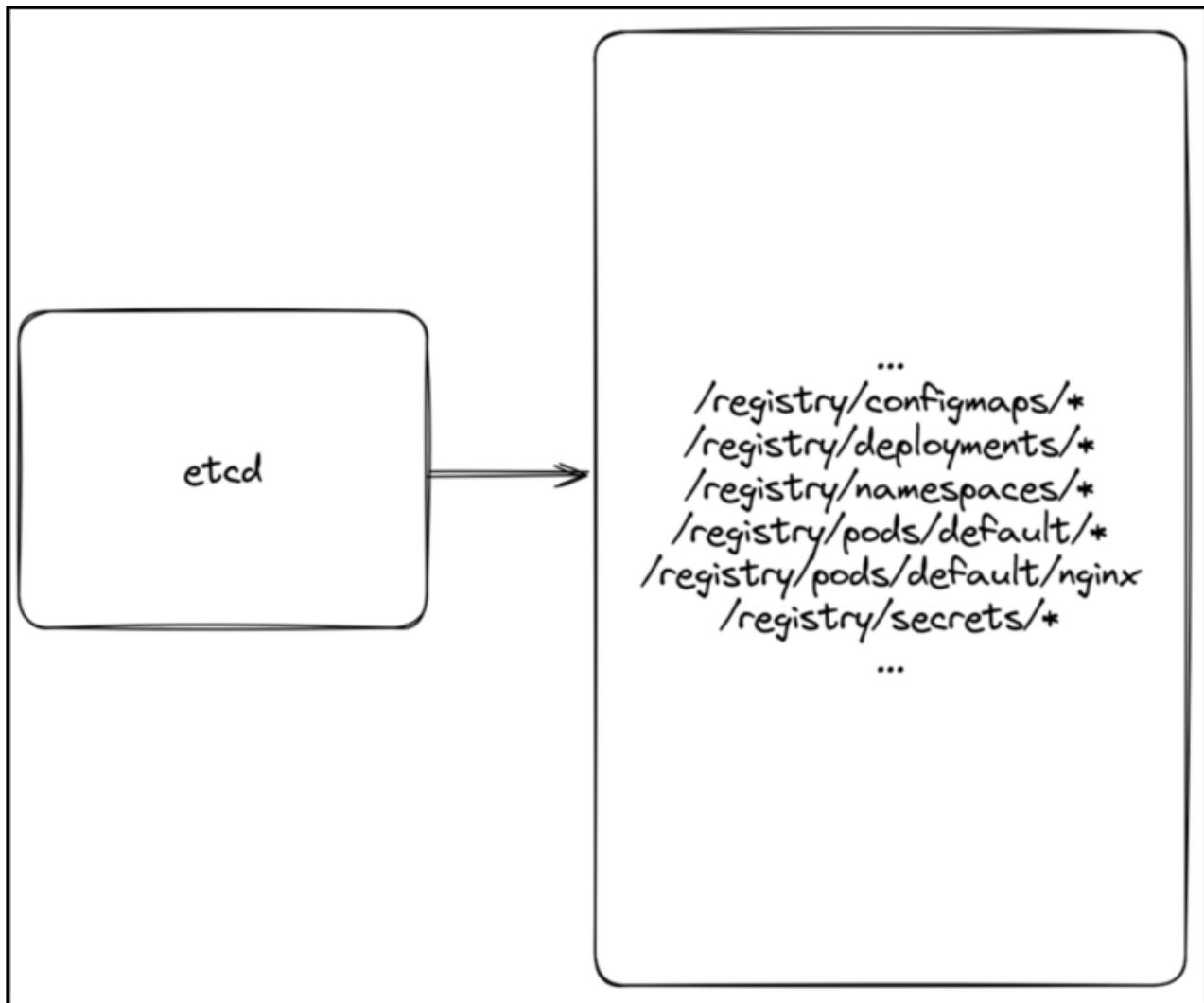


Figura 3.3 Esempio di valori contenuti nel database di etcd.

Pod, questo sconosciuto

Durante questo viaggio incontreremo molte parole nuove, ma non per questo bisogna scoraggiarsi: al momento, per poter procedere con il resto del capitolo, ti basti immaginare che un Pod è un oggetto che “ingloba” uno o più container e che, in Kubernetes, rappresenta un singolo servizio, o applicazione.

API Server

Questo componente è l’hub centrale del cluster Kubernetes che espone le API per comunicare con Kubernetes. Sia gli utenti che gli altri componenti del cluster comunicano con esso tramite questo server; pertanto, quando utilizzeremo il comando `kubectl` per gestire il cluster, dietro alle quinte staremo effettivamente comunicando con il server API tramite delle chiamate REST HTTP. Non tutti usano però questo meccanismo: alcuni componenti interni del cluster come lo *scheduler*, il *controller* e così via comunicano con il server API utilizzando gRPC, un sistema che si occupa di fornire un processo di gestione delle chiamate per collegare tra loro più componenti. In altre parole, gRPC semplifica di molto la comunicazione tra due o più oggetti, fornendo un sistema di messaggistica tra componenti che possono comunicare in maniera sicura. La comunicazione tra il server API e altri componenti nel cluster avviene comunque tramite TLS per impedire l’accesso non autorizzato al cluster.

Questo server è dunque responsabile di gestire le API ed esporre un endpoint come punto di accesso per gestire le richieste in ingresso, di autenticarle tramite certificati o token, di elaborarle e convalidarne l’output, e lo fa comunicando direttamente con etcd. Si occupa dunque di coordinare tutti i processi tra i nodi control-plane e quelli worker.

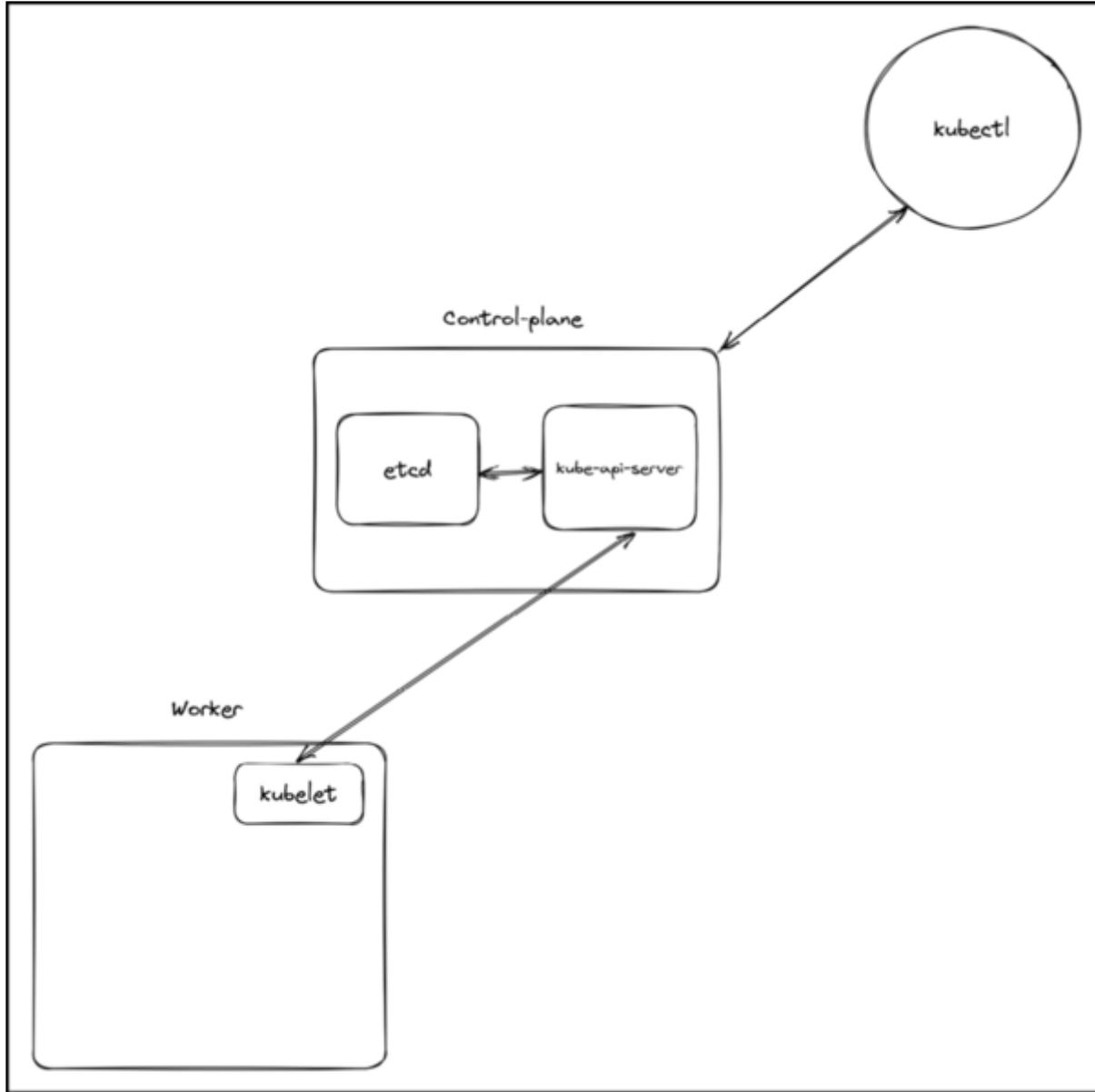


Figura 3.4 Funzionamento del componente kube-api-server.

kube-scheduler

Questo elemento è responsabile della pianificazione dei Pod sui nodi worker: questo vuol dire che si occuperà di gestire le richieste per la creazione di nuove risorse applicative ospitate all'interno di container all'interno di uno dei nodi del cluster, secondo alcune specifiche. In effetti, quando vorremo avviare un'applicazione (genericamente parlando), andremo a specificare i requisiti in termini di CPU, memoria, persistenza e altre informazioni; tutti questi dati serviranno allo `scheduler` per identificare il nodo migliore per ospitare il servizio applicativo e quindi richiederne la creazione, se questo soddisfa i requisiti. Questo vuol dire anche che deve conoscere la capacità totale dei nodi e le risorse già allocate ai carichi di lavoro esistenti su ciascun server; queste informazioni può ottenerle grazie alla comunicazione con il server che mette a disposizione le API relative al cluster, ossia `kube-api`.

Prima di parlare però di come effettivamente lo scheduler sia in grado di decidere dove e quando eseguire un Pod, parliamo dei restanti componenti.

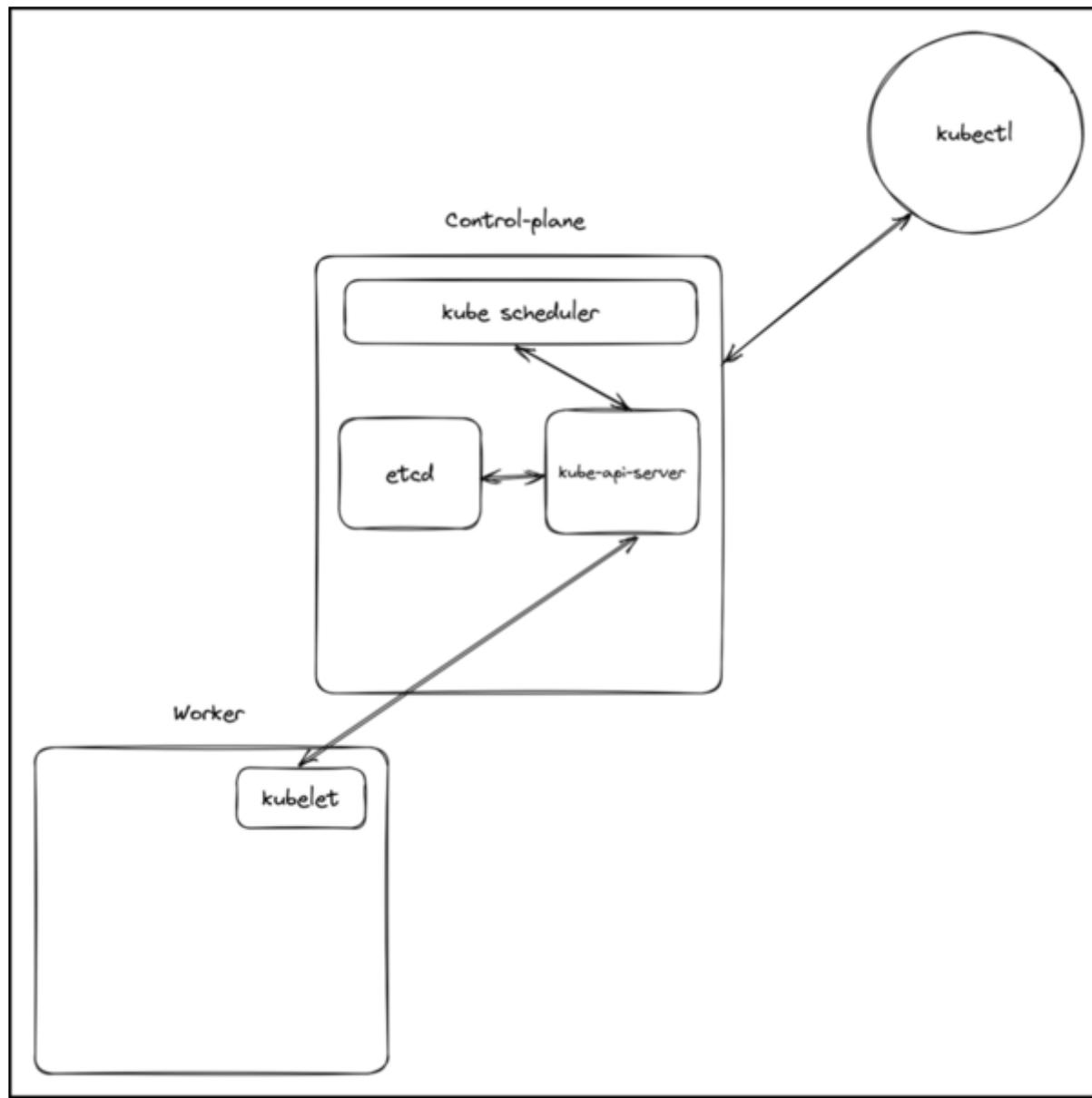


Figura 3.5 Funzionamento del componente kube-scheduler.

Kube controller manager

Come indica il nome, un controller è un oggetto che esegue un loop infinito per verificare delle condizioni; questo significa che è costantemente alla ricerca di un cambiamento dello stato del cluster e ne controlla lo stato reale con quello desiderato. Se c'è una differenza tra lo stato effettivo e quello desiderato, questo componente garantisce che la risorsa/oggetto Kubernetes si trovi nello stato desiderato. Secondo la documentazione ufficiale (riferimento originale in inglese):

<https://kubernetes.io/docs/concepts/architecture/controller/>): “in Kubernetes, i controller sono loop di controllo che osservano lo stato del tuo cluster, quindi apportano o richiedono modifiche dove necessario. Ogni controller tenta di riportare lo stato corrente del cluster più vicino possibile allo stato desiderato”. Immagina dunque di voler creare un Pod, al cui interno abbiamo un container che ospita la

nostra applicazione in Python: questa risorsa si occuperà di verificare che, una volta che lo scheduler avrà trovato per l'oggetto una collocazione, sia effettivamente presente un Pod con quelle caratteristiche. Il Kube controller manager è un componente che gestisce tutti i controller Kubernetes: come vedremo nei prossimi capitoli, le risorse e gli oggetti Kubernetes come Pod, namespaces, Jobs e molti altri sono gestiti dai rispettivi controller; e indovinate chi gestisce lo scheduler? Proprio il Kube controller manager.

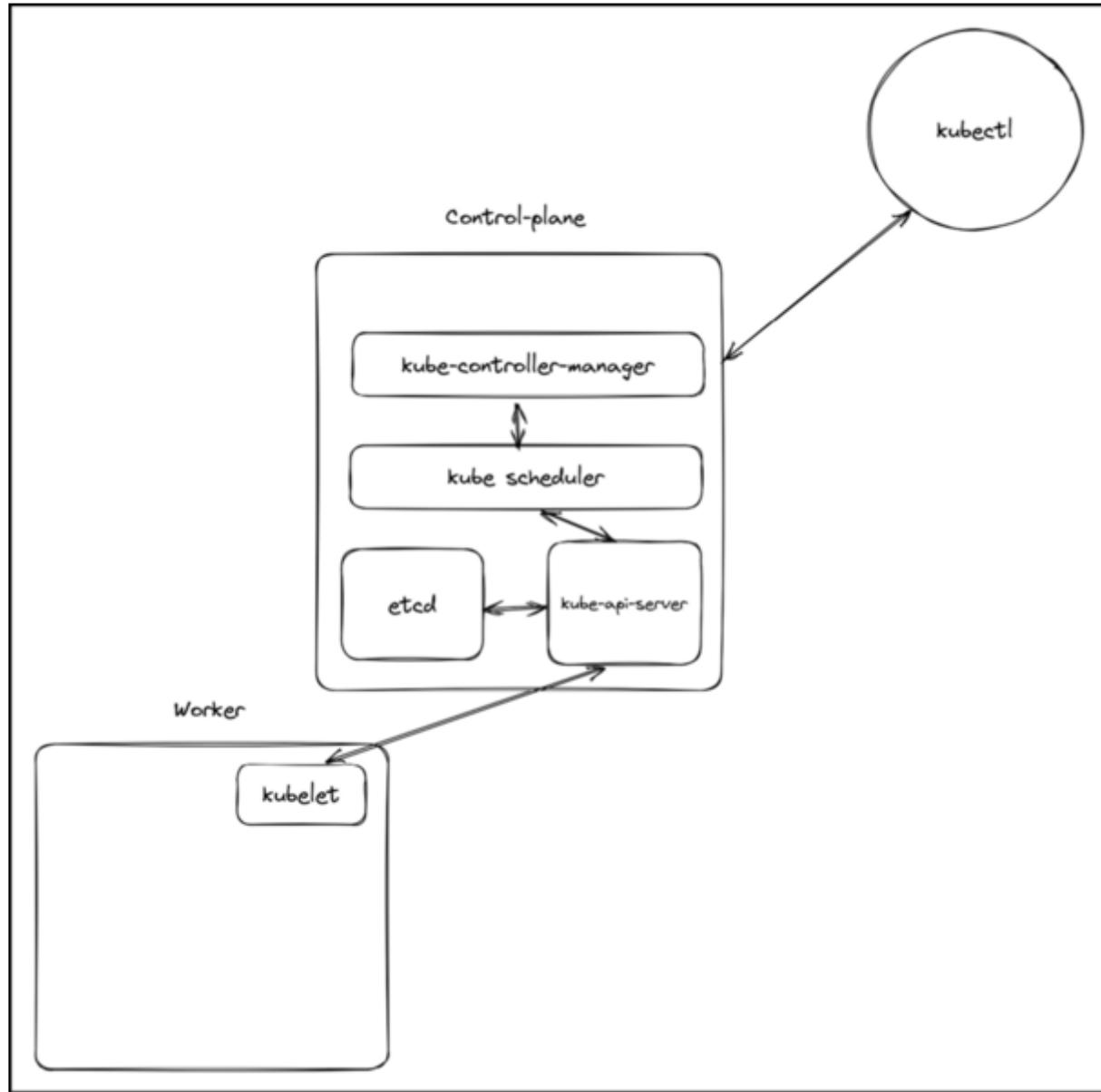


Figura 3.6 Funzionamento del componente kube-controller-manager.

Cloud Controller Manager

Ultimo, ma non per importanza, il CCM: quando Kubernetes viene distribuito in ambienti cloud, questo controller funge da ponte tra le API della piattaforma utilizzata e il cluster. Kubernetes può essere distribuito in molti ambienti diversi e può interagire con vari provider per comprendere e gestire lo stato delle risorse nel cluster. In questo modo i componenti principali di Kubernetes possono funzionare in modo indipendente e consentire ai fornitori di servizi cloud di integrarsi con Kubernetes.

utilizzando i relativi plugin. L'integrazione del controller consente al cluster Kubernetes di eseguire il provisioning di risorse cloud come istanze (per aggiungere dei nodi), Load Balancer (per la comunicazione di rete) e volumi di archiviazione (per i volumi persistenti) in maniera trasparente. Questo infatti è stato originariamente creato per consentire di sviluppare Kubernetes indipendentemente dall'implementazione dello specifico cloud provider; in altre parole, funge da collante per consentire a Kubernetes di interagire con i provider e le relative funzionalità, mantenendo al contempo costrutti relativamente generici al proprio interno.

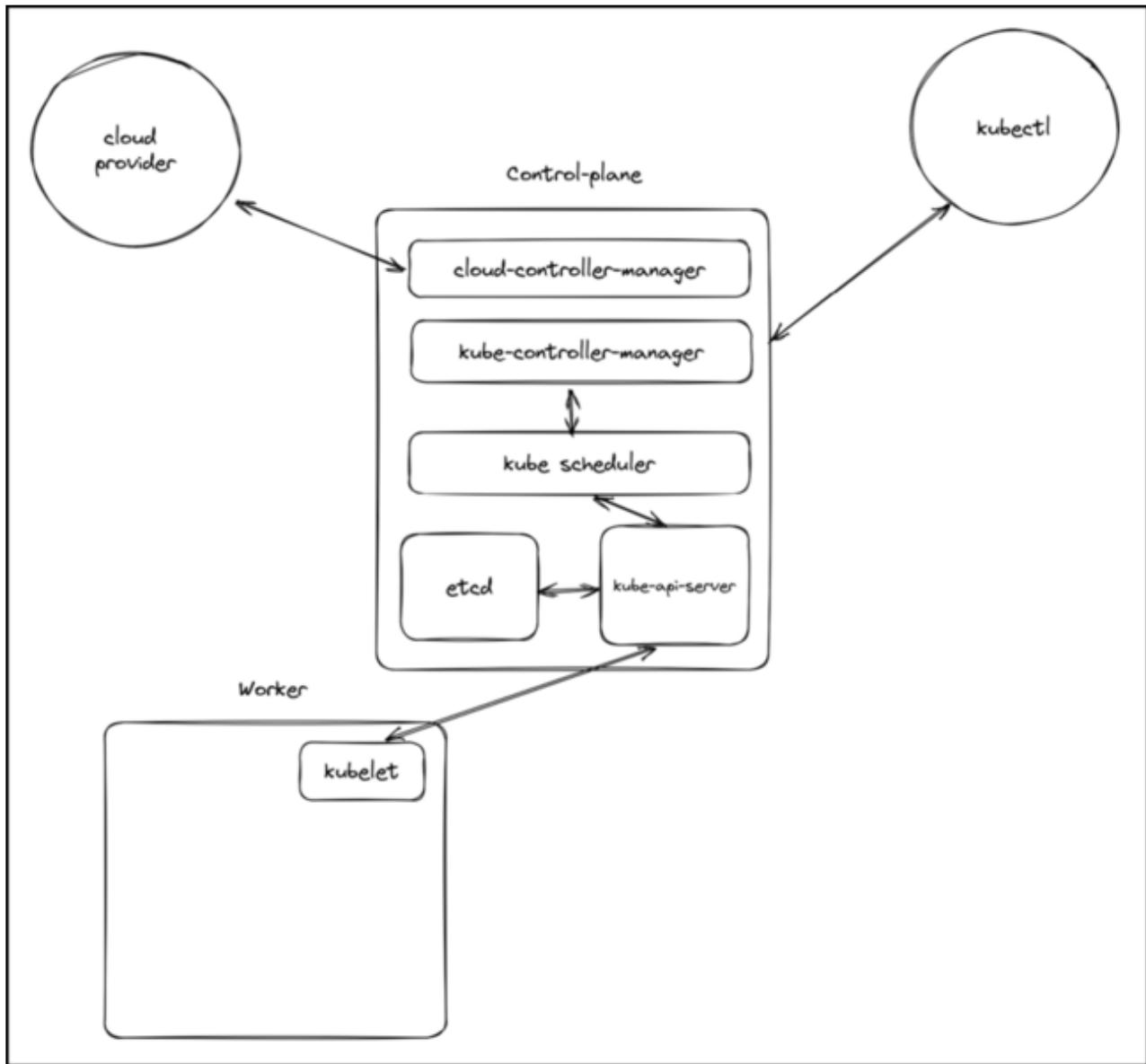


Figura 3.7 Funzionamento del componente cloud-controller-manager.

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. Il CCM ci tornerà particolarmente utile quando parleremo di Kubernetes on cloud, verso la fine del manuale, e di come questo possa essere utilizzato attraverso diversi cloud provider.

Componenti del worker

Ora che abbiamo una prima panoramica delle risorse che gestiscono attivamente il cluster, parliamo dei componenti che si occupano degli applicativi, e che risiedono nei nodi worker. Questi sono infatti fondamentali per far sì che l'applicazione funzioni correttamente e sia fruibile anche da un utente esterno!

kubelet

Il punto di contatto principale per ciascun nodo con il gruppo cluster è un piccolo servizio chiamato `kubelet`. `kubelet` è un agente che viene eseguito su ogni nodo worker del cluster come demone, gestito da `systemd`. Per dirla semplicemente, è responsabile di quanto segue: creazione, modifica ed eliminazione dei container per il Pod, gestione del loro avvio e della loro corretta esecuzione, dei relativi volumi e della raccolta dei dati dello stato del nodo e del Pod. `kubelet` è anche un controller che verifica le modifiche relative ai Pod e utilizza l'ambiente di esecuzione dei container del nodo per effettuare il pull delle immagini e tutte le attività che riguardano l'applicazione.

Come accennato in precedenza, `kubelet` comunica con i nodi control-plane per autenticarsi con il cluster e ricevere le istruzioni per poter operare; la definizione delle risorse con cui deve operare avviene per mezzo di un file in formato YAML, che viene anche definito *manifest*, il quale definisce l'oggetto e i parametri operativi di ciò che deve creare e/o gestire. Il processo `kubelet` si assume quindi la responsabilità di mantenere lo stato delle risorse coerente a quanto descritto nel database di `etcd`.

Container Runtime

Se hai mai sentito parlare di JRE, o Java Runtime, allora avrai un'idea di che cosa rappresenti questo componente: nel caso di Java si tratta del software richiesto per eseguire programmi scritti tramite questo linguaggio di programmazione su un qualsiasi host. Allo stesso modo, è necessario un ambiente di esecuzione che, tramite del software, sia in grado di gestire e avviare dei container. In effetti, il primo componente che ogni nodo worker deve avere è un *container runtime*.

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; viene eseguito su tutti i nodi nel cluster Kubernetes ed è responsabile del pull delle immagini dai registry come DockerHub o simili, dell'esecuzione dei container, dell'allocazione e dell'isolamento delle risorse per i container e della gestione dell'intero ciclo di vita di un container su un host. In questo dominio, esistono due concetti chiave.

- *Container Runtime Interface (CRI)* è quel componente che rappresenta un insieme di API per consentire a Kubernetes di interagire con diversi ambienti di esecuzione per container. In questo modo, è possibile utilizzare in modo intercambiabile diversi strumenti per container con Kubernetes, visto che definisce un'API per la creazione, l'avvio, l'arresto e l'eliminazione di container, nonché per la gestione di immagini e la loro comunicazione. Per fare un esempio, Docker Engine è un container runtime che rispetta gli standard definiti all'interno di CRI.
- *Open Container Initiative (OCI)* è invece un insieme di standard per i formati e i tempi di esecuzione dei container.

Kubernetes supporta più container runtime, e ciò significa che tutti questi implementano l'interfaccia CRI ed espongono le API per l'esecuzione e la gestione delle immagini.

Open Container Initiative

L'OCI è un progetto formatosi grazie alla Linux Foundation per definire uno standard per tutte le risorse che gravitano attorno al mondo dei container. L'OCI è stato lanciato il 22 giugno 2015 da Docker, CoreOS e altri leader nel settore dei container.

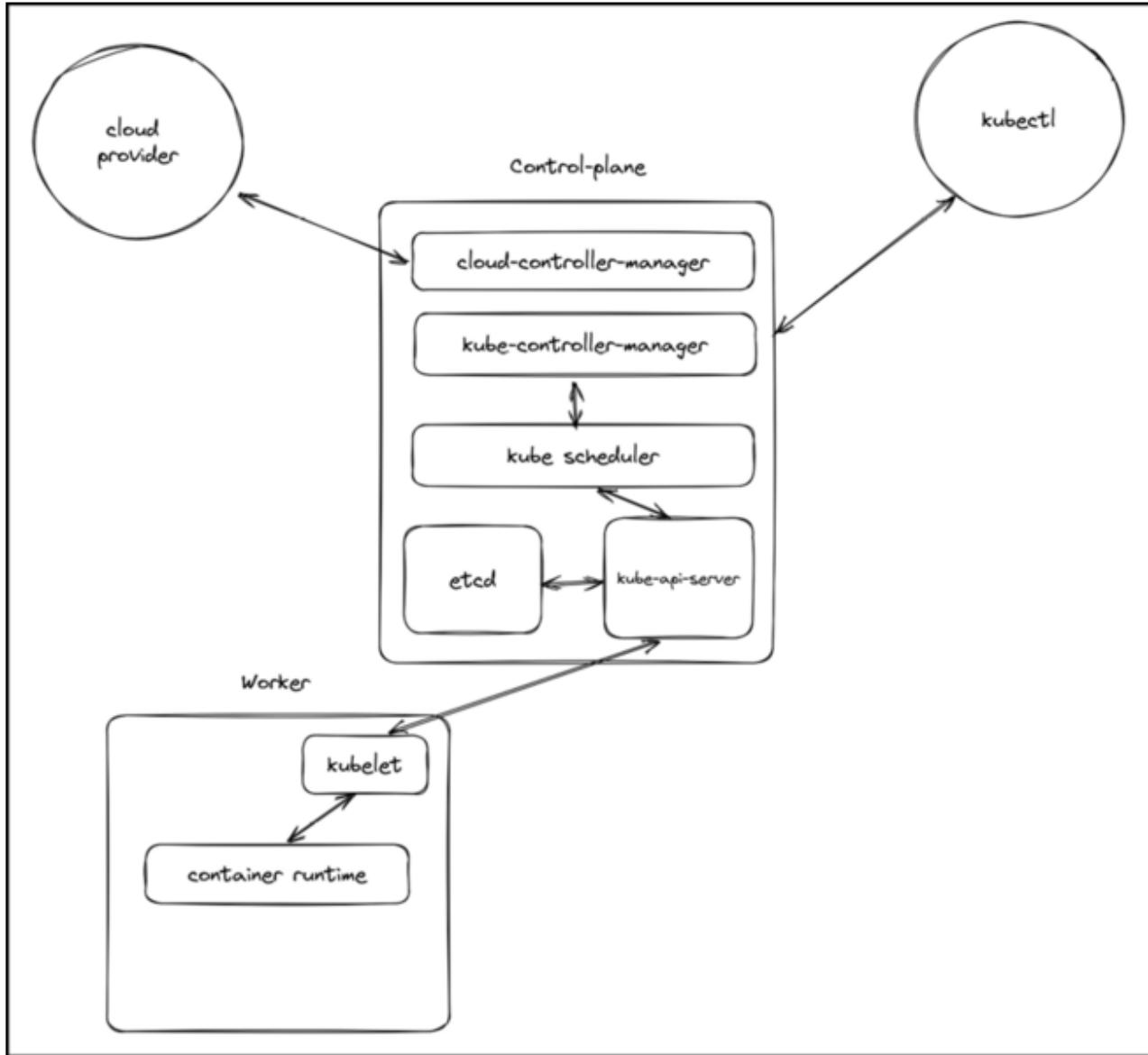


Figura 3.8 Contesto di funzionamento di kubelet all'interno dell'architettura del cluster.

kube-proxy

Infine, abbiamo questo elemento: ricordando che i container lavorano e operano come fossero tanti piccoli host separati e isolati gli uni dagli altri, in un contesto dove la complessità aumenta, è necessario gestire la rete di questi singoli host per rendere disponibili dei servizi ad altri componenti; quindi, su ciascun nodo worker, viene eseguito un piccolo servizio proxy chiamato `kube-proxy`. Questo processo inoltra le richieste ai container giusti, può eseguire il bilanciamento del carico delle richieste di base ed è generalmente responsabile di assicurarsi che la rete sia correttamente configurata e

accessibile, ma anche isolata se necessario. `kube-proxy` è infatti un demone (come `kubelet`) che implementa il concetto di *Services* per i Pod, concetto che vedremo più in là nel manuale.

Oltre ai componenti principali descritti finora, il cluster Kubernetes necessita di componenti aggiuntivi per essere pienamente operativo, e la scelta di un componente piuttosto che un altro dipende dai requisiti del progetto e dai casi d'uso. Proviamo quindi a riassumere molto velocemente quali sono quelli principali e perché potresti incontrarli.

CoreDNS

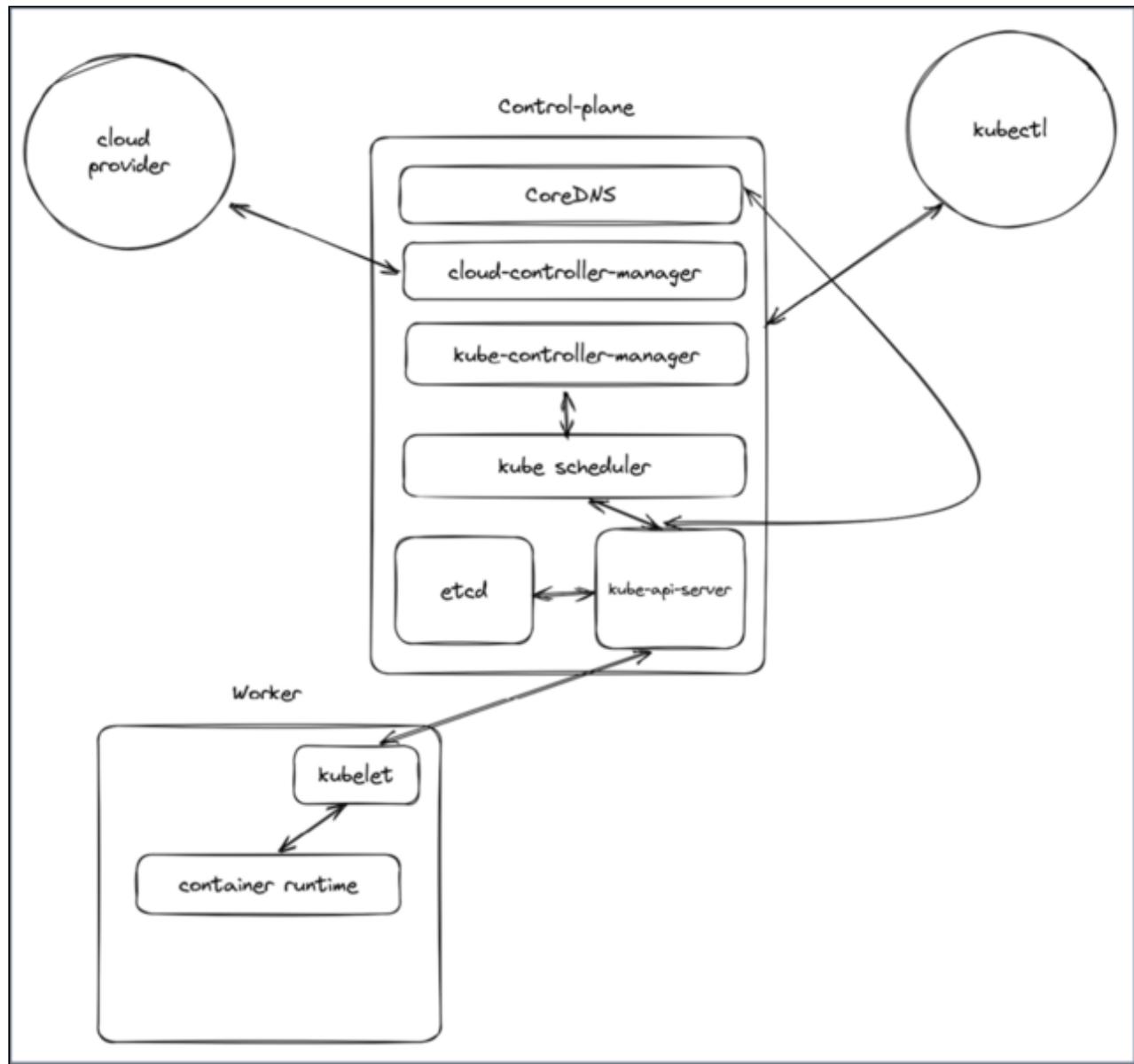


Figura 3.9 Contesto di funzionamento di CoreDNS all'interno dell'architettura del cluster.

Il DNS è una delle funzionalità principali di Kubernetes, dal momento che ci permetterà di lavorare con oggetti come i Services per esporre verso l'esterno le nostre applicazioni; kube-dns o CoreDNS come estensione indispensabile per il cluster K8s. Quando il Pod dovrà comunicare con altre risorse, all'interno e all'esterno del cluster, dovrà sapere dove inviare la richiesta. Con il sistema DNS, i Services in Kubernetes potranno essere referenziati per nome che corrisponderà ai Pod gestiti dal

Service stesso, utilizzando un nome di dominio completo (chiamato anche FQDN, ossia *Fully Qualified Domain Name*), al posto dell'indirizzo IP. CoreDNS può essere utilizzato per fornire un servizio di questo tipo ed è parte di Kubernetes dalla versione 1.13, al posto del precedente kube-dns, che è stato dismesso per avere una maggior efficienza e una minor occupazione delle risorse. Questo componente, essendo di vitale importanza, gira sui nodi *control-plane*.

CNI Plugin

Quando si tratta di reti di container, le aziende potrebbero avere requisiti diversi come l'isolamento della rete, la sicurezza, la crittografia e via dicendo. Con l'avanzare della tecnologia dei container, molti provider di rete hanno creato soluzioni basate su CNI per container con un'ampia gamma di funzionalità di rete. Kubernetes 1.26 supporta i plugin CNI (acronimo di *Container Network Interface*) per la gestione del networking dei cluster e sono disponibili diversi plugin (sia open source che closed source) nell'ecosistema Kubernetes. Ciò consente agli utenti di scegliere una soluzione di rete che meglio si adatta alle loro esigenze da diversi fornitori, dal momento che è necessario un plugin CNI per implementare il modello di rete Kubernetes. Come funziona il plugin con Kubernetes? Il Kube-controller-manager è responsabile dell'assegnazione del CIDR (ossia l'allocazione dello spazio degli indirizzi IP, acronimo di *Classless Inter-Domain Routing*) degli indirizzi IP relativi a ciascun Pod per ogni nodo. Come vedremo, ogni Pod riceve un indirizzo IP univoco a partire dal CIDR del Pod. Kubelet interagisce con il container runtime per avviare il Pod e il plugin CRI che fa parte dell'ambiente di esecuzione del container interagisce con il plugin CNI per configurare la rete di Pod. Questo consente il collegamento in rete tra Pod distribuiti su nodi uguali o diversi utilizzando una rete sovrapposta. Alcuni plugin conosciuti includono Calico, Flannel, Weave Net, Cilium, Amazon VPC CNI, Azure CNI. Come avrai capito, il networking Kubernetes è un argomento importante e differisce in base alle piattaforme di hosting.

Che cosa abbiamo imparato

- L'architettura di Kubernetes è piuttosto articolata ed è costituita da tanti piccoli elementi che consentono l'avvio di applicazioni in maniera efficiente.
- Dopo una panoramica sui nodi che compongono un cluster e su quali sono le differenze principali tra essi, abbiamo analizzato gli strumenti a disposizione di ognuno di questi, come `kubelet` o `kube-proxy`, per far funzionare l'intero processo come un orologio.
- Abbiamo anche aggiunto alcuni componenti che rendono la nostra infrastruttura adattabile al provider utilizzato, soprattutto se lavoriamo su ambienti on cloud.

Installazione

La parola “femmina”, quando inserita dopo qualcosa, è sempre con una nota di sorpresa. COO donna, pilota donna, chirurgo donna - come se il genere implicasse sorpresa... Un giorno non ci saranno leader donne. Ci saranno solo leader.

– Sheryl Sandberg, COO di Facebook

Avere a disposizione un ambiente dove iniziare a lavorare con Kubernetes è fondamentale: in tutti i migliori manuali tecnici si richiede infatti di mettere in pratica quanto raccontato e riportato nelle diverse pagine. Questo non perché chi scrive vuole avere ragione, ma per permettere di memorizzare al meglio i concetti che vengono esposti man mano che si va avanti con la lettura.

Nel caso di Kubernetes, avere a disposizione un cluster dove poter testare i primi comandi può sembrare non semplice; come abbiamo detto in precedenza, questo strumento è utilizzato da piccole e grandi aziende per distribuire le applicazioni agli utenti finali e richiede delle risorse di calcolo non indifferenti. La configurazione di un ambiente Kubernetes locale come ambiente di sviluppo resta comunque l’opzione consigliata, indipendentemente dalla situazione, perché questa configurazione può creare un processo di rilascio dell’applicazione sicuro e agile.

Che cosa installo?

Esistono sul mercato diverse opzioni che ci permettono di utilizzare Kubernetes senza dover “faticare” troppo per la sua configurazione: fortunatamente, esistono infatti più piattaforme adatte a diversi scopi che è possibile provare per eseguire Kubernetes in locale e sono tutte open source e disponibili con la licenza Apache 2.0.

- *Docker Desktop*, ambiente già ampiamente adottato da chi lavora con Docker e che nelle nuove versioni permette di configurare anche il supporto a Kubernetes tutto in una sola installazione.
- *CodeReady Containers* (abbreviato in CRC o chiamato anche in precedenza *Red Hat OpenShift Local*) è in grado di gestire un cluster OpenShift 4.x locale ottimizzato per scopi di test e sviluppo, con un’interfaccia grafica piuttosto complessa, sempre ricordando che sotto al cofano gira Kubernetes;
- *Minikube*: nasce con l’obiettivo principale di essere lo strumento adatto per il rilascio di applicazioni su Kubernetes ed è anche quello utilizzato da più tempo; richiede l’installazione/configurazione di un sistema di virtualizzazione e di *kubectl*
- *Rancher Desktop* è un progetto open source che porta Kubernetes e la gestione dei container sul desktop; funziona su Windows, macOS e Linux ed è un’applicazione basata su Electron che racchiude i diversi elementi all’interno di un’unica applicazione coerente.

La scelta di utilizzo di un ambiente o l’altro dipende molto dalla familiarità che si ha con questi strumenti, ma anche dal sistema che si ha a disposizione: per chi viene dal mondo Docker, quasi sicuramente l’ultima opzione sarà quella più semplice da utilizzare, dal momento che non richiede una configurazione particolare e delle risorse relativamente limitate. Al contrario, una soluzione come la seconda riportata nell’elenco ha maggiori requisiti in termini di risorse di calcolo, il che potrebbe non

essere l'ideale per chi ha bisogno di un ambiente di sviluppo rapido, ma perfetto per chi vuole studiare Kubernetes per poi approcciarsi al mondo di OpenShift.

Per questo motivo, nel capitolo non elencheremo pro e contro di ciascuna soluzione ma si riporteranno i processi di installazione di alcune di queste piattaforme, lasciando a chi legge la possibilità di provare i diversi ambienti e trovare quello più adatto. In ogni caso, tutti gli esempi che vedremo nei prossimi capitoli richiederanno l'utilizzo della riga di comando più che l'interfaccia grafica di un prodotto specifico, motivo per cui la scelta su quale opzione adottare resta libera.

Kubernetes online

Se invece volessi "giocare" con Kubernetes senza installare nulla sul tuo PC, puoi utilizzare il sito <https://labs.play-with-k8s.com/>: questo ti permette di creare un'istanza temporanea del cluster su cui eseguire i tuoi test, semplicemente collegandoti tramite browser e seguendo le (pochissime) istruzioni fornite nella pagina principale.

Installazione tramite Docker Desktop

In questo capitolo andremo a sporcarci le mani iniziando a capire come funziona Kubernetes sotto il cofano con Docker Desktop: è disponibile gratuitamente in un'edizione community, per Windows e Mac. Inizia scaricando e installando la versione giusta dal sito ufficiale:

<https://hub.docker.com/editions/community/docker-ce-desktop-windows>.

Versione di Docker Desktop

Attualmente la versione più recente di Docker Desktop è la 4.15.0, ma i rilasci avvengono piuttosto frequentemente. Tutto ciò che vedremo nei prossimi capitoli è solitamente compatibile con le versioni che vanno dalla 4.3, con delle differenze che per il momento sono trascurabili.

Per poterne usufruire su Windows, è necessario avere almeno Windows 10 (professionale o aziendale), 4 GB di RAM e WSL 2 backend installato. Nel caso di MacOS, è possibile installarlo su dispositivi con chip Intel o Apple silicon, purché abbiano 4 GB di RAM e una versione superiore a Sierra, come Big Sur (11), Monterey (12), o Ventura (13).

WSL – Windows Subsystem for Linux

Il sottosistema Windows per Linux consente agli sviluppatori di eseguire un ambiente GNU/Linux, inclusa la maggior parte degli strumenti, delle utilità e delle applicazioni da riga di comando, direttamente su Windows, senza modifiche, senza l'overhead di una macchina virtuale tradizionale o una configurazione *dualboot*. Questo sistema è stato introdotto per permetterti di scegliere le distribuzioni GNU/Linux preferite dal Microsoft Store, eseguire strumenti da riga di comando comuni come *grep*, *sed*, *awk* o altri binari, eseguire script tramite Bash e applicazioni da riga di comando GNU/Linux, installare software aggiuntivo utilizzando il gestore di pacchetti di distribuzione GNU/Linux o anche richiamare le applicazioni Windows utilizzando una shell della riga di comando simile a Unix.

Docker Desktop semplifica lo sviluppo di applicazioni per Kubernetes. Fornisce un'esperienza di configurazione Kubernetes fluida nascondendo la complessità dell'installazione e della configurazione di rete del sistema che lo ospiterà. Questo permette alle persone che sviluppano di concentrarsi interamente sul proprio lavoro piuttosto che occuparsi dei dettagli di configurazione di Kubernetes.

Per configuralo, è sufficiente dunque installare Docker Desktop e aviarlo: non appena pronto, sarà possibile notare in basso a destra l'iconica balena simbolo del marchio Docker Inc. su sfondo verde, che rappresenta il corretto funzionamento del Docker Engine.

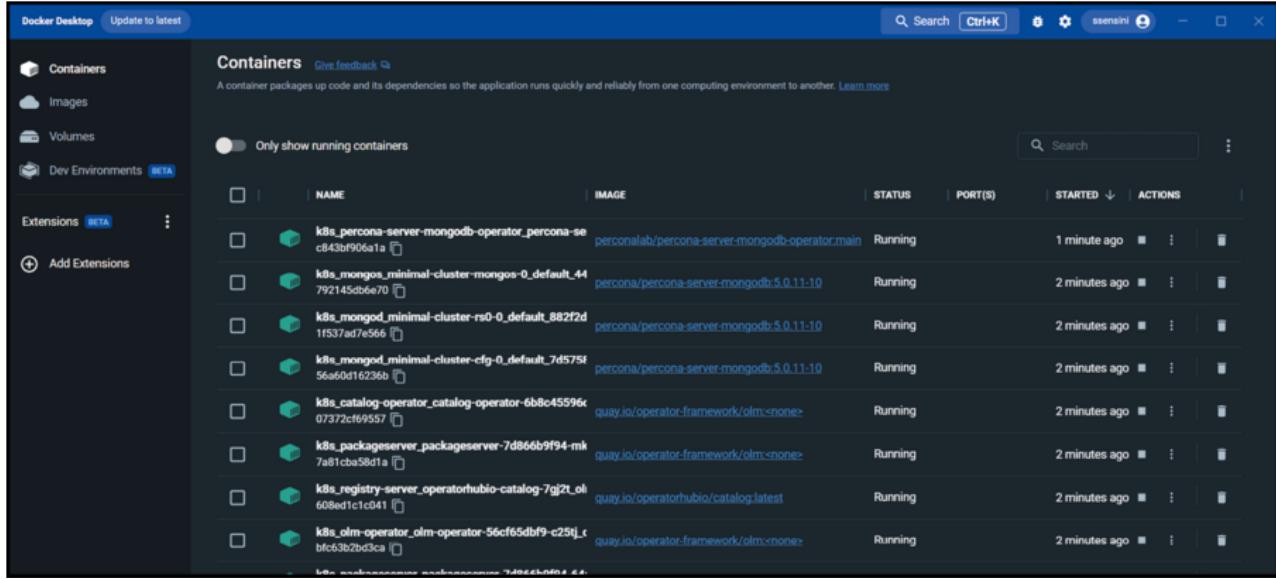


Figura 4.1 Schermata principale di Docker Desktop.

A questo punto, possiamo cliccare sulle impostazioni in alto a destra tramite l'icona apposita e selezionare nel menu di sinistra la voce “Kubernetes”:

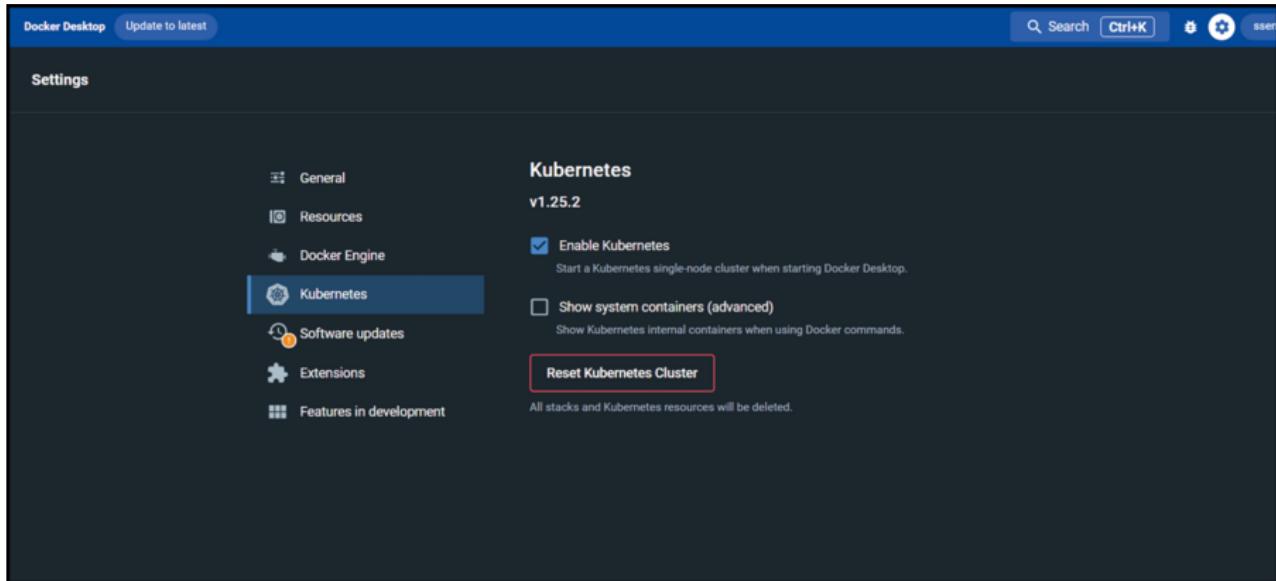


Figura 4.2 Configurazione di Kubernetes tramite le impostazioni di Docker Desktop.

Questa presenta due opzioni: la prima riguarda l'abilitazione di Kubernetes, che ci permetterà non solo di installarlo, ma anche di lasciare a Docker Desktop il compito di configurarlo per noi, predisponendo un cluster basato su singolo nodo. Questo vuol dire che al momento il cluster che utilizzeremo avrà una conformazione come quella riportata nella figura seguente: non vengono riportati tutti i diversi componenti presenti nell'architettura Kubernetes, ma ci basti sapere che avremo a che fare con un solo nodo che lavorerà sia come control-plane, sia come worker.

La seconda opzione ci presenta la possibilità di mostrare, nell'elenco dei containers, anche quelli utilizzati a livello di sistema. Questo vuol dire che container come quelli che gestiscono l'API server, lo scheduler e così via saranno mostrati nell'elenco presente nell'interfaccia grafica:

Ai fini di quanto vedremo all'interno di questo manuale, questo non è necessario, per cui questa opzione può essere tralasciata.

Quando avremo selezionato le opzioni che ci interessano, potremo finalmente passare all'installazione: questa richiederà qualche minuto e una buona connessione Internet, quindi armatevi di pazienza e attendete che la configurazione sia completata. Docker Desktop verrà riavviato almeno una volta e dovrete attendere qualche minuto affinché Kubernetes riesca a essere in esecuzione.

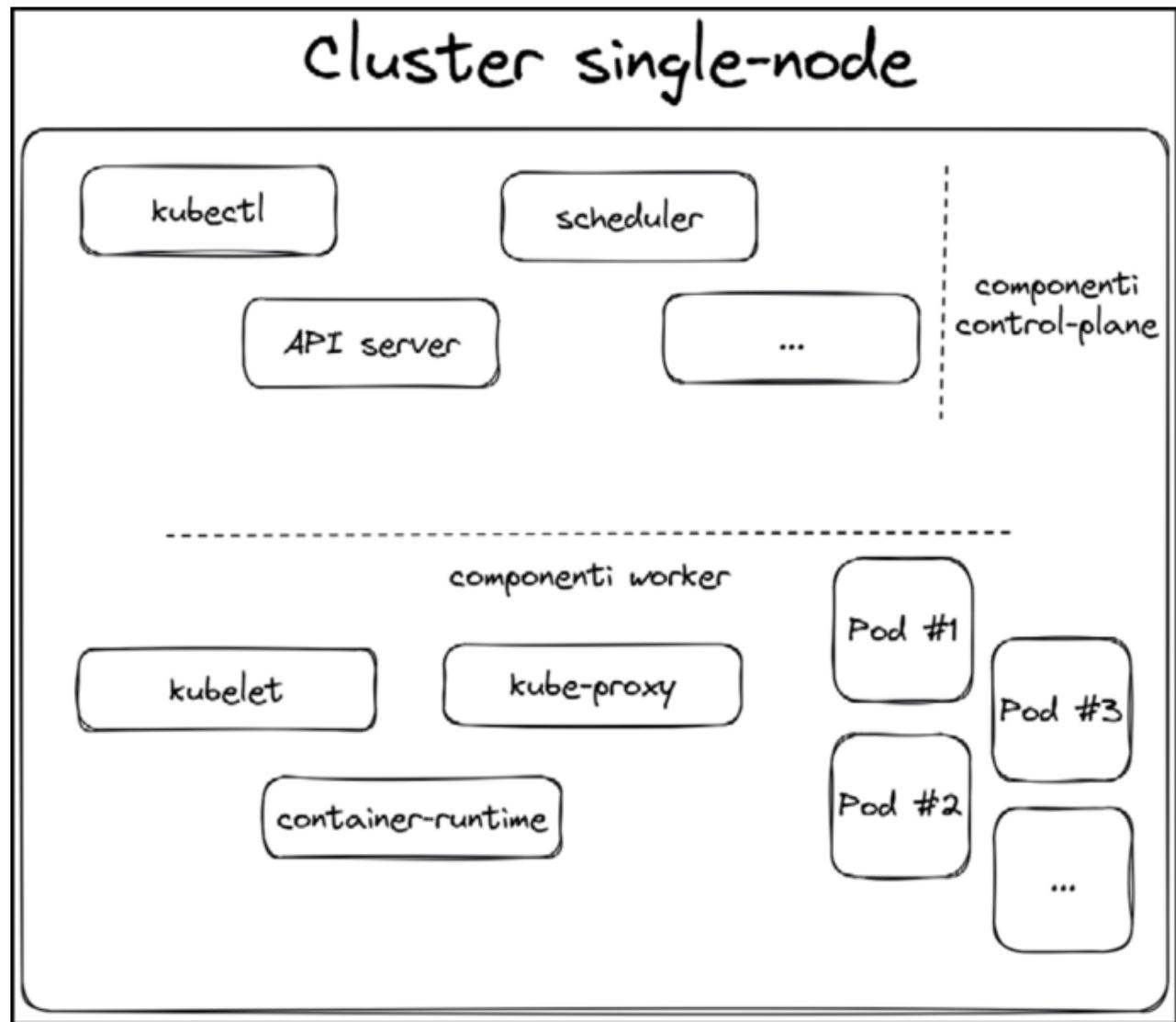


Figura 4.3 Architettura di un cluster a un solo nodo.

Containers Give feedback						
A container packages up code and its dependencies so the application runs quickly and reliably from one computing environment to another. Learn more						
<input type="checkbox"/>	NAME	IMAGE	STATUS	PORT(S)	STARTED	ACTIONS
<input type="checkbox"/>	k8s_percona-server-mongodb-operator_percona-se-c843bf906a1a	perconalab/percona-server-mongodb-operator.main	Running		5 minutes ago	<input type="button"/> <input type="button"/> <input type="button"/>
<input type="checkbox"/>	k8s_mongos_minimal-cluster-mongos-0_default_44792145db6e70	percona/percona-server-mongodb:5.0.11-10	Running		6 minutes ago	<input type="button"/> <input type="button"/> <input type="button"/>
<input type="checkbox"/>	k8s_mongod_minimal-cluster-rs0-0_default_882f2d1f537ad7e566	percona/percona-server-mongodb:5.0.11-10	Running		6 minutes ago	<input type="button"/> <input type="button"/> <input type="button"/>
<input type="checkbox"/>	k8s_mongod_minimal-cluster-cfg-0_default_7d575656a60d16236b	percona/percona-server-mongodb:5.0.11-10	Running		6 minutes ago	<input type="button"/> <input type="button"/> <input type="button"/>
<input type="checkbox"/>	k8s_catalog-operator_catalog-operator-6b8c45596c07372cf69557	quay.io/operator-framework/olm:<none>	Running		6 minutes ago	<input type="button"/> <input type="button"/> <input type="button"/>
<input type="checkbox"/>	k8s_packageserver_packageserver-7d866b9f94-mk7a1tcb58d1a	quay.io/operator-framework/olm:<none>	Running		6 minutes ago	<input type="button"/> <input type="button"/> <input type="button"/>
<input type="checkbox"/>	k8s_registry-server_operatorhubio-catalog-7gj2t_oli608ed1c1c041	quay.io/operatorhubio/catalog:latest	Running		6 minutes ago	<input type="button"/> <input type="button"/> <input type="button"/>
<input type="checkbox"/>	k8s_olm-operator_olm-operator-56cf65dbf9-c25tj_bfc63b2bd3ca	quay.io/operator-framework/olm:<none>	Running		6 minutes ago	<input type="button"/> <input type="button"/> <input type="button"/>
<input type="checkbox"/>	k8s_packageserver_packageserver-7d866b9f94-64:186ee3db2667	quay.io/operator-framework/olm:<none>	Running		6 minutes ago	<input type="button"/> <input type="button"/> <input type="button"/>
<input type="checkbox"/>	k8s_kubernetes-dashboard_kubernetes-dashboard-19390a0e211d4287a7a	bitnami/kubernetes-dashboard:7.0.0	Running		6 minutes ago	<input type="button"/> <input type="button"/> <input type="button"/>

Showing 27 items

Figura 4.4 Elenco dei Pod di sistema di Kubernetes.

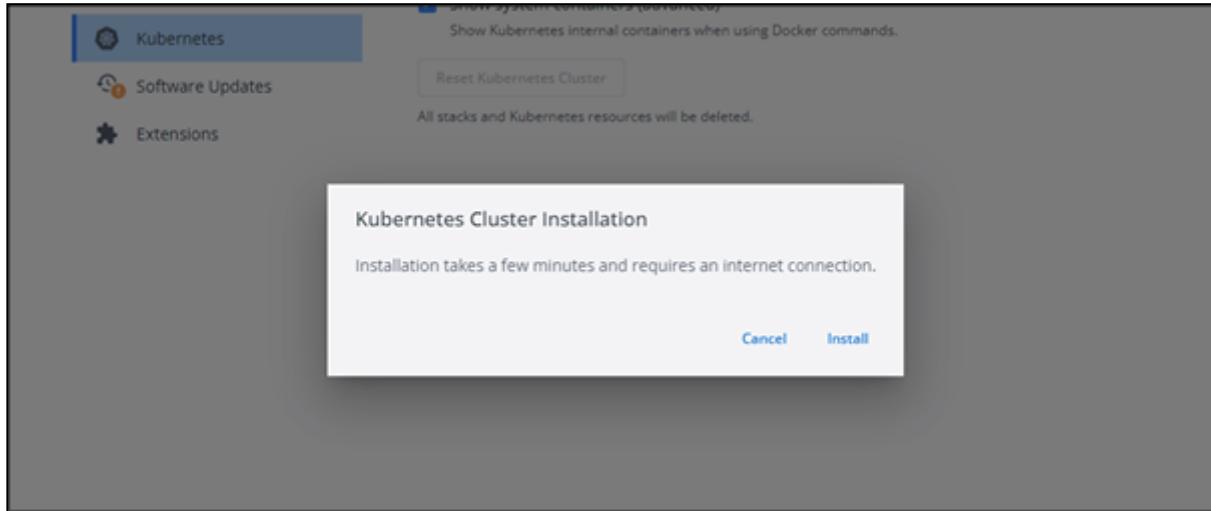


Figura 4.5 Installazione del cluster Kubernetes.

Una volta terminata l'installazione, e partito il Docker Engine insieme a Kubernetes, noteremo che le icone in basso a destra sono due e sono verdi.

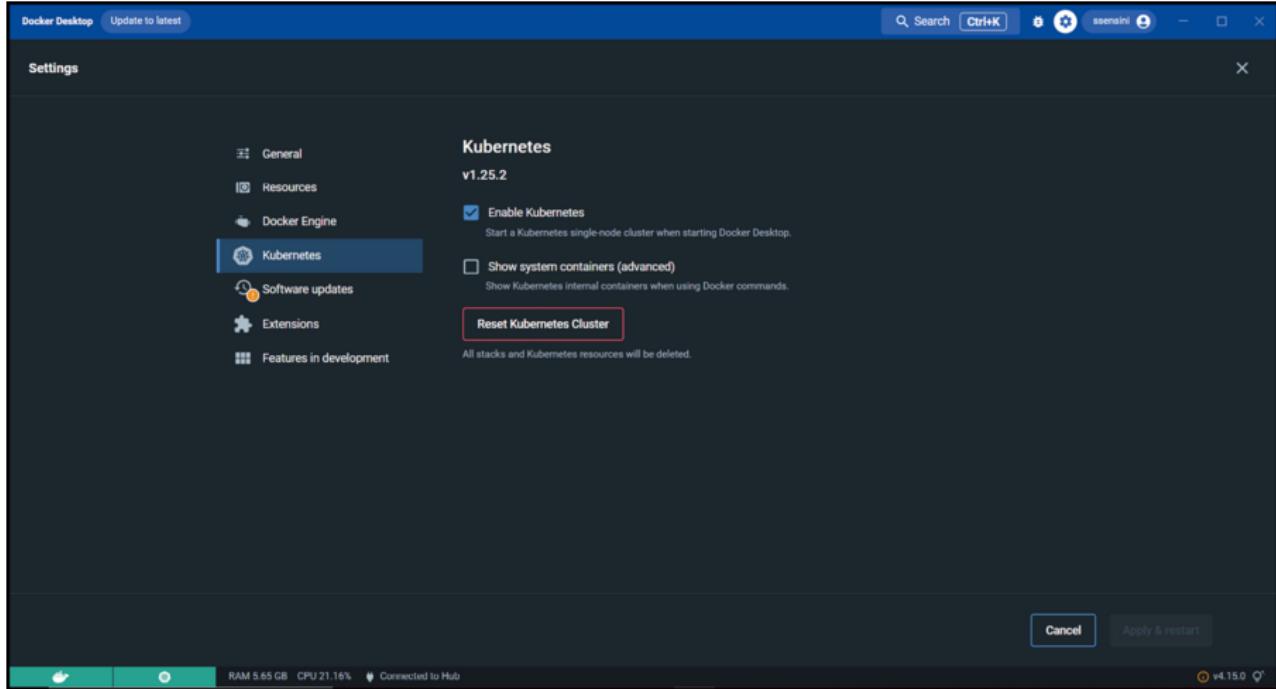


Figura 4.6 Kubernetes e Docker sono entrambi attivi, come visibile dalle due icone in verde in basso a sinistra.

A questo punto, possiamo iniziare a lavorare con Kubernetes e con i suoi primi componenti. Facciamo però un passo indietro prima di proseguire con le prossime attività: cosa succede dietro le quinte di Docker Desktop, e com'è possibile utilizzare Kubernetes su di un semplice laptop, senza dover configurare un cluster che richieda molte più risorse di quelle di cui disponiamo?

Internamente, le seguenti azioni vengono attivate nel backend di Docker Desktop:

- generazione di certificati;
- configurazione del cluster;
- download e installazione dei componenti interni di Kubernetes;
- avvio del cluster;
- configurazione della gestione di rete e della persistenza.

Il diagramma seguente mostra le interazioni tra i componenti interni di Docker Desktop per la configurazione del cluster.

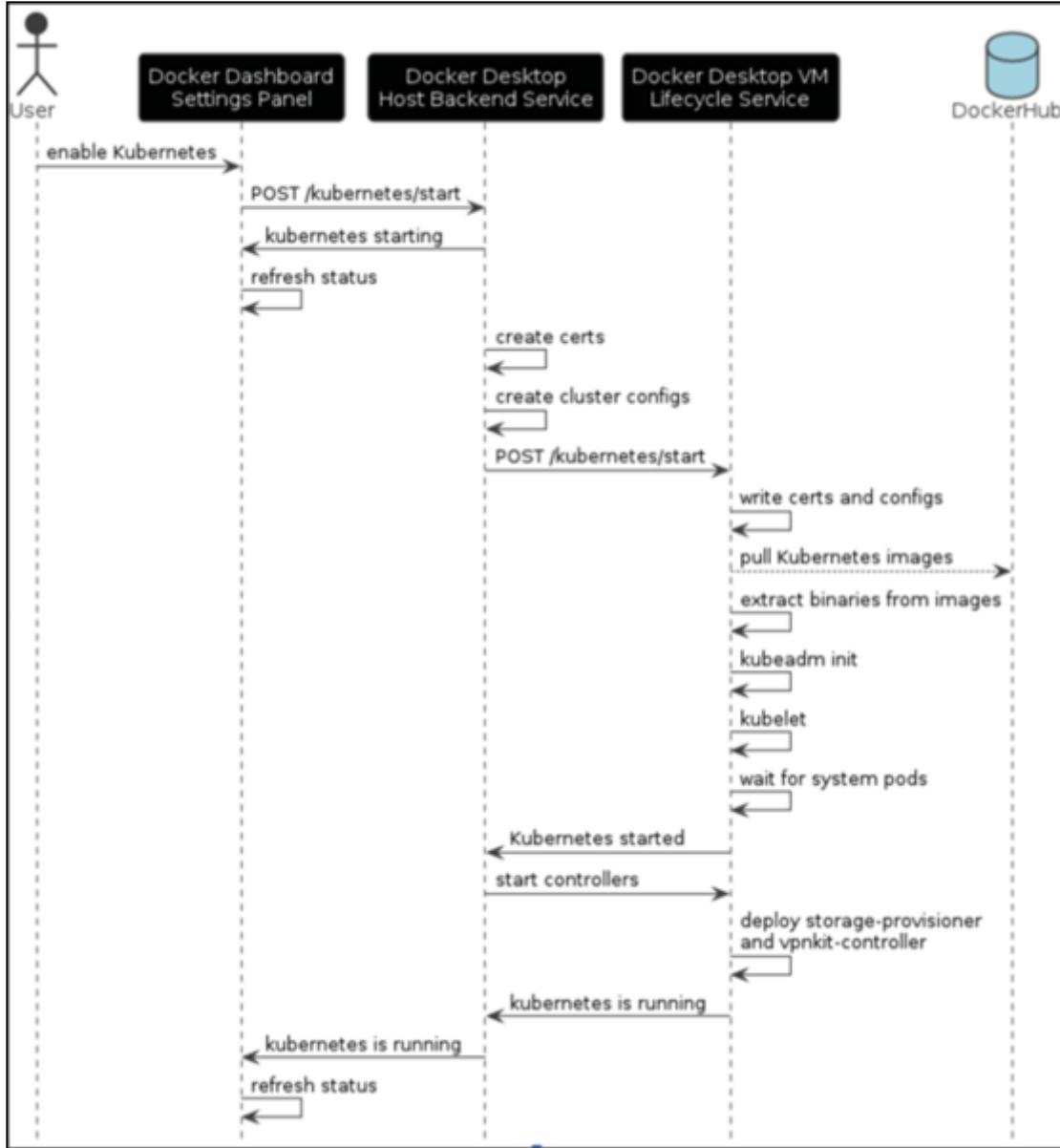


Figura 4.7 Funzionamento di Docker Desktop e dei suoi componenti.

Per poter sfruttare al meglio le potenzialità di questo ambiente, è interessante spendere due parole su ognuna di queste attività, e comprendere quanto lavoro ci viene risparmiato con questa installazione.

Generazione dei certificati

Kubernetes richiede dei certificati con relative chiavi per le connessioni autenticate tra i suoi componenti interni e con l'esterno. Docker Desktop si occupa di generare questi certificati server e client per i principali servizi interni: kubelet (node manager), service account management, frontproxy, API server e i componenti etcd saranno quindi autenticati e potranno comunicare tra loro in sicurezza.

Configurazione del cluster

Docker Desktop installa Kubernetes utilizzando `kubeadm`, pertanto deve creare il runtime `kubeadm` e la configurazione a livello di cluster. Ciò include la configurazione di rete del cluster, i certificati, l'endpoint del nodo e così via. Anche tutti i nomi che verranno utilizzati dagli URL delle risorse che utilizzeremo per far comunicare i diversi componenti saranno sempre impostati tramite Docker Desktop, mentre l'endpoint globale del cluster utilizzerà il nome DNS <https://kubernetes.docker.internal:6443>. La porta 6443 è infatti la porta predefinita a cui è associato il nodo principale di Kubernetes.

Installazione dei componenti principali

All'interno di Docker Desktop, un processo di gestione chiamato *Lifecycle service* si occupa dell'avvio di servizi come il demone Docker e della notifica del loro cambiamento di stato. Per esempio, una volta generati i certificati nel primo passaggio e completata la configurazione di Kubernetes nel secondo, viene inviata una richiesta al servizio Lifecycle per installare e avviare Kubernetes. La richiesta contiene i certificati necessari per l'installazione. Questo servizio inizia quindi a estrarre tutte le immagini dei componenti interni di Kubernetes da Docker Hub e a scaricarle per il successivo avvio. Queste immagini contengono file binari come `kubelet`, `kubeadm`, `kubectl`, `crlctl` ecc. che vengono estratti e inseriti nel path `/usr/bin`.

Avvio del cluster

Una volta che questi binari sono attivi e funzionanti, e i file di configurazione sono stati aggiunti nei percorsi corretti, il servizio Lifecycle esegue il comando `kubeadm init` per inizializzare il cluster e quindi avviare il processo `kubelet`. Poiché si tratta di un cluster con un singolo nodo, viene eseguita solo un'istanza `kubelet`. Il servizio Lifecycle attende quindi che i Pod di sistema come `coredns`, `kube-controller-manager` e `kube-apiserver` siano in esecuzione per notificare al servizio principale di Docker Desktop che Kubernetes è stato avviato.

Configurazione della gestione di rete e della persistenza

Una volta avviati i servizi interni di Kubernetes, Docker Desktop avvia l'installazione di controller aggiuntivi come lo *storage-provisioner* e *vpnkit-controller*. I loro ruoli riguardano la persistenza dello stato dell'applicazione a seguito di possibili riavvii/aggiornamenti e come accedere alle applicazioni una volta che queste sono state installate nel cluster. Una volta che questi controller sono attivi e in esecuzione, il cluster Kubernetes è completamente operativo e possiamo eseguire i comandi `kubectl` per distribuire le applicazioni.

Configurazione della dashboard

Sarebbe davvero bello se Docker Desktop includesse una GUI per Kubernetes pronta all'uso, ma possiamo ottenere un'esperienza simile utilizzando una *Kubernetes Dashboard*. Tramite il suo progetto ufficiale (repository del progetto: <https://github.com/kubernetes/dashboard>), sempre frutto del lavoro del team di lavoro di Kubernetes, viene fornita un'interfaccia utente Web per la gestione del cluster, per garantire un modo semplice che permetta di visualizzare una panoramica sullo stato del cluster e persino per la modifica e l'esecuzione dei container e delle altre risorse che vedremo.

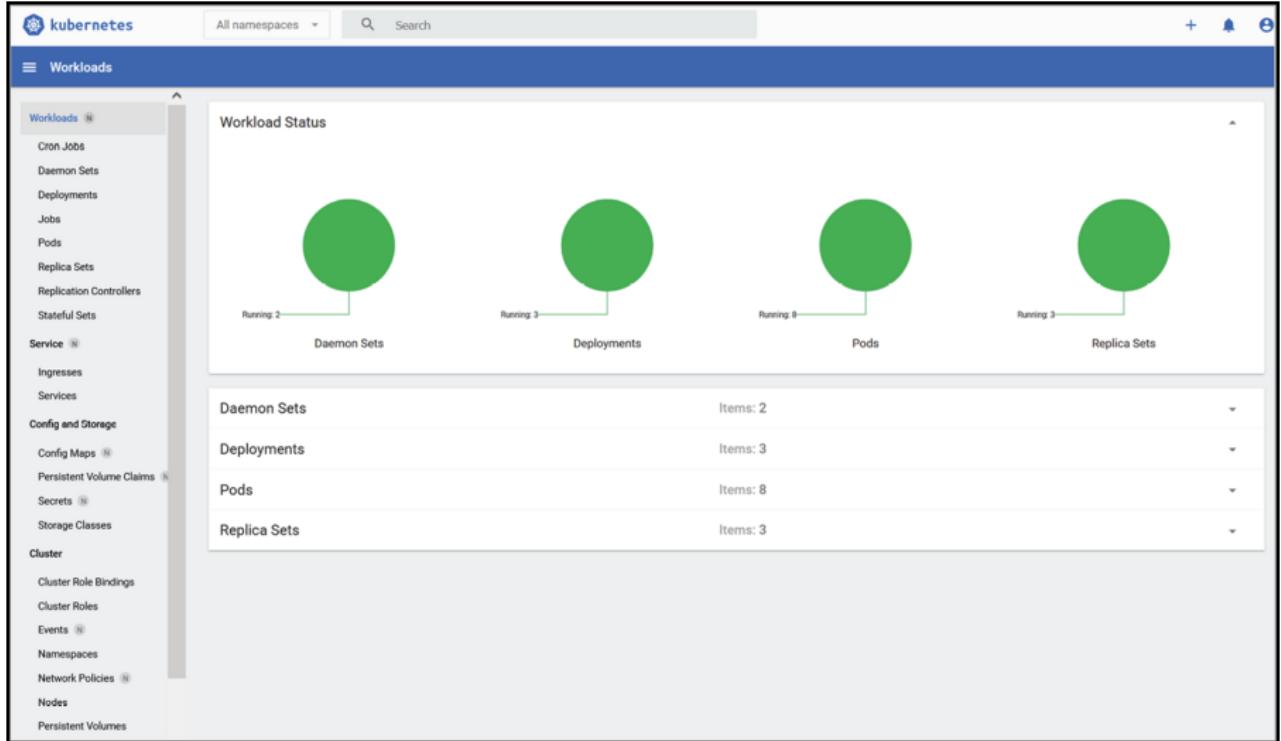


Figura 4.8 Pagina principale della dashboard di Kubernetes.

Per installare Kubernetes Dashboard, apri un terminale ed esegui quanto segue:

Listato 4.1 Configurazione della dashboard

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml

>>>
namespace/kubernetes-dashboard created
serviceaccount/kubernetes-dashboard created
service/kubernetes-dashboard created
secret/kubernetes-dashboard-certs created
secret/kubernetes-dashboard-csrf created
secret/kubernetes-dashboard-key-holder created
configmap/kubernetes-dashboard-settings created
role.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrole.rbac.authorization.k8s.io/kubernetes-dashboard created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
deployment.apps/kubernetes-dashboard created
service/dashboard-metrics-scraper created
deployment.apps/dashboard-metrics-scraper created
```

L'output dovrebbe essere simile a quanto riportato; se adesso questi comandi sembreranno confusi e poco chiari, non c'è da preoccuparsi: tutto sarà più chiaro quando entreremo nel vivo di questa tecnologia. Per ora, basti sapere che il comando precedente ha preso un file YAML e l'ha dato in pancia al motore di Kubernetes, che ha creato una serie di oggetti a partire dalla loro definizione.

Dal momento che non ci intendiamo ancora di utenti, dovremo anche fare qualche step in più per poter accedere alla dashboard da browser senza alcuna credenziale: eseguiamo i seguenti comandi per poter “aggiustare” le risorse appena create, così che siano in grado di garantirci l'accesso al cluster in maniera insicura. Questo è un procedimento da seguire *solo* se stiamo lavorando in un ambiente di sviluppo come il nostro laptop: il rischio è quello di permettere a utenti malintenzionati di entrare nel

cluster! La dashboard Kubernetes consente di ignorare la pagina di accesso se modifichiamo la risorsa creata in precedenza aggiungendo degli argomenti, o tramite il comando `kubectl edit` o tramite `kubectl patch`. In questo caso, si seguirà il secondo approccio, che è più semplice: si andrà a richiedere di abilitare il pulsante per saltare la login (primo comando), per abilitare una login non "sicura" e per rimuovere la richiesta di un utente per accedere alla dashboard.

Listato 4.2 Patch del Deployment kubernetes-dashboard

```
kubectl patch deployment kubernetes-dashboard -n kubernetes-dashboard --type 'json' -p
'[{"op": "add", "path": "/spec/template/spec/containers/0/args/-", "value": "--enable-skip-
login"}]'
>>>
deployment.apps/kubernetes-dashboard patched

kubectl patch deployment kubernetes-dashboard -n kubernetes-dashboard --type 'json' -p
'[{"op": "add", "path": "/spec/template/spec/containers/0/args/-", "value": "--enable-
insecure-login"}]'
>>>
deployment.apps/kubernetes-dashboard patched

kubectl patch deployment kubernetes-dashboard -n kubernetes-dashboard --type 'json' -p
'[{"op": "add", "path": "/spec/template/spec/containers/0/args/-", "value": "--disable-
settings-authorizer"}]'
>>>
deployment.apps/kubernetes-dashboard patched
```

L'ultimo step permette di far sì che tutti i progetti che useremo siano visibili attraverso questa dashboard: assegnamo alla risorsa che ha installato la dashboard e di cui assumeremo l'identità i permessi massimi per agire all'interno del cluster:

Listato 4.3 Abilitazione dei permessi per tutti i namespace

```
kubectl delete clusterrolebinding kubernetes-dashboard
>>>
clusterrolebinding.rbac.authorization.k8s.io "kubernetes-dashboard" deleted

kubectl create clusterrolebinding kubernetes-dashboard --clusterrole=cluster-admin --
serviceaccount=kubernetes-dashboard:kubernetes-dashboard
>>>
clusterrolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
```

Una volta completati questi step, potremo accedere alla dashboard digitando il comando `kubectl proxy` nel terminale, che ci farà da ponte tra il laptop locale e il cluster, e collegandoci tramite browser al seguente indirizzo:

Listato 4.4 Indirizzo della dashboard

`http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-
dashboard:/proxy/`

Installazione tramite CRC

OpenShift sta diventando una delle soluzioni di fatto per le implementazioni di software su cloud e soprattutto per qualsiasi azienda che stia pianificando di passare a una soluzione dove l'infrastruttura è gestita dal provider. Con l'ultima versione di OpenShift, Minishift non è più supportato e così, qualche tempo fa, RedHat ha rilasciato OpenShift Local (alias CodeReady Containers), per consentire alle persone che sviluppano di testare una configurazione di OpenShift 4.x in un ambiente locale installando un cluster a nodo singolo. Lo scopo di CodeReady Containers è infatti quello di fornirti un cluster Openshift minimo ottimizzato per scopi di sviluppo. Il processo di installazione è molto semplice

ed è descritto in questi passaggi: avrai solo bisogno di un account RedHat (è gratis!) e di configurare alcune risorse gratuite nel tuo laptop. Attenzione: il cluster CodeReady Containers è effimero e non è destinato all'uso in produzione, ma è utile per testare le applicazioni e verificare se il carico di lavoro funziona come previsto.

I requisiti software prevedono:

- un hypervisor per eseguire la VM contenente OpenShift;
- *macOS*: xhyve (predefinito), VirtualBox;
- *GNU/Linux*: KVM (predefinito), VirtualBox;
- *Windows*: Hyper-V (predefinito), VirtualBox;
- un account su RedHat (è gratis!) per accedere a questa console e poter ottenere il pull secret.

Notare che su sistemi Microsoft Windows, Red Hat OpenShift Local richiede Windows 10 Fall Creators Update (versione 1709) o successiva. Non funziona con le versioni precedenti di Microsoft Windows. Microsoft Windows 10 Home Edition non è supportato. Per quanto riguarda Mac OS, Red Hat OpenShift Local richiede macOS 11 Big Sur o versioni successive. Non funziona su versioni precedenti di macOS. Infine, per chi utilizza Linux, è supportato solo sulle ultime due versioni minori di Red Hat Enterprise Linux/CentOS 7, 8 e 9 e sulle ultime due versioni stabili di Fedora. Ubuntu 18.04 LTS o successivo e Debian 10 o successivo non sono supportati e potrebbero richiedere la configurazione manuale del computer host.

Per i requisiti hardware, è necessario disporre di 4 core CPU fisici, 9 GB di memoria libera e almeno 35 GB di spazio di archiviazione.

Come primo step, accedi alla RedHat Console e facendo clic sulla voce *All apps and services* scegli il servizio *OpenShift*; altrimenti, vai su questo link: <https://console.redhat.com/openshift>.

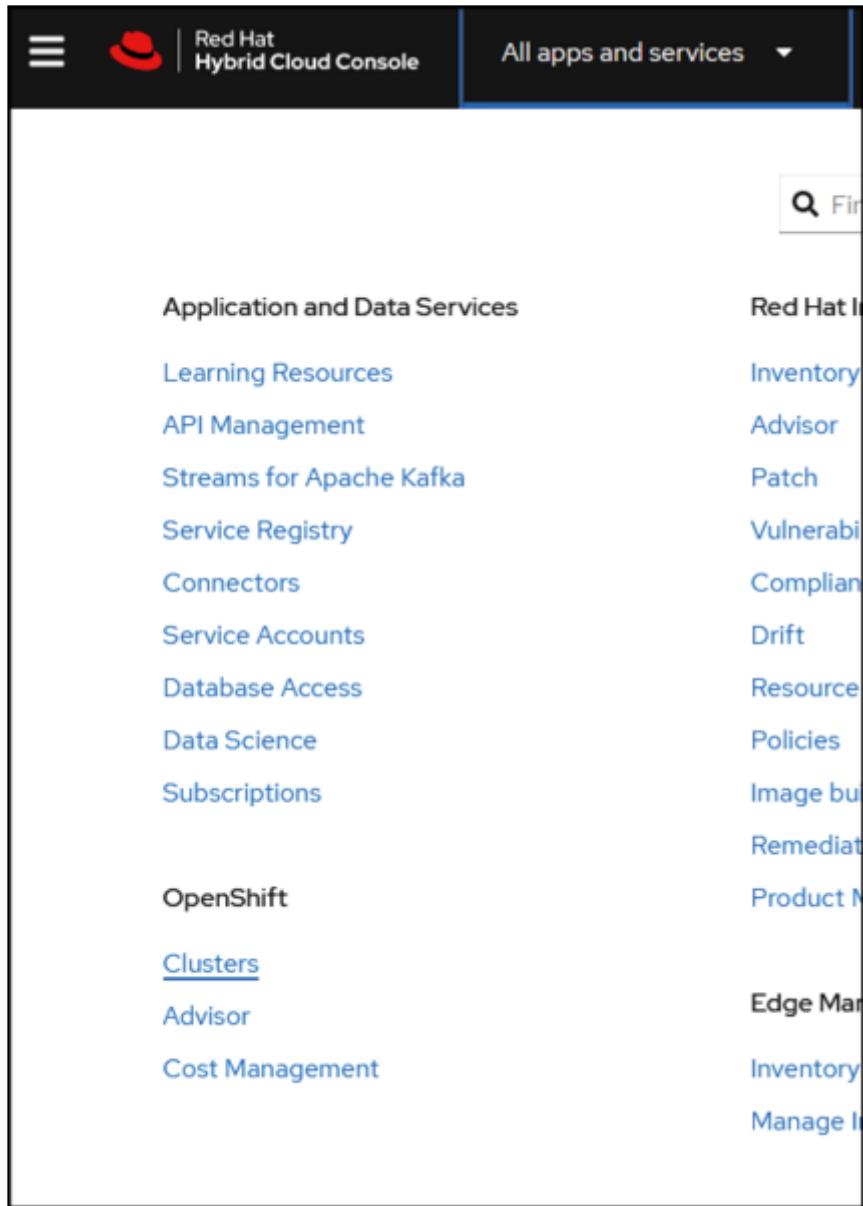


Figura 4.9 Menu della console RedHat per accedere a OpenShift Local.

Fai clic su *Cluster* nel menu a sinistra e poi su *Create*, selezionando la scheda *Local*, come mostrato nella Figura 4.10.

Ti verrà chiesto di scaricare un pacchetto in base al tuo sistema operativo (vedi Figura 4.11); clicca su *Download OpenShift Local* e non chiudere la pagina; dovremo recuperare tramite questa finestra il pull secret tra un minuto.

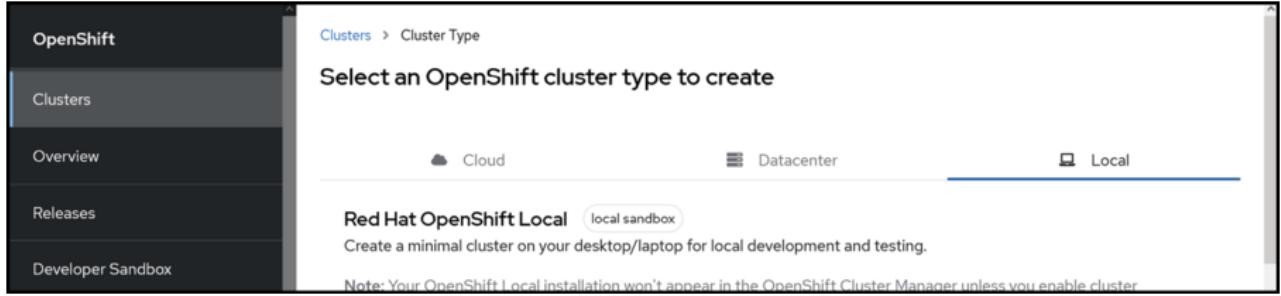


Figura 4.10 Selezione del tipo di installazione per OpenShift.

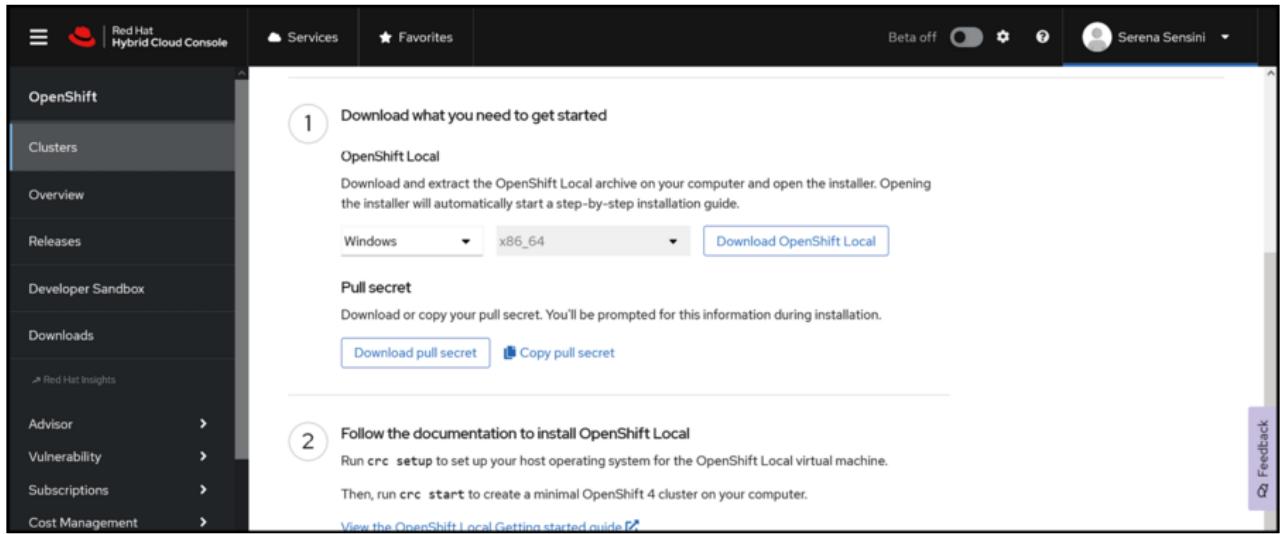


Figura 4.11 Scelta dell'eseguibile da scaricare per procedere all'installazione.

In base al tuo sistema operativo, installa il file eseguibile e segui ogni passaggio descritto dalla procedura guidata e, a scopo di supporto, lascia il percorso predefinito per l'installazione.

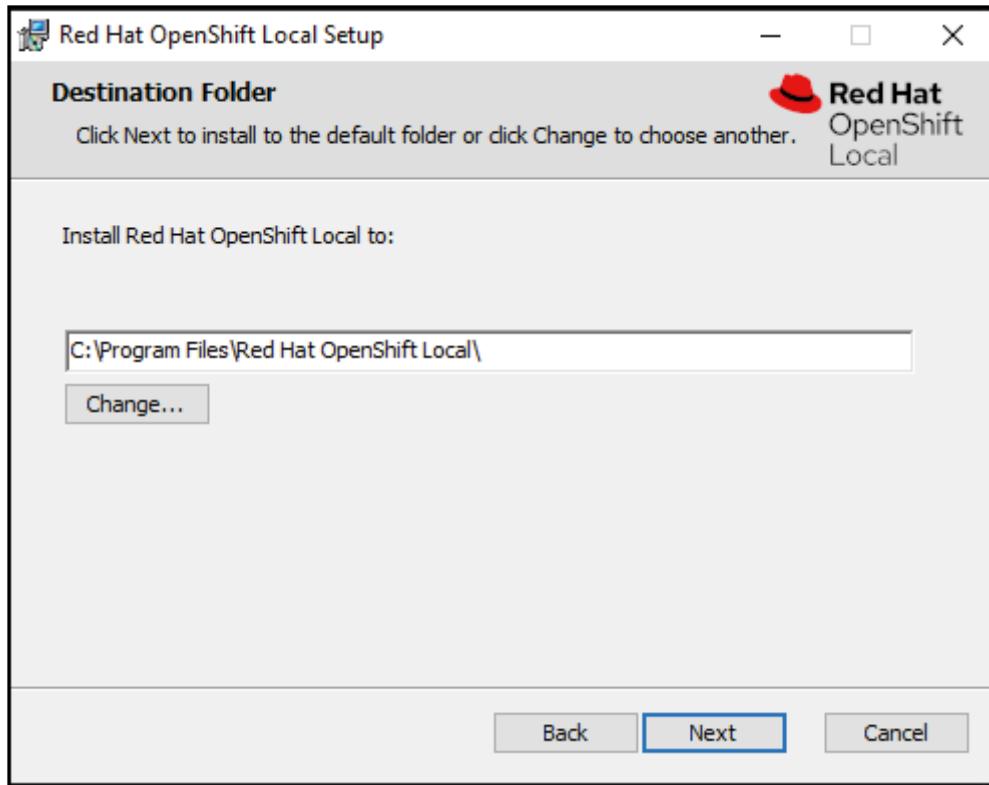


Figura 4.12 Schermata di installazione di OpenShift Local.

Una volta riavviato il sistema, apri RedHat OpenShift Local, fai clic su *Next* e vai al secondo passaggio. Qui puoi scegliere se vuoi installare il preset di OpenShift o Podman; non si escludono a vicenda, ma puoi scegliere di installare Podman in un secondo momento o manualmente. Lascia la selezione corrente e continua.

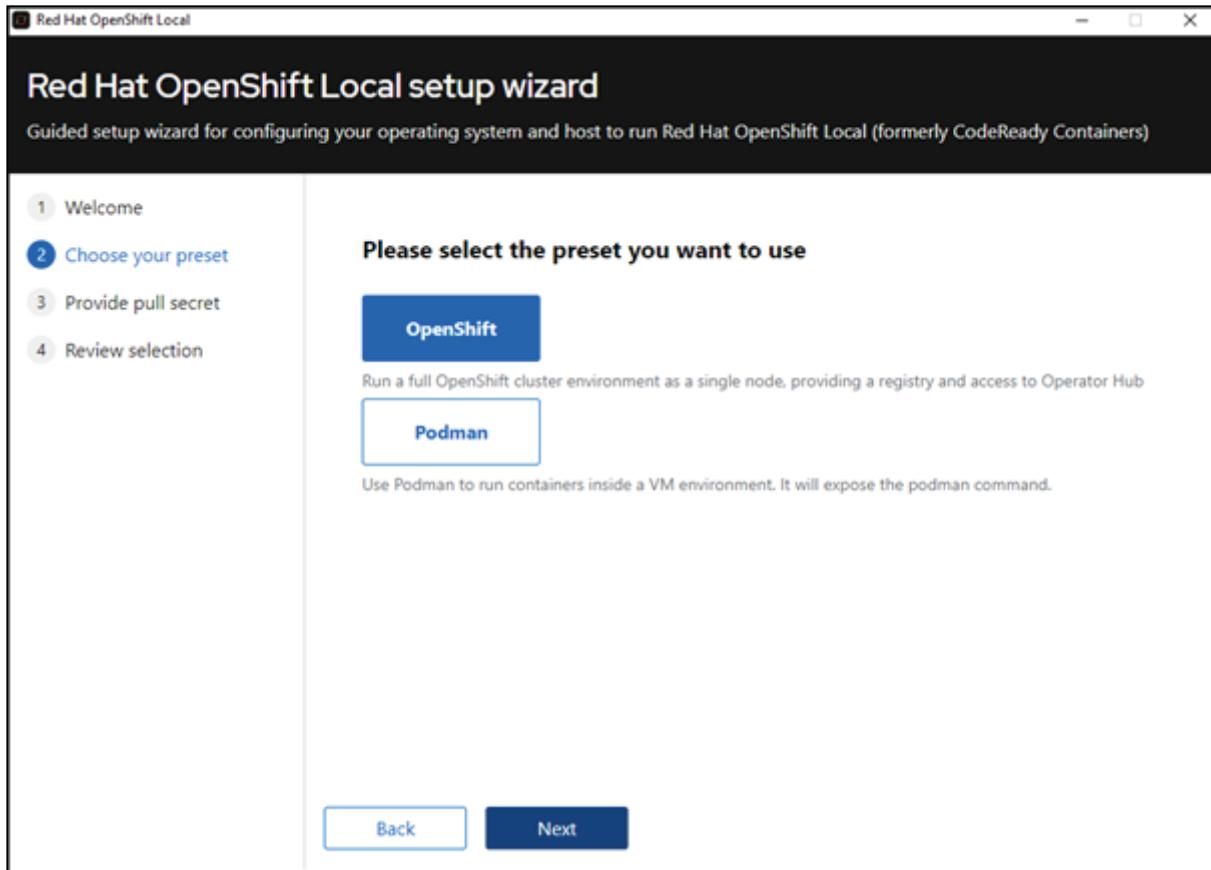


Figura 4.13 Scelta dello strumento da utilizzare: la selezione è su OpenShift.

Torna alla console di RedHat e copia e incolla il pull secret facendo clic sul collegamento mostrato nell'immagine seguente:

1 Download what you need to get started

OpenShift Local

Download and extract the OpenShift Local archive on your computer and open the installer. Opening the installer will automatically start a step-by-step installation guide.

Windows x86_64 [Download OpenShift Local](#)

Pull secret

Download or copy your pull secret. You'll be prompted for this information during installation.

[Download pull secret](#) [Copy pull secret](#)

Figura 4.14 Recupero del pull secret.

Per gestire il tuo cluster e migliorare il supporto RedHat, consenti la telemetria e fai clic su *Esegui installazione* per completare l'installazione. L'operazione richiede fino a 5-15 minuti. Se richiede di più, prova a chiudere e ricominciare.

Per le persone che usano Windows, dovresti essere in grado di vedere l'icona rossa nella barra degli strumenti: fai clic destro su di essa e controlla se vedi il cluster in esecuzione.

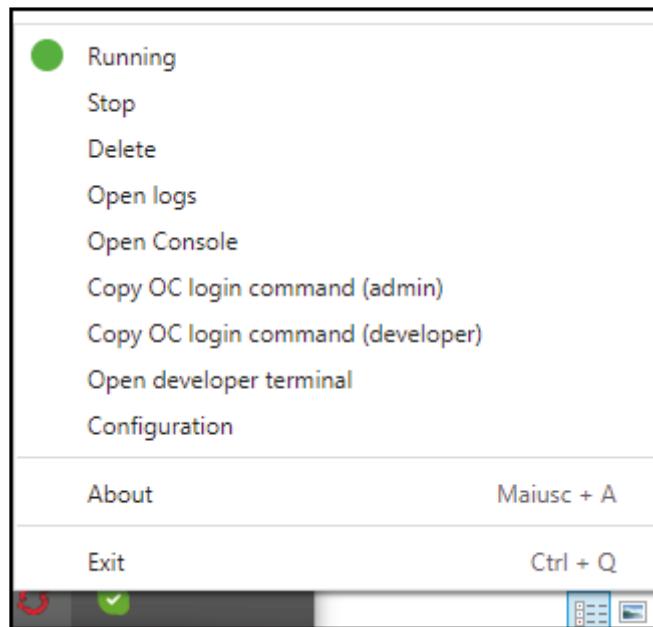


Figura 4.15 Menu di gestione di OpenShift Local.

Tieni presente che questo strumento verrà gestito attraverso questa finestra: qui puoi fermare il cluster, cancellarlo e ricominciare da zero (opzione *Delete*), aprire la console web (opzione *Open console*), copiare e incollare il comando di login come amministratore (opzione *Copy...*), e così via.

Una volta avviato il tuo cluster (utilizzando `crc start` nella riga di comando o utilizzando l'interfaccia utente), la tua console sarà disponibile su <https://oauth.openshift.apps-crc.testing>. Per ottenere le tue credenziali, esegui il seguente comando:

Listato 4.5 Login tramite CRC

```
$ crc console --credentials

To login as a regular user, run 'oc login -u developer -p developer
https://api.crc.testing:6443'.
To login as an admin, run 'oc login -u kubeadmin -p PASSWORD https://api.crc.testing:6443'
```

Ultima nota: nell'impostazione di Docker Desktop, nella sezione relativa a Kubernetes, noteremo anche un pulsante rosso per resettare il cluster Kubernetes: questo è uno dei motivi per cui Docker Desktop è la migliore delle opzioni Kubernetes locali. Se stiamo sviluppando o facendo dei test che potrebbero portare il cluster in uno stato inconsistente, è sufficiente fare clic su di esso e il cluster verrà ripristinato alla versione iniziale come su una nuova installazione di Kubernetes, senza doverci rimettere mano.

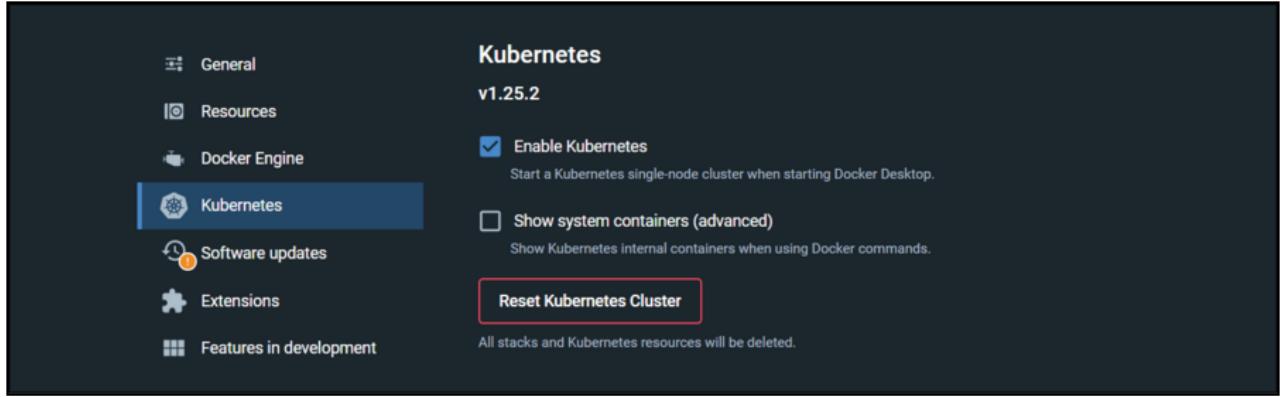


Figura 4.16 Pulsante di reset di Kubernetes.

Minikube

Questa soluzione permette di installare velocemente un cluster composto da un singolo nodo su qualsiasi sistema operativo, il cui principale requisito è quello di avere un software di virtualizzazione. Questo vuol dire che potremo installarlo sfruttando un container (sì, possiamo usare Docker come driver per installarlo) oppure usare uno dei sistemi di virtualizzazione come Hyper-V, KVM, Virtualbox e via dicendo. Prima di procedere, verifichiamo se è già presente uno di questi strumenti nel nostro sistema, e poi procediamo con l'installazione: dovremo infatti prima scaricare Minikube e poi installare anche kubectl per poter “comunicare” con il cluster. Sarà infatti Minikube a occuparsi di creare il contenitore del nostro cluster per permetterci di gestirne il ciclo di vita, mentre kubectl ci permetterà di governare le risorse installate al suo interno!

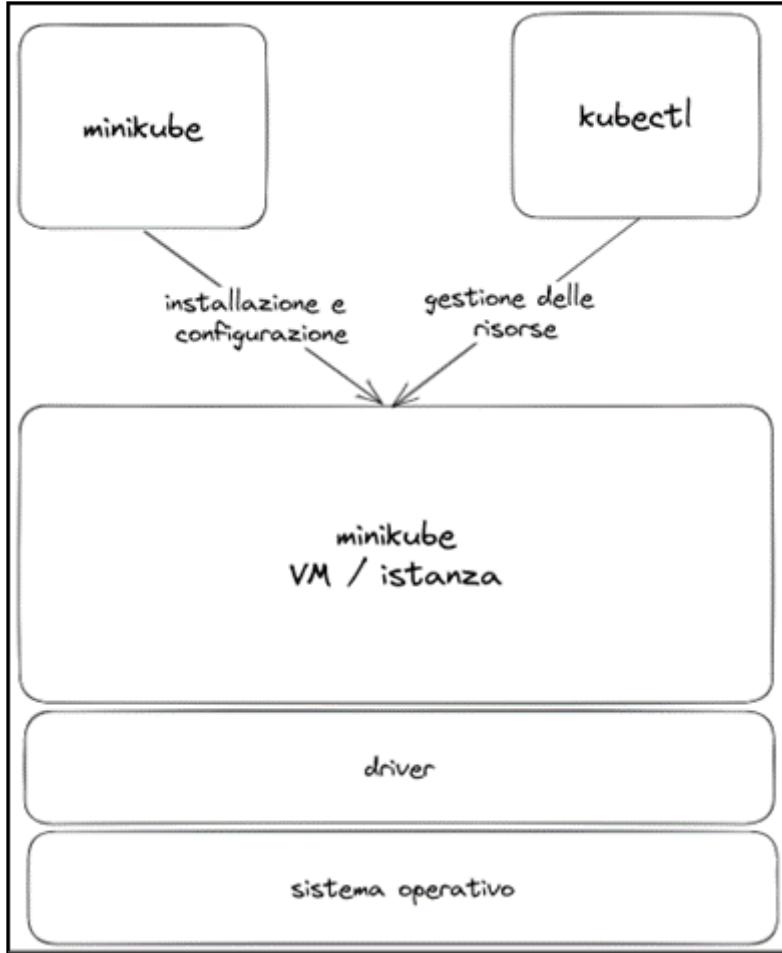


Figura 4.17 Architettura di Minikube.

Come visibile in figura, per lavorare con Minikube, quello che avviene è la generazione di un'istanza che ci permette di creare un cluster a singolo nodo (vedremo più in là cosa significa) attraverso lo strato di virtualizzazione che si occupa di comunicare con il sistema operativo. Questo strato intermedio tra Minikube e il sistema operativo può essere uno di quelli citati, ma il risultato non cambia: Minikube avrà il suo ambiente, con cui potremo interagire successivamente grazie a kubectl.

Di seguito viene quindi riportata la modalità con cui eseguire le diverse installazioni a seconda del sistema operativo utilizzato, anche se è possibile fare riferimento alla documentazione ufficiale presente sul sito di Minikube: <https://minikube.sigs.k8s.io/docs/start/>.

Windows

Per installare Minikube, scarichiamo l'ultima release disponibile nel formato .exe (o sfruttando Chocolatey, se presente) e aggiungendo il percorso dove questo verrà salvato alla variabile di ambiente PATH, così da poterlo richiamare tramite il terminale senza problemi.

1 Installation

Click on the buttons that describe your target platform. For other architectures, see [the release page](#) for a complete list of minikube binaries.

Operating system	Linux	macOS	Windows
Architecture	x86-64		
Release type	Stable	Beta	
Installer type	.exe download	Windows Package Manager	Chocolatey

To install the latest minikube **stable** release on **x86-64 Windows** using **.exe download**:

1. Download and run the installer for the [latest release](#).
Or if using `PowerShell`, use this command:

```
New-Item -Path 'c:\' -Name 'minikube' -ItemType Directory -Force
Invoke-WebRequest -OutFile 'c:\minikube\minikube.exe' -Uri 'https://github.com/kubernetes/minikube/releases/latest/dc
< >
```

2. Add the `minikube.exe` binary to your `PATH`.
Make sure to run PowerShell as Administrator.

Figura 4.18 Istruzioni di installazione di Minikube per Windows.

Linux/macOS

Per questo tipo di sistemi operativi, il procedimento è ancora più semplice: con entrambi abbiamo la possibilità di eseguire da terminale il comando `curl` per scaricare l'ultima release di Minikube e di installarla sempre tramite comando. L'unico cambiamento da apportare tra i due comandi è nel tipo di architettura adottata, che a seconda del sistema può cambiare:

Listato 4.6 Installazione di Minikube

```
# Esempio per macOS
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-darwin-amd64
sudo install minikube-darwin-amd64 /usr/local/bin/minikube
# Esempio per Linux
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

1 Installation

Click on the buttons that describe your target platform. For other architectures, see [the release page](#) for a complete list of minikube binaries.

Operating system	<input checked="" type="button"/> Linux	<input type="button"/> macOS	<input type="button"/> Windows		
Architecture	<input checked="" type="button"/> x86-64	<input type="button"/> ARM64	<input type="button"/> ARMv7	<input type="button"/> ppc64	<input type="button"/> S390x
Release type	<input checked="" type="button"/> Stable	<input type="button"/> Beta			
Installer type	<input checked="" type="button"/> Binary download	<input type="button"/> Debian package	<input type="button"/> RPM package		

To install the latest minikube **stable** release on **x86-64 Linux** using **binary download**:

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

Figura 4.19 Istruzioni di installazione di Minikube per Linux.

Una volta configurato Minikube, potremo avviare il cluster: con questa installazione, viene fornito infatti il comando `minikube`, che ci permette di gestire il ciclo di vita del cluster. Per esempio, possiamo tirare su il cluster con questo comando:

Listato 4.7 Avvio del cluster

```
minikube start
```

Se Minikube non si avvia, bisogna consultare la pagina dei driver per configurare un sistema di virtualizzazione compatibile; Per esempio, se volessimo specificare a Minikube di utilizzare Hyper-V, potremmo indicare in questo modo di utilizzarlo come driver:

Listato 4.8 Specifica del driver Hyper-V

```
minikube start --driver=hyperv
minikube config set driver hyperv # per renderlo il driver di default
```

Se invece volessimo utilizzare Docker, potremmo eseguire lo stesso comando precedente, ma specificando `docker` come valore del driver.

Listato 4.9 Specifica del driver Docker

```
minikube start --driver=docker
minikube config set driver docker # per renderlo il driver di default
```

Abbiamo detto che `minikube` è il comando per gestire il cluster, ma per poter interagire con le risorse al suo interno, avremo bisogno di `kubectl`: per installarlo tramite Minikube, possiamo utilizzare il comando seguente che andrà a scaricare la versione corretta e ci permetterà di utilizzarlo in funzione dell'istruzione `minikube`:

Listato 4.10 Esempio di utilizzo di kubectl

```
minikube kubectl -- get po -A
```

Che cosa succede nel pratico? Nel caso di Virtualbox (o simili), Minikube andrà a creare di default una macchina virtuale chiamata `minikube` che fungerà da contenitore del cluster: questa verrà creata e avviata quando eseguiamo per la prima volta il comando `minikube start`, e potrà anche essere spenta o cancellata utilizzando i comandi appropriati:

Listato 4.11 Gestione cluster tramite il comando minikube

```
minikube start # avvio del cluster e dell'istanza della macchina virtuale  
minikube stop # arresto della macchina virtuale  
minikube delete # cancellazione della macchina virtuale
```

Non solo: Minikube, per come lavora, genera un'istanza per ogni cluster creato: questo vuol dire che, utilizzando un parametro aggiuntivo, sarà possibile creare più di un cluster a seconda dello scopo che vogliamo raggiungere, per esempio disporre di un ambiente di test e di uno di sviluppo.

Listato 4.12 Creazione di più cluster

```
minikube start --profile test  
minikube start -p dev
```

Rancher Desktop

Rancher Desktop, ora nella versione 1.7, è un ambiente per la gestione dei container per Windows, macOS e Linux. È una soluzione basata su Kubernetes che esegue un cluster *K3s* (la versione light di Kubernetes) all'interno di una macchina virtuale. Potrai utilizzare strumenti compatibili con Kubernetes come `kubectl` e `Helm` per interagire con i suoi componenti e avviare nuovi container sia da riga di comando, che tramite l'interfaccia grafica. Un'installazione di questo tipo è l'ideale per le persone che sviluppano e che desiderano creare software containerizzato senza mantenere manualmente l'infrastruttura o avere a che fare con il terminale anche per le operazioni di deploy. L'approccio di Rancher Desktop è simile a quello di Docker Desktop, ma, a differenza del prodotto Docker, Rancher Desktop è una soluzione gratuita e open source sviluppata da SUSE, una multinazionale tedesca che da sempre vende prodotti Linux ed è fondatrice del progetto openSUSE. Rancher Desktop è stato progettato attorno a Kubernetes, mentre Docker Desktop ha implementato il supporto più avanti nel progetto, e questo lo rende più simile agli ambienti di produzione consolidati.

Per installarlo, è sufficiente andare sul sito ufficiale e scaricare l'eseguibile più adatto al proprio sistema operativo: l'interfaccia che avremo a disposizione una volta installata è molto simile a quella di Docker Desktop, e ci permette di gestire i container, la rete attraverso la quale questi comunicano e anche di configurare le impostazioni delle risorse utilizzate da Rancher Desktop, in termini di CPU e memoria.

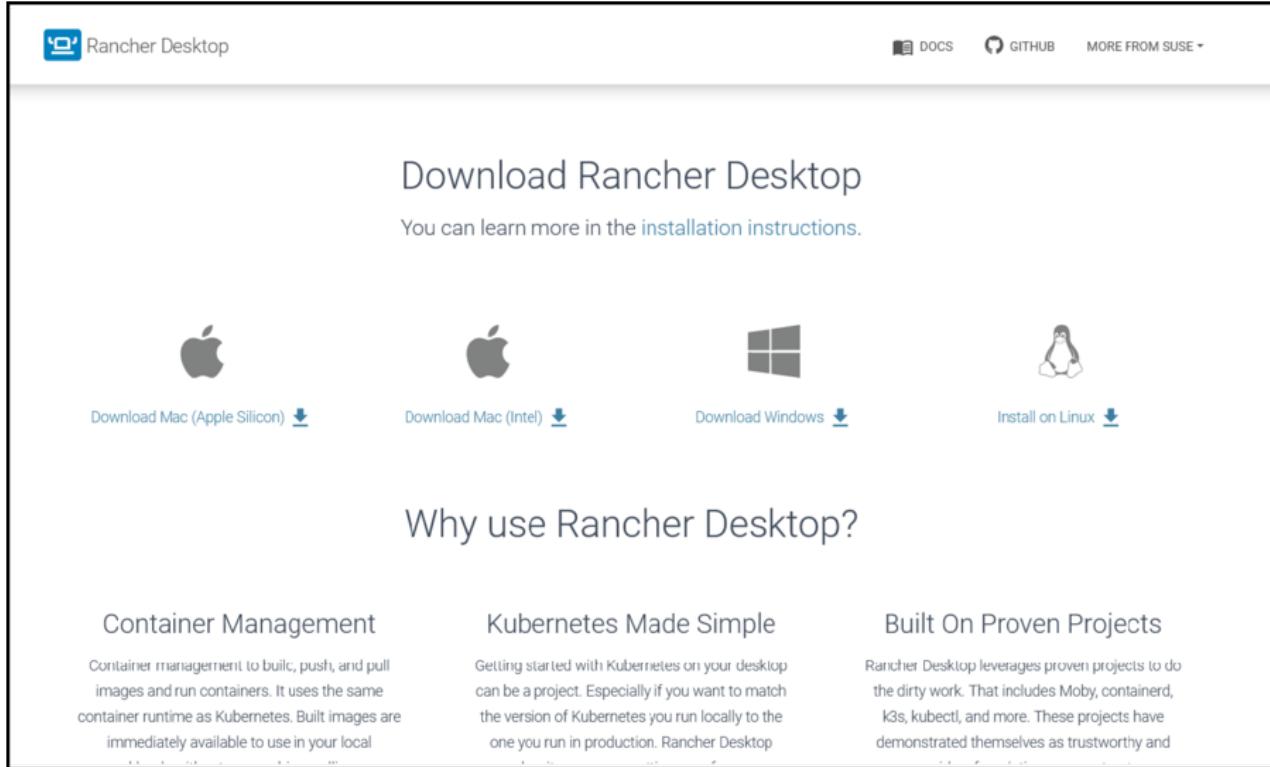


Figura 4.20 Pagina principale del sito di Rancher Desktop.

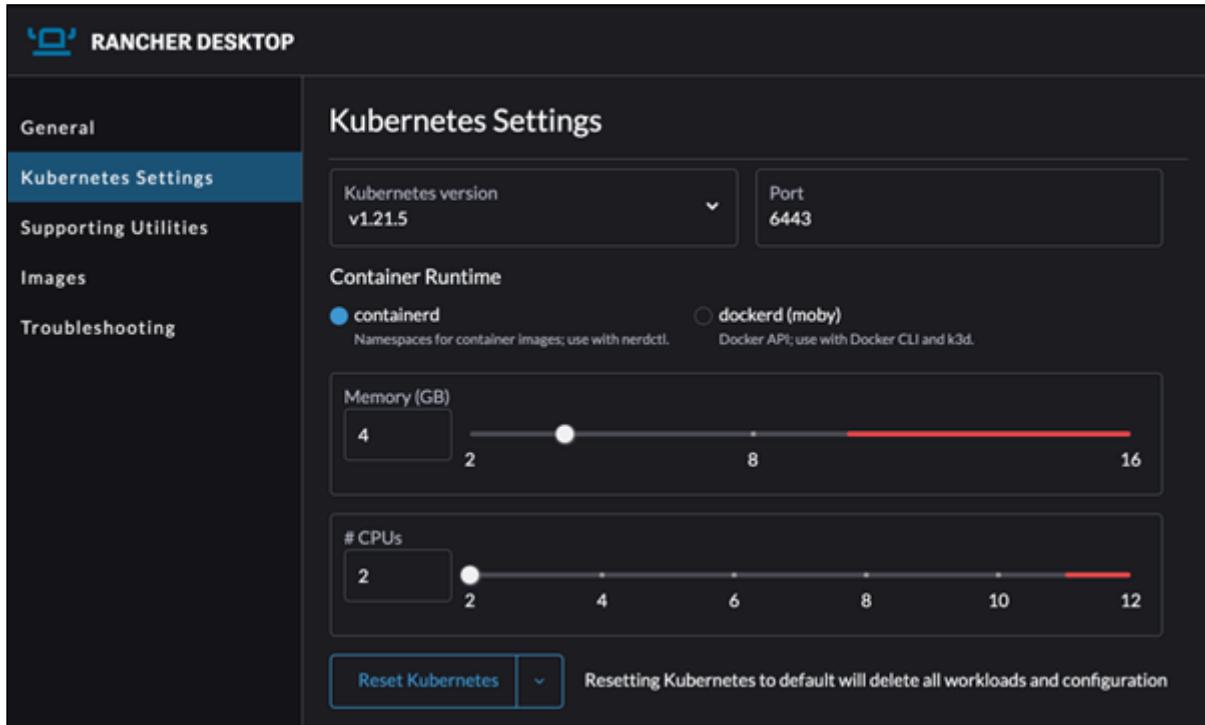


Figura 4.21 Schermata delle impostazioni di Rancher Desktop.

Hello world

Per verificare che sia tutto funzionante, prendiamo come esempio un file YAML Kubernetes per l'avvio di una semplice applicazione per iniziare, il tutorial di Docker Desktop. Non importa se al momento alcuni (o tutti) i componenti di questo file ci sono sconosciuti: sono tutti oggetti che andremo ad approfondire nel dettaglio nel prossimo capitolo!

Copiamo e incolliamo il seguente testo all'interno di un file che chiamiamo `helloworld-k8s.yaml` e poi eseguiamo il comando successivo:

Listato 4.13 Esempio di file YAML per Kubernetes

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    com.docker.project: tutorial
  name: tutorial
spec:
  replicas: 1
  selector:
    matchLabels:
      com.docker.project: tutorial
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        com.docker.project: tutorial
    spec:
      containers:
        - image: docker/getting-started
          name: tutorial
          ports:
            - containerPort: 80
      restartPolicy: Always
---
apiVersion: v1
kind: Service
metadata:
  name: tutorial
spec:
  ports:
    - name: 80-tcp
      port: 80
      protocol: TCP
      targetPort: 80
  selector:
    com.docker.project: tutorial
  type: LoadBalancer
```

Listato 4.14 Creazione degli oggetti tramite kubectl

```
$ kubectl apply -f helloworld.yaml
---
service/tutorial created
deployment.apps/tutorial created
```

Se tutto ha funzionato a dovere, ci verrà mostrato un output simile a quello riportato e potremo accedere all'applicazione appena creata sfruttando un browser qualsiasi: apriremo la pagina `localhost:80` per vedere il seguente risultato!

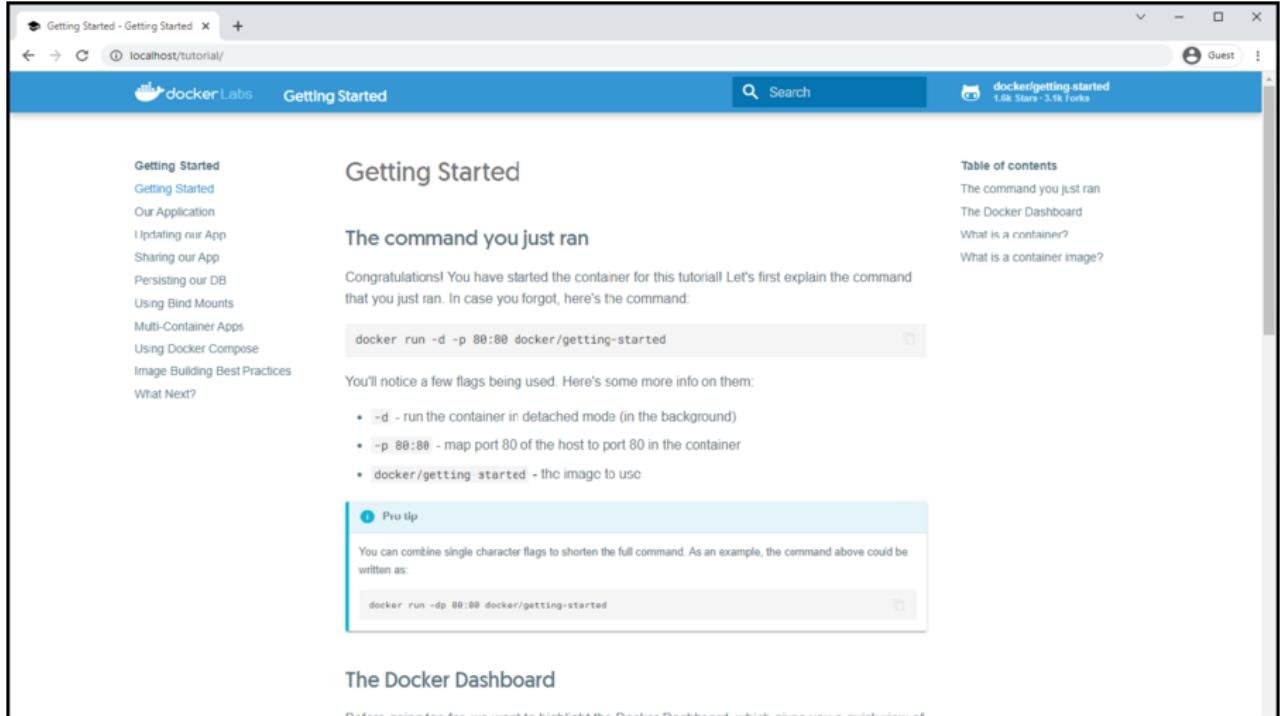


Figura 4.22 Pagina di “Hello world!” dopo l’avvio del container nel cluster Kubernetes.

Che cosa abbiamo imparato

- Quali soluzioni abbiamo a disposizione per creare un cluster Kubernetes e quali sono le principali differenze tra essi.
- Come installare Docker Desktop o Rancher Desktop, e perché preferire l’uno all’altro.
- Che cos’è Minikube, come funziona e come configurarlo.
- Che cos’è CodeReadyContainers e come utilizzarlo non solo per Kubernetes, ma anche per altre tecnologie che derivano da Kubernetes.

Kubernetes per lo sviluppo

Riconosci e abbraccia la tua unicità. Non credo che le differenze sociali cambieranno presto. Ma non credo che debba essere uno svantaggio. Essere una donna di colore, essere una donna in generale, in una squadra di soli uomini, significa che avrai una voce unica. È importante accettarlo.

– Erin Teague, Head of Product di YouTube

Come avrai modo di apprezzare, in ogni capitolo di questo libro vestiremo diversi cappelli a seconda del ruolo e del componente che spiegheremo. Questo capitolo in particolare è pensato per tutte quelle persone che sviluppano e che hanno bisogno di capire profondamente il significato di ognuno degli oggetti che questa tecnologia ci mette a disposizione: ogni sezione è pensata per spiegare, attraverso moltissimi esempi, in che modo queste risorse differiscono e come possono adattarsi ai nostri casi d'uso sotto diversi punti di vista. Per il momento, potrebbe sembrare che ognuno di questi oggetti sia slegato rispetto agli altri e che non ci sia un vero punto di connessione tra tutto ciò che vedremo: teniamo allora a mente che questa vuole essere un'introduzione agli strumenti utili nella nostra cassetta degli attrezzi, prima di dedicarci interamente a ciò che significa trasformare un'applicazione tradizionale in un'applicazione che può essere eseguita in un cluster con Kubernetes. Ma prima di passare a quelle che sono le risorse che ci permettono di descrivere e “migrare” le nostre applicazioni verso Kubernetes, partiamo da una definizione: che cosa si intende per “cluster”?

Cluster

La risposta può sembrare scontata, ma una buona fetta di persone del settore che lavora (o gravita) intorno a questa tecnologia ha qualche problema a capire che cosa sia praticamente un cluster, come funziona e quali siano i suoi limiti. Fondamentalmente, Kubernetes è un modo per gestire i container su larga scala attraverso l'automazione e la messa a punto di alcune configurazioni che con tecnologie come Docker saranno gestite manualmente. Si basa sempre sul concetto di container, i quali vengono orchestrati in cluster. Sappiamo che un container non è altro che un modo per eseguire applicazioni nel tuo computer in modo che non siano a conoscenza di altre applicazioni in esecuzione su di esso. Esistono alternative per ottenere lo stesso risultato (come le macchine virtuali), ma i container hanno la caratteristica di riutilizzare la maggior parte delle risorse sottostanti del tuo sistema operativo, quindi sono più piccoli e più veloci da creare rispetto alle alternative. Ciò significa poter eseguire più container nello stesso sistema e impiegare molto meno tempo per essere pronti al lancio.

Il concetto di container era già presente nei sistemi UNIX e Linux da molto tempo, ma sono stati resi popolari quando Docker ha introdotto il *Dockerfile* e *l'immagine*, un modo per definire che cosa c'è all'interno del container in modo da poter creare, ricostruire e condividere facilmente i container con altre persone. L'esecuzione di container su larga scala (parliamo di poche decine o di migliaia) comporta però nuove sfide che l'esecuzione a livello locale non deve affrontare. Immagina questa situazione: all'improvviso i tuoi container devono comunicare tra loro e soprattutto con l'esterno, e la rete si mette in mezzo. Questo strato richiede delle strutture aggiuntive, come la configurazione di network, e così anche l'archiviazione su sistemi distribuiti introduce di per sé nuovi problemi.

Kubernetes è una soluzione a questa sfida: migliora il *runtime* del container garantendo una serie di

funzionalità necessarie per fornire scalabilità e gestione automatica, attraverso un gruppo di nodi che funge da motore: in particolare, in questo capitolo, avremo a che fare con i nodi (chiamati in precedenza *worker*) che sono parte fondamentale di un cluster Kubernetes alla stregua dei *control-plane*. Infatti, mentre questi ultimi servono a fornire API e le interfacce per definire, distribuire e gestire il ciclo di vita dei container, oltre che la rete necessaria per connettersi ai Pod, i nodi “comuni” sono macchine fisiche o virtuali che offrono servizi per eseguire i container.

kubeconfig

Tutto in Kubernetes è descritto tramite un file YAML, e lo è anche la descrizione del cluster: esiste un file, in effetti, che si chiama `kubeconfig` e che permette di descrivere i dettagli per accedere al cluster, come l’indirizzo a cui raggiungere le API, eventuali certificati e via dicendo. Questo file è memorizzato all’interno del path `$HOME/.kube/config` e solitamente ha un aspetto simile a questo:

Listato 5.1 Esempio di kubeconfig

```
cat .kube/config
>>>
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: xxx
    server: https://kubernetes.docker.internal:6443
    name: my-cluster
contexts:
- context:
    cluster: my-cluster
    user: myuser
    name: my-cluster
current-context: my-cluster
kind: Config
preferences: {}
users:
- name: myuser
  user:
    token:
```

Questo vuol dire che si ha bisogno delle seguenti informazioni chiave per connettersi ai cluster Kubernetes:

- `certificate-authority-data`: l’Autorità Certificativa che ci garantisce l’accesso al cluster;
- `server`: l’endpoint del cluster (di uno dei nodi *control-plane*);
- `name`: nome del cluster;
- `user`: nome dell’utente che accede al cluster;
- `token`: token dell’account utente.

Nel caso in cui si stia utilizzando Docker Desktop per Kubernetes, il file potrebbe avere questo aspetto: tutte le informazioni che ci permettono la connessione al cluster vengono infatti configurate al posto nostro, così come i certificati:

Listato 5.2 Esempio di kubeconfig per Docker Desktop

```
cat .kube/config
>>>
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: xxx
    server: https://kubernetes.docker.internal:6443
```

```

name: docker-desktop
contexts:
- context:
    cluster: docker-desktop
    user: docker-desktop
    name: docker-desktop
current-context: docker-desktop
kind: Config
preferences: {}
users:
- name: docker-desktop
  user:
    client-certificate-data: xxx
    client-key-data: yyy

```

Inoltre, se ci colleghiamo a più di un cluster in diverse occasioni, ognuno di questi potrebbe riportare le proprie informazioni all'interno del `kubeconfig`: non a caso, il campo `clusters` e `contexts`, così come `users`, sono liste, che permettono la definizione di più entità.

Listato 5.3 Esempio di kubeconfig per più cluster

```

cat .kube/config
>>>
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: xxx
    server: https://kubernetes.docker.internal:6443
    name: my-cluster
- cluster:
    certificate-authority-data: xxx
    server: https://kubernetes.docker.internal:6443
    name: docker-desktop
contexts:
- context:
    cluster: my-cluster
    user: myuser
    name: my-cluster
- context:
    cluster: docker-desktop
    user: docker-desktop
    name: docker-desktop
current-context: my-cluster
kind: Config
preferences: {}
users:
- name: myuser
  user:
    token: www
- name: docker-desktop
  user:
    client-certificate-data: xxx
    client-key-data: yyy

```

In realtà, se abbiamo a che fare con un cluster diverso da quello locale, potremmo doverci collegare utilizzando delle altre metodologie: per esempio, è possibile configurare la variabile di ambiente `KUBECONFIG` del proprio terminale (o di quello di server a cui accediamo) con il percorso del file `kubeconfig` per la connessione al cluster. In questo modo, ovunque tu stia utilizzando il comando `kubectl` dal terminale, la variabile di ambiente `KUBECONFIG` dovrebbe essere disponibile. Impostando questa variabile, il contesto del cluster corrente viene sovrascritto e potrai collegarti al nuovo senza difficoltà. È possibile impostare la variabile utilizzando il seguente comando, dove `my_kubeconfig` è il nome del file `kubeconfig` che definisce le proprietà di accesso al cluster.

Listato 5.4 Configurazione di kubeconfig come variabile di ambiente

```
KUBECONFIG=$HOME/.kube/my_kubeconfig
```

Fatta questa panoramica sul cluster, possiamo prepararci a rilasciare qualche applicazione sul cluster: prendiamo fiato e iniziamo a studiare le prime risorse.

Componenti applicativi

Tutte le risorse con cui andremo a lavorare d'ora in poi e, più in generale, tutti gli oggetti del cluster, possono essere gestiti grazie al comando `kubectl`: pronunciato spesso come `kube-control`, o anche `kube-cuddle`, si tratta dello strumento con il quale possiamo comunicare con le API di Kubernetes. Ogni comando che ci permetterà di creare una risorsa, piuttosto che ottenerne l'elenco o modificarla, avrà questa istruzione come principale.

Piccola nota: la documentazione è sicuramente più esaustiva di quanto riusciremo a racchiudere nelle poche pagine di questo libro, per cui è importante tenere a mente che ogni comando e ogni istruzione che riporteremo sarà sicuramente spiegata in maniera opportuna, ma che le funzionalità di questa tecnologia sono pressoché infinite, per cui un po' di curiosità insieme al riepilogo dei comandi che avrai a disposizione alla fine del libro tramite le appendici, ti saranno sicuramente di aiuto.

Pod

Il Pod è la più piccola unità distribuibile in Kubernetes: può contenere uno o più container, anche se, il più delle volte, abbiamo solo bisogno di un unico container per Pod. In alcuni casi speciali, più di un container è incluso nello stesso Pod, come nel caso di quelli che chiamiamo *sidecar*, ma ne parleremo più avanti. Un Pod, in genere, rappresenta uno o più container che devono essere gestiti come fossero una singola applicazione stateless: questo vuol dire che, se il container va in errore, questo può essere riavviato. 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.

Stateless: che cosa significa

Un'applicazione “stateless” è un'applicazione che non dipende dall'archiviazione dei dati: l'unica cosa di cui è responsabile è il codice e altri contenuti statici ospitati su di esso. Tutto qui, nessuna modifica fatta alla configurazione, nessun file all'interno del container persistrà quando il Pod viene eliminato e questo non porterà nessuna conseguenza.

I container nello stesso Pod vengono eseguiti in un contesto condiviso, sullo stesso nodo, condividendo il namespace e la rete, così come i volumi. Il Pod è inoltre progettato come “mortale” o transiente: quando un Pod “muore”, come nel caso in cui questo avvenga a causa del controller Kubernetes che è costretto ad arrestarlo per risorse insufficienti, questo non potrà essere riavviato autonomamente. In questo caso, avremo bisogno di un oggetto di più alto livello che ci permetta di gestire lo stato dei Pod al posto nostro, ed è il caso dei controller.

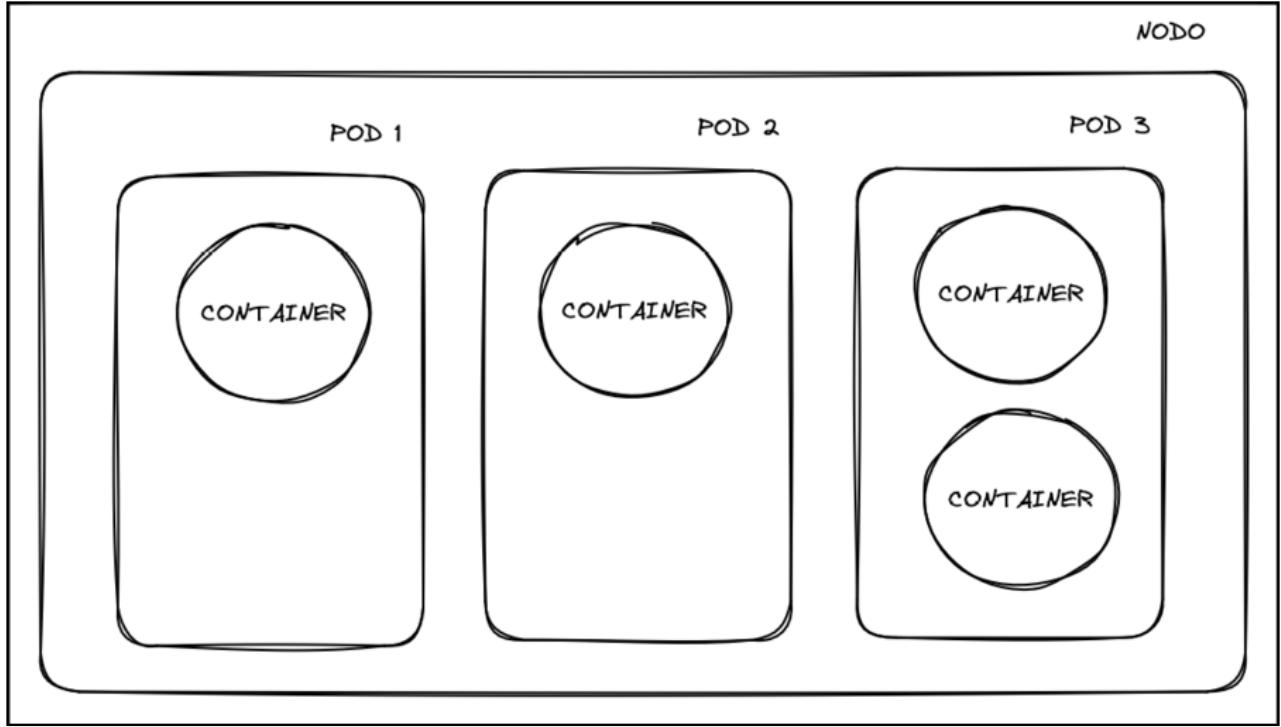


Figura 5.1 Esempio di nodo con dei Pod al suo interno.

Come sono fatti i Pod? Iniziamo a prendere confidenza con la riga di comando e, dopo aver lanciato l'ambiente che ospita il nostro cluster, sfruttiamo il terminale per scoprire di più sui Pod con il seguente comando:

Listato 5.5 Definizione della risorsa Pod

```
kubectl explain pods
>>>
KIND:      Pod
VERSION:   v1

DESCRIPTION:
Pod is a collection of containers that can run on a host. This resource is
created by clients and scheduled onto hosts.

FIELDS:
apiVersion  <string>
APIVersion defines the versioned schema of this representation of an
object. Servers should convert recognized schemas to the latest internal
value, and may reject unrecognized values. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#resources

kind <string>
Kind is a string value representing the REST resource this object
represents. Servers may infer this from the endpoint the client submits
requests to. Cannot be updated. In CamelCase. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds

metadata    <Object>
Standard object's metadata. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#metadata
```

```

spec <Object>
  Specification of the desired behavior of the pod. More info:
    https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

status <Object>
  Most recently observed status of the pod. This data may not be up to date.
    Populated by the system. Read-only. More info:
      https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

```

Con il comando `kubectl explain` abbiamo la possibilità di richiedere a Kubernetes di darci maggiori dettagli su una qualsiasi tipologia di risorsa: quello che otteniamo è infatti una breve descrizione dell'oggetto, e poi i campi che sono supportati per definire una risorsa di questo genere. Notiamo tra questi la versione dell'API di riferimento (obbligatoria per tutte le risorse), la tipologia (nel nostro caso, `Pod`), i metadata, che rappresentano le informazioni descrittive dell'oggetto, le specifiche e lo stato, che rappresentano rispettivamente le informazioni sull'immagine del container ospitato nel Pod e a che punto del ciclo di vita si trova il Pod.

Conoscere YAML

Tutti i file con cui lavoreremo, così come le risorse Kubernetes, utilizzano il linguaggio YAML; è quindi fondamentale conoscerne la sintassi che, per fortuna, è estremamente semplice. Per approfondire o rivedere come funziona, sfrutta l'Appendice C.

Cominciamo però dall'inizio, e partiamo con la creazione di un Pod: in questo caso, utilizziamo un file YAML all'interno del quale definiamo come il Pod dovrà essere messo in esecuzione e lo creiamo tramite il comando `kubectl create`.

Listato 5.6 Definizione di un Pod per Nginx

```

apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    name: nginx
    env: development
spec:
  containers:
  - name: nginx
    image: nginx

```

Creazione del Pod

```

kubectl create -f pod.yaml
>>>
pod/nginx created

```

Analizziamo un po' la definizione del Pod: abbiamo detto che per creare oggetti Kubernetes utilizzando il formato YAML ci sono alcuni campi di cui impostare i valori in maniera obbligatoria: il campo `apiVersion` è uno di questi, e la versione corrente è la `v1`. Questo vuol dire che, cercando nella documentazione ufficiale di Kubernetes, troveremo la definizione esatta di quali campi e quale tipologia di dato o valore ci si aspetta all'interno della definizione.

Usare le API

Le API delle risorse Kubernetes, così come per molte tecnologie, sono un dizionario di riferimento per scrivere correttamente la definizione degli oggetti che useremo. In questo caso, possiamo fare riferimento per tutte le risorse con cui lavoreremo in questo manuale alla documentazione presente in questo link: <https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.19>.

Riprendendo l'esempio, abbiamo poi utilizzato il campo `kind` per definire la tipologia di risorsa con cui lavoreremo: questo perché, come vedremo più avanti, esistono molti oggetti con cui possiamo avere a che fare e i campi che seguono, anche se simili nei nomi, assumono un significato diverso tra un'entità e l'altra; inoltre la tipologia permette a Kubernetes di validare l'oggetto con cui stiamo lavorando e scovare eventuali errori nella sintassi, sempre sfruttando le API.

Field	Description
<code>apiVersion</code> <code>string</code>	APIVersion defines the versioned schema of this representation of an object. Servers should convert recognized schemas to the latest internal value, and may reject unrecognized values. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#resources
<code>kind</code> <code>string</code>	Kind is a string value representing the REST resource this object represents. Servers may infer this from the endpoint the client submits requests to. Cannot be updated. In CamelCase. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds
<code>metadata</code> <code>ObjectMeta</code>	Standard object metadata.
<code>spec</code> <code>DeploymentSpec</code>	Specification of the desired behavior of the Deployment.
<code>status</code> <code>DeploymentStatus</code>	Most recently observed status of the Deployment.

Figura 5.2 Estratto della documentazione ufficiale delle API di Kubernetes.

All'interno di `metadata` inseriamo informazioni descrittive della risorsa, come il nome del Pod, eventuali label (di cui ignoriamo il significato, per il momento) e così via. All'interno del campo `spec`, invece, descriviamo gli oggetti ospitati e tutto ciò che riguarda il ciclo di vita del Pod: possiamo elencare tutti i container che vogliamo il Pod includa all'interno della sezione `containers`, così come eventuali informazioni su una rete diversa, come gestire il riavvio del container, e altro ancora. Il campo `status` è l'unico campo di cui non dovremo occuparci: questo viene compilato da Kubernetes e riporta lo stato attuale del Pod. Nella maggior parte dei casi, le persone che lavorano con queste risorse non devono modificarlo.

Dopo aver creato il Pod, possiamo sempre utilizzato `kubectl` per controllarne lo stato: con il seguente comando, è possibile recuperare l'elenco dei Pod (o delle risorse) e delle informazioni relative al loro ciclo di vita:

Listato 5.7 Elenco dei Pod

```
kubectl get pods
>>>
NAME          READY   STATUS    RESTARTS   AGE
nginx        1/1     Running   0          12s
```

Il risultato, nel caso di esempio, è del Pod creato, che risulta attivo e funzionante (colonna `ready` e `status`); la colonna `restarts` rappresenta il numero di riavvii, magari dovuti a un errore del container, e `age` rappresenta la data di creazione. Lo stato del Pod, e tutto il resto delle informazioni (eccetto il nome) vengono recuperate proprio dal campo `status` del file che descrive il Pod: infatti, utilizzando il comando `kubectl describe` possiamo ottenere i dettagli di una risorsa, compreso una fotografia dello stato, che Kubernetes ha tracciato per noi:

Listato 5.8 Descrizione del Pod nginx

```
kubectl describe pod
>>>
kubectl describe pod nginx
Name:           nginx
Namespace:      default
```

```

Priority: 0
Node: XXX
Start Time: Fri, 20 Jan 2023 17:39:53 +0100
Labels: env=development
        name=nginx
Annotations: kubernetes.io/psp: eks.privileged
Status: Running
IP: 172.31.35.108
IPs:
    IP: 172.31.35.108
Containers:
    nginx:
        Container ID: containerd://f43a57cec76b039198c6248a9cce80d3bd91736ac10dd0f2ee9262f6075426b6
        Image: nginx
        Image ID: docker.io/library/nginx@sha256:b8f2383a95879e1ae064940d9a200f67a6c79e710ed82ac42263397367e7cc4e
        Port: <none>
        Host Port: <none>
        State: Running
        Started: Fri, 20 Jan 2023 17:39:57 +0100
        Ready: True
        Restart Count: 0
        Environment: <none>
        Mounts:
            /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-pqd4j (ro)
Conditions:
    Type Status
    Initialized True
    Ready True
    ContainersReady True
    PodScheduled True
Volumes:
    kube-api-access-pqd4j:
        Type: Projected (a volume that contains injected data from multiple sources)
        TokenExpirationSeconds: 3607
        ConfigMapName: kube-root-ca.crt
        ConfigMapOptional: <nil>
        DownwardAPI: true
    QoS Class: BestEffort
    Node-Selectors: <none>
    Tolerations: node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                  node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
    Type Reason Age From Message
    ---- ---- -- -- -----
    Normal Scheduled 27s default-scheduler Successfully assigned default/nginx to XXX
    Normal Pulling 27s kubelet Pulling image "nginx"
    Normal Pulled 23s kubelet Successfully pulled image "nginx" in
4.062814862s
    Normal Created 23s kubelet Created container nginx
    Normal Started 23s kubelet Started container nginx

```

Prima di proseguire con il dettaglio del Pod, spendiamo due parole sul suo ciclo di vita: abbiamo detto che lo stato di quello appena creato nell'esempio proposto è uguale a `running`, che chiaramente rappresenta che il Pod è in esecuzione correttamente. Questo valore definisce infatti una "fase" del ciclo di vita del Pod, che non ha un intervallo di tempo limitato, ma è la sua condizione momentanea. Altre fasi che il Pod può "vivere" durante il suo utilizzo all'interno di un cluster sono lo stato `pending`, ossia un intervallo di tempo nel quale il Pod è stato accettato da Kubernetes, ma il container e la relativa immagine non sono ancora state create. Un altro stato è `failed`, che chiaramente rappresenta

la condizione in cui tutti i container presenti nel Pod sono stati arrestati, e almeno uno di essi ha riportato un errore, producendo un codice diverso da zero durante la terminazione.

Non-zero status

Uno stato di uscita diverso da zero indica un errore e viene utilizzato da sempre per differenziare una situazione in cui si presenta un errore a cui magari può essere associato un codice con un messaggio preciso. Quando un comando termina con un numero N , in Unix si utilizza un valore N come stato di uscita: in altre parole, significa che c'è stato un errore e che i log ci torneranno utili!

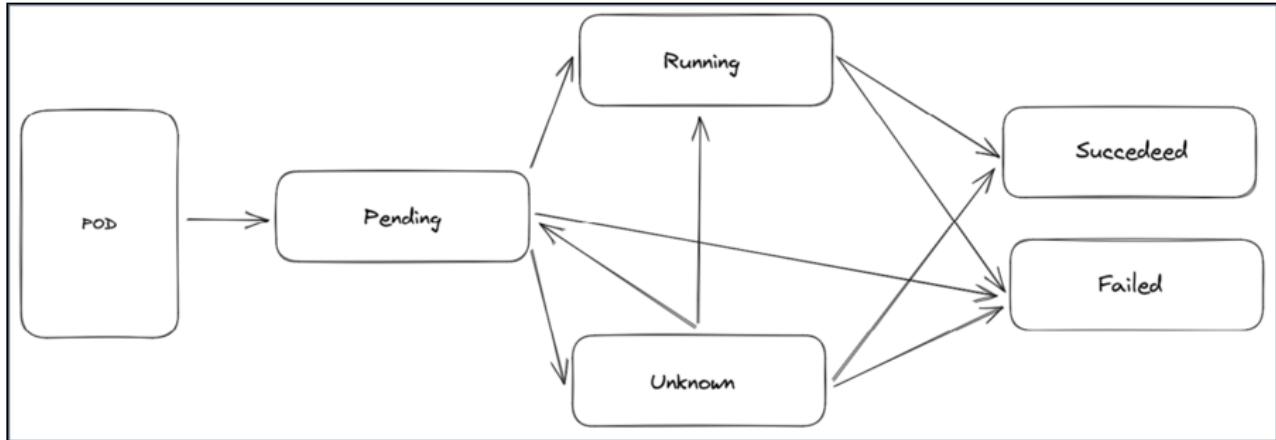


Figura 5.3 Ciclo di vita del Pod.

Che cosa succede, all'atto pratico, quando viene istanziato un Pod? Per prima cosa, questo avrà uno stato `pending`: sarà necessario infatti che lo `scheduler` trovi un nodo pronto a ospitare il Pod e `kubelet` si occuperà di scaricare l'immagine del container e di avviarlo. Questo rimarrà in stato `pending` fintanto che non saranno state predisposte queste informazioni, e poi potrà finalmente porre in stato `running` il Pod. Se il Pod venisse arrestato per qualche motivo, potrebbe non essere ricreato se non sono presenti dei controller (che vedremo in seguito) e quindi assumerebbe lo stato `failed` o `succeeded`, a seconda del codice di stato di uscita dei container: se il Pod viene arrestato e termina senza problemi, avrà come valore `succeeded`, mentre i Pod che si trovano in uno stato `pending` passano allo stato `failed` a causa, per esempio, di un errore di battitura nel nome dell'immagine o nella versione dell'immagine. Infine, se il Pod è nello stato `unknown`, significa che non è stato possibile ottenere lo stato del Pod, e questo solitamente quando c'è un problema di comunicazione con il nodo su cui questo dovrebbe risiedere o un problema con il cluster.

Un altro comando estremamente utile è quello che ci permette di mostrare i log del Pod, e quindi dei container al suo interno.

Listato 5.9 Log del Pod nginx

```
kubectl logs nginx
>>>
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform
configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of
/etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in
/etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
```

```
/docker-entrypoint.sh: Configuration complete; ready for start up
2023/01/20 16:39:57 [notice] 1#1: using the "epoll" event method
2023/01/20 16:39:57 [notice] 1#1: nginx/1.23.3
2023/01/20 16:39:57 [notice] 1#1: built by gcc 10.2.1 20210110 (Debian 10.2.1-6)
2023/01/20 16:39:57 [notice] 1#1: OS: Linux 5.4.226-129.415.amzn2.x86_64
2023/01/20 16:39:57 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2023/01/20 16:39:57 [notice] 1#1: start worker processes
2023/01/20 16:39:57 [notice] 1#1: start worker process 29
2023/01/20 16:39:57 [notice] 1#1: start worker process 30
2023/01/20 16:39:57 [notice] 1#1: start worker process 31
2023/01/20 16:39:57 [notice] 1#1: start worker process 32
...
...
```

Questo ci mostra non tanto i file di log che magari un'applicazione può produrre all'interno del container, ma piuttosto lo standard output (abbreviato `stdout`), molto simile al `docker logs`. Recuperare eventuali log è fondamentale quando le cose non stanno andando per il verso giusto e vogliamo verificare se sono presenti degli errori che ci possono aiutare a far funzionare il Pod correttamente; allo stesso modo, questo presuppone che il flusso di log sia tutto scritto nello `stdout`: questo è il metodo di gestione più semplice e più adottato per le applicazioni containerizzate, anche per quei cluster che hanno a disposizione strumenti in grado di leggere questi log e visualizzarli in un formato più leggibile, come accade con tool come Kibana.

Ma perché abbiamo bisogno dei Pod e non possiamo usare direttamente i container? Abbiamo detto che un Pod è costituito da uno o più container: questo oggetto ci permette di separare i processi, così come avviene grazie al concetto di container. Che cosa succederebbe se però avessimo più container che devono essere gestiti come una singola unità? Per quanto possa sembrare fuorviante e in contraddizione, il concetto di Pod lascia una porta aperta non solo al concetto di *sidecar*, ma a tutte quelle situazioni in cui un insieme processi strettamente correlati debbano lavorare insieme e avere (quasi) lo stesso ambiente, come se fossero tutti in esecuzione in un singolo container, pur mantenendoli in qualche modo isolati. In questo modo, si ottiene il meglio da entrambi i mondi: puoi sfruttare tutte le funzionalità fornite dai container, dando allo stesso tempo ai processi l'illusione di funzionare insieme. Quando si tratta di filesystem, le cose sono leggermente diverse: per impostazione predefinita il filesystem di ciascun container è completamente isolato dagli altri; tuttavia, è possibile condividere directory e file utilizzando il concetto di “volume”, di cui parleremo approfonditamente nel Capitolo 8.

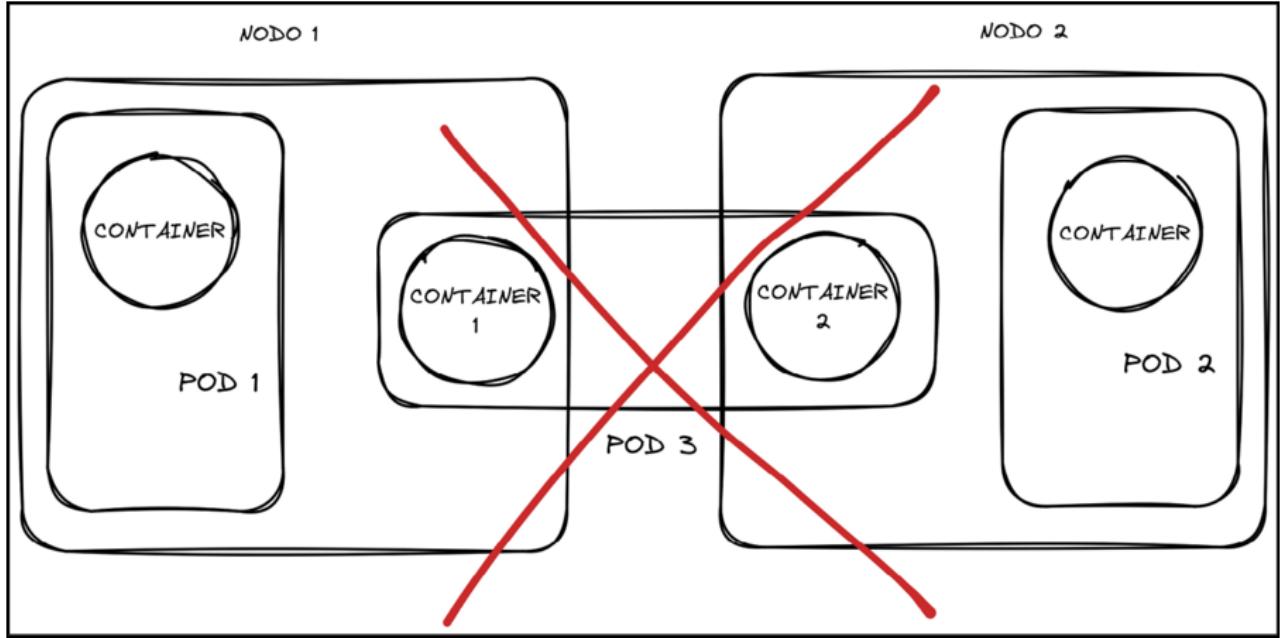


Figura 5.4 Rappresentazione di un Pod che distribuisce i container su più nodi, situazione non possibile.

L'altra nota su cui va posto l'accento è il fatto che più container all'interno dello stesso Pod condividono lo stesso spazio di rete e di conseguenza le stesse porte: questo vuol dire che i processi che sono in esecuzione in container all'interno dello stesso Pod devono prestare attenzione alle porte utilizzate, per evitare conflitti; questo significa anche che tutti i container all'interno del Pod appartengono alla stessa interfaccia di rete, per cui un container può comunicare con gli altri tramite localhost.

Sidecar container

Di solito, i Pod sono costituiti da un container *principale* che soddisfa lo scopo generale del carico di lavoro e, in alcuni casi, da alcuni container di supporto che facilitano attività strettamente correlate. Si tratta di servizi che beneficiano dell'esecuzione e della gestione nei propri container, ma sono strettamente legati all'applicazione principale. Per comprendere al meglio le ragioni che possono portare una persona a creare un Pod con più di un singolo container, proviamo a fare qualche esempio. Immaginiamo di avere un Pod con un container che esegue il server primario dell'applicazione: questo potrebbe avere necessità di avere un componente di supporto che estrae i file in una cartella condivisa quando vengono rilevate modifiche in un repository esterno. In questo caso, il container di supporto viene chiamato *sidecar container*, che in gergo significa proprio "a rimorchio": il secondo container nell'esempio è di solo sostegno al primo, e totalmente dipendente da esso.

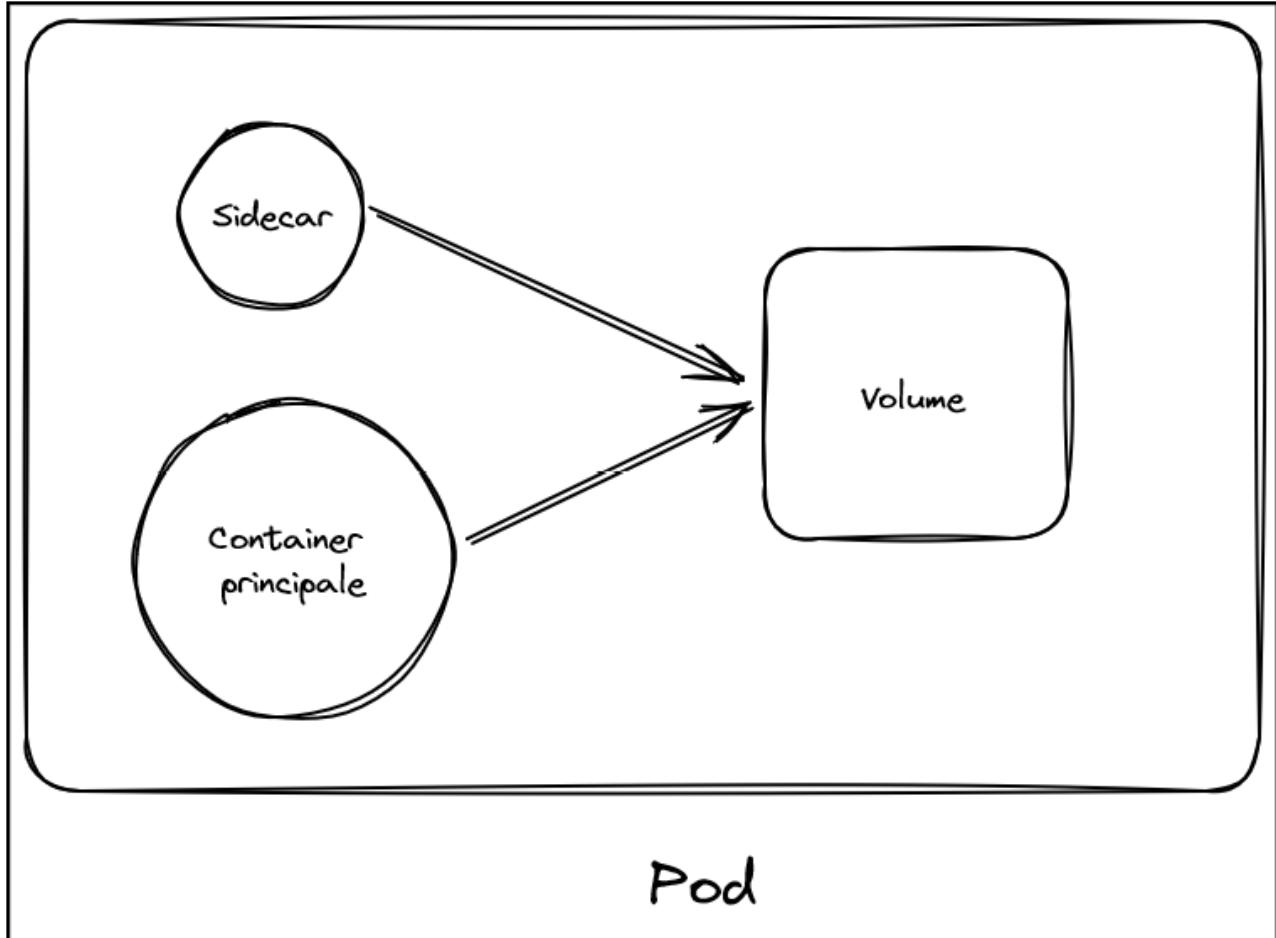


Figura 5.5 Rappresentazione di un sidecar container all'interno di un Pod.

Il caso d'uso più comune di questa tipologia di accoppiamento è il caso in cui si lavori con un web server che ha bisogno di memorizzare e gestire i log tramite un servizio apposito: il container principale conterrà quindi il server che espone l'applicazione, mentre il sidecar si occuperà del servizio relativo ai log.

Sidecar pattern

Nell'ambito del design architetturale, questo tipo di struttura ha dato il via a un modello specifico, che prende il nome del container secondario: il *sidecar pattern* è un tipo di modello in cui ci sono due container, dove il primo rappresenta il servizio applicativo, senza il quale l'applicazione non esisterebbe, mentre il secondo ha lo scopo di aggiungere valore al primo, condividendo rete e storage (Fonte: *Designing Distributed Systems* di Brendan Burns. O'Reilly Media, Inc. 2018).

Un altro esempio riguarda la sincronizzazione del codice sorgente dell'applicazione con un repository Git: in questo caso, il *sidecar container* si occupa di aggiornare il codice rispetto agli ultimi commit, al posto di utilizzare una pipeline apposita per questa attività; oppure, nel caso in cui si lavori con uno strumento come Nginx o Apache, è possibile che sia necessario modificare la configurazione di questi server per far sì che l'aggiornamento sia immediato. Utilizzare un sidecar container permette, per esempio, di leggere la configurazione del server aggiornata da un sito statico e ricaricare quella attuale del server se vengono rilevate delle modifiche. Questi sono solo alcuni degli esempi, che però sono molto specifici e che riguardano soprattutto dei compiti che vengono delegati al container secondario per poter mantenere il primo attivo e aggiornato.

Per riassumere il modo in cui si debba decidere se raggruppare dei container in un Pod, oppure in due Pod separati, è possibile porsi le seguenti domande.

- I container devono essere eseguiti insieme o possono essere eseguiti su diversi host?
- I container rappresentano un insieme unico o sono componenti indipendenti?
- I container contengono più processi con funzionalità diverse?
- I container devono essere ridimensionati insieme o singolarmente?

L'ultima domanda ci pone, infatti, davanti a un assunto importante: nel caso in cui si abbiano più container in un Pod, il ridimensionamento orizzontale è generalmente sconsigliato, perché si rischia di trattare in maniera egualitaria due container con risorse e necessità differenti, soprattutto dal momento che esistono altri oggetti di livello superiore più adatti all'attività. In altre parole, devi sempre valutare come prima opzione di separare i tuoi container in diversi Pod, a meno che un motivo specifico non richieda che facciano parte dello stesso Pod, e che quindi una delle domande poste in precedenza abbia una risposta affermativa, come riportato in maniera schematica nella Figura 5.6.

Abbiamo accennato che questa tipologia di Pod condivide lo storage: questo comporta che il filesystem a disposizione dei due container sia comune e che permetta a uno di scrivere, e all'altro di leggere, informazioni aggiornate. Un esempio molto semplice di Pod con due container che condividono lo storage è il seguente: il Pod contiene un container primario che espone una pagina HTML tramite Nginx e che ha come cartella principale la directory `/usr/share/nginx/html`, mentre il secondario ha come compito quello di creare un file `index.html` all'interno della stessa cartella principale del server Nginx. Una volta terminato il suo compito, questo si arresta:

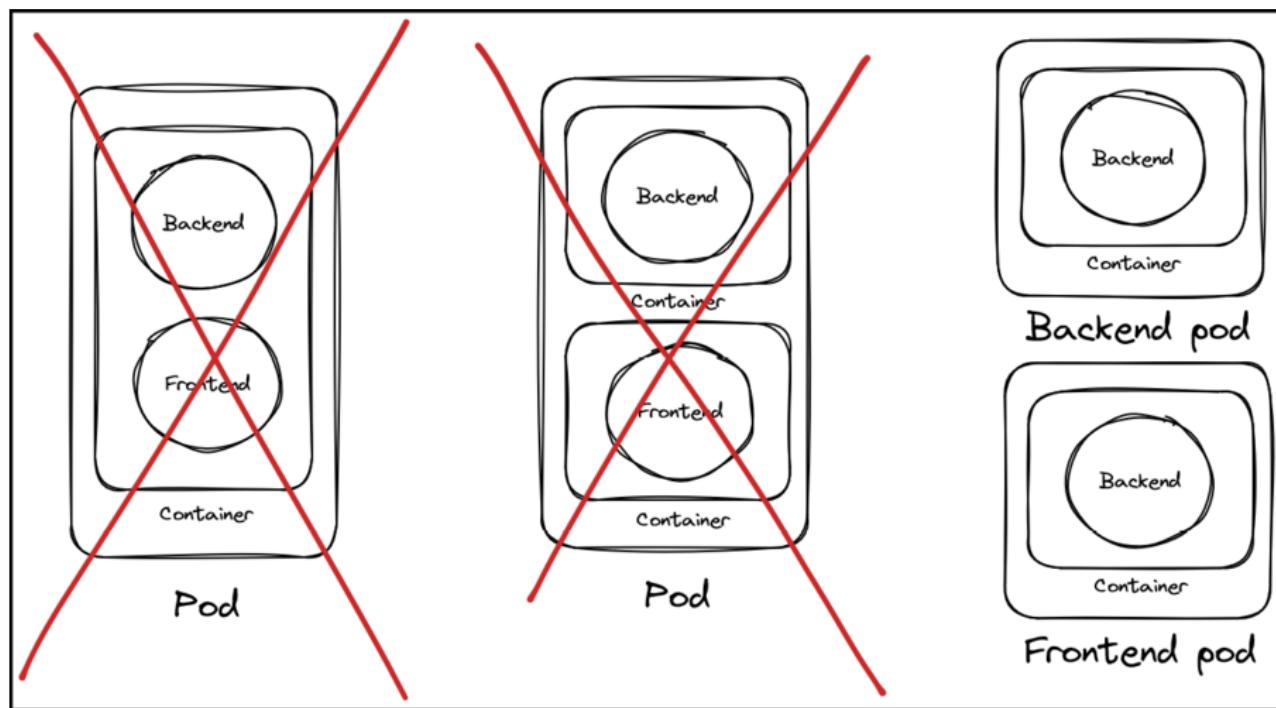


Figura 5.6 Sulla sinistra, un esempio di architettura che accoppia due microservizi all'interno dello stesso container; al centro un esempio di due container con due business logic diverse all'interno dello stesso Pod; sulla destra, un esempio di architettura ideale.

Listato 5.10 Esempio di Pod con due container

```
apiVersion: v1
kind: Pod
```

```

metadata:
  name: two-containers
spec:
  restartPolicy: Never
  volumes:
    - name: shared-data
      emptyDir: {}
  containers:
    - name: nginx-container
      image: nginx
      volumeMounts:
        - name: shared-data
          mountPath: /usr/share/nginx/html
    - name: sidecar-container
      image: debian
      volumeMounts:
        - name: shared-data
          mountPath: /my-data
      command: ["/bin/sh"]
      args: ["-c", "echo Hello from the sidecar container > /my-data/index.html"]

```

Notiamo che il container Debian ha creato il file `index.html` nella directory principale di Nginx: questo vuol dire che, usando `curl` per inviare una richiesta al server nginx, otterremo questo risultato (vedi Listato 5.11).

Listato 5.11 GET request al server Nginx

```
kubectl exec -it two-containers -c nginx-container -- /bin/bash # per accedere al container tramite terminale
```

```
curl localhost # per richiamare la pagina principale del server
>>>
Hello from the sidecar container
```

Il volume in questo esempio fornisce ai container un modo per comunicare durante il ciclo di vita del Pod; attenzione, però: come vedremo più avanti (nel Capitolo 8) se il Pod viene eliminato, tutti i dati memorizzati nel volume condiviso verranno persi.

Controller

I controller sono il cuore di Kubernetes e di qualsiasi operatore: il loro compito è quello di garantire che, per un dato oggetto, lo stato effettivo del cluster corrisponda allo stato desiderato dall'oggetto. Ogni controller si concentra su un tipo specifico di risorsa, e questo processo di verifica e gestione delle risorse allo stato desiderato viene chiamato “riconciliazione”. Vedremo a breve diverse tipologie di controller, ognuna delle quali farà del suo meglio per mantenere lo stato delle risorse come desiderato, tramite un ciclo infinito di controllo e riconciliazione delle entità che gestisce: lo vedremo in pratica nella prossima sezione, anche tramite una figura per visualizzarne il funzionamento in maniera semplice.

ReplicationController

Come già detto, i Pod rappresentano l’unità minima in Kubernetes: abbiamo visto come crearli, supervisionarli e gestirli manualmente. Ma, nei casi d’uso del mondo reale, vuoi che le tue applicazioni rimangano attive e funzionanti automaticamente, senza alcun intervento manuale e, per fare ciò, non si lavora mai direttamente con i Pod. Invece, si usano altri tipi di risorse, come i ReplicationController o i Deployment, che quindi creano e gestiscono i Pod e il loro stato di integrità.

Come Kubernetes crea un Pod

All’inizio del libro abbiamo parlato di quali componenti girano all’interno del cluster Kubernetes: ma come viene avviato un Pod? Immaginiamo una persona che, tramite il comando `kubectl` richieda la creazione di un Pod. L’istruzione

`kubectl` si collegherà al server Kube API e sarà pronto per lavorare con il cluster. Ma come funziona questo processo nel dettaglio? Il primo step consiste, da parte del componente `kube-api`, nell'autenticare l'utente e convalidare la sua richiesta per verificare che abbia i permessi per poter procedere. Una volta che questo passaggio è stato smarcato, comunica a `etcd` la richiesta e questa verrà inserita all'interno del datastore per procedere alla sua esecuzione. Nel frattempo, `kube-api` comunica all'utente che la creazione è in corso, anche se siamo ancora all'inizio del processo. Questo avviene perché sia `kube-api` che `etcd` hanno definito lo stato desiderato in cui il sistema dovrà essere. Questo vuol dire che tutti i componenti lavoreranno affinché lo stato desiderato sia quello *reale* e *attuale*, una volta terminate le operazioni necessarie al completamento della richiesta.

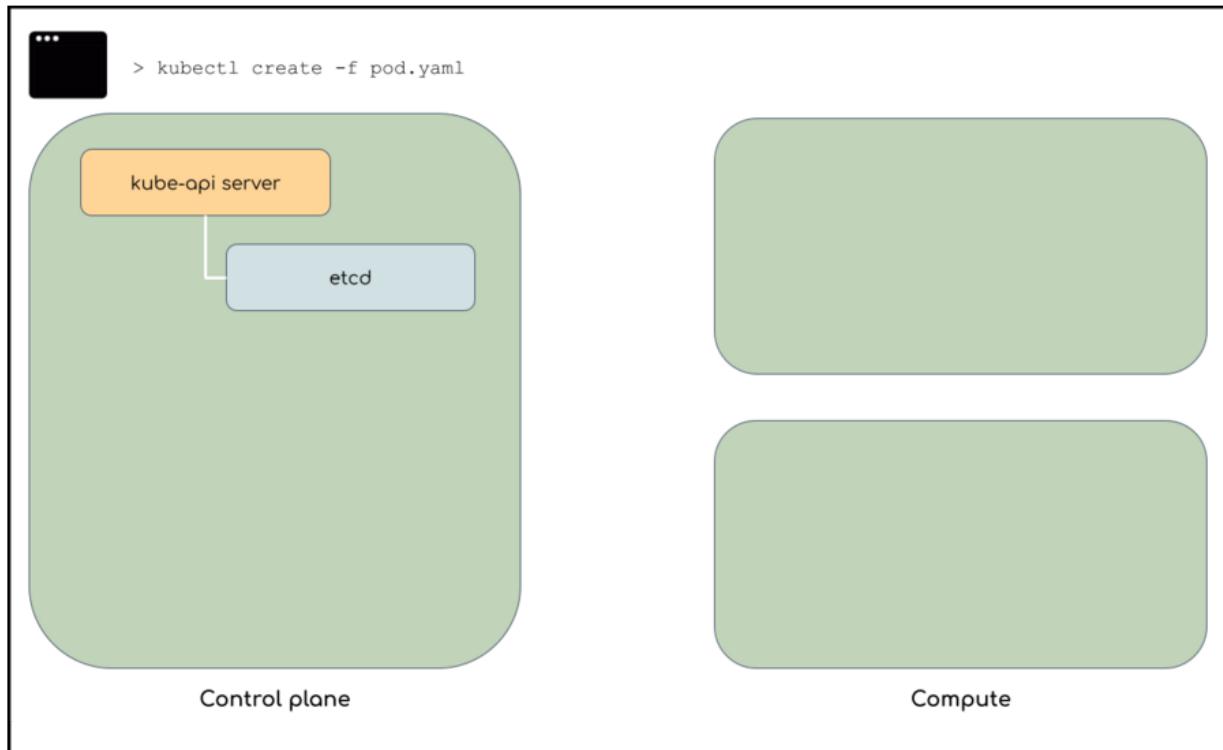


Figura 5.7 Invocazione di etcd per la creazione di un Pod.

Ora è il turno dello `scheduler`: questo si occupa di tenere d'occhio le richieste di possibili attività che devono essere eseguite e scegliere su quale nodo indirizzarle. Per farlo, richiede a `kube-api` a intervalli regolari (circa 5 secondi) se ci siano attività da svolgere e, in tal caso, si occupa della loro pianificazione.

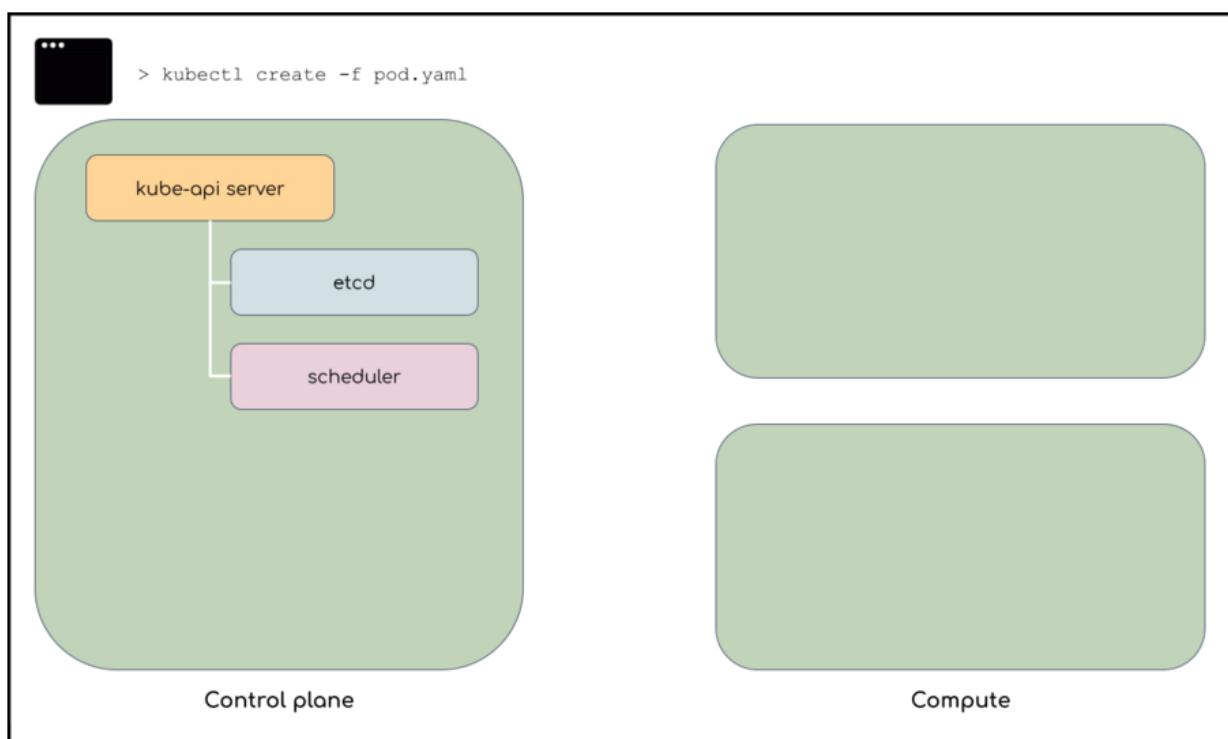


Figura 5.8 Pianificazione del Pod su un nodo del cluster tramite lo scheduler.

Una volta che questa richiesta arriva e che lo `scheduler` deve eseguirla, comunica con i nodi `worker` grazie a `kubelet`: questo componente permette infatti a nodi `control-plane` e applicativi di interagire attraverso `kube-api`. Tra i diversi compiti di `kubelet`, c'è il controllo periodico dello stato dei nodi, così da informare `kube-api` in caso di problematiche, nonché la creazione dei diversi oggetti.

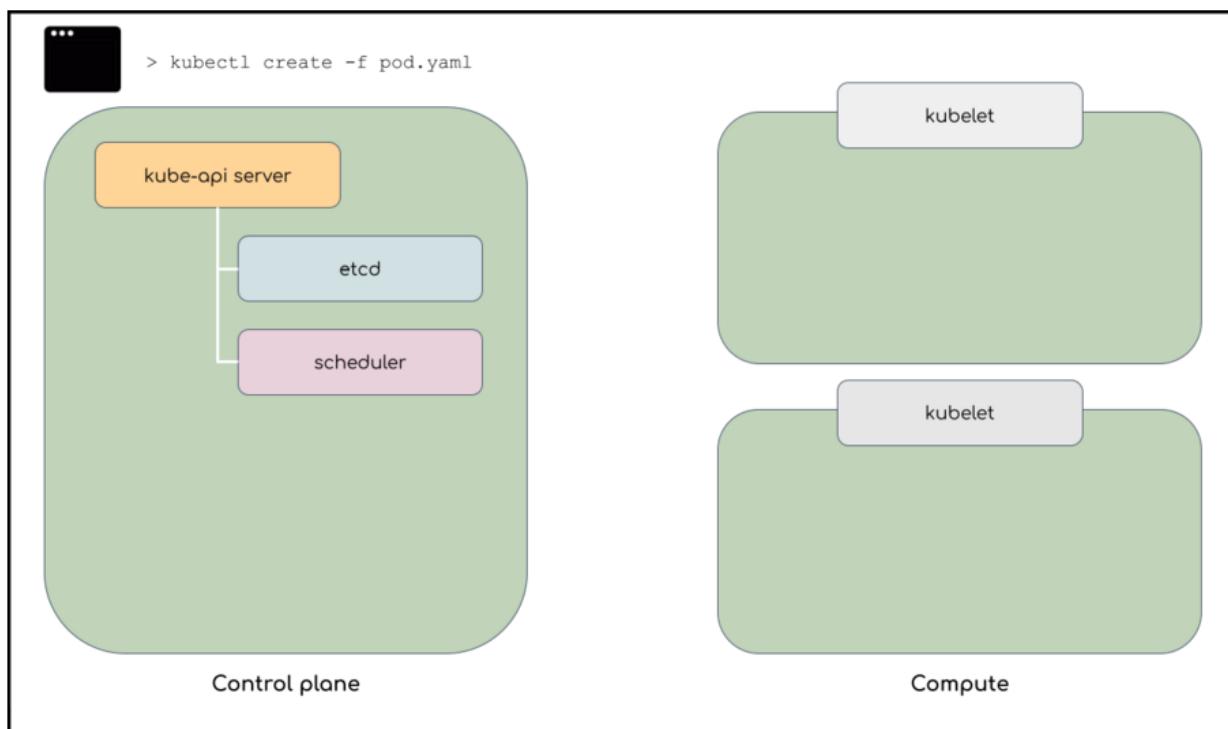


Figura 5.9 Comunicazione tra scheduler e kubelet.

All'interno dei nodi worker c'è inoltre un componente che lavorerà come motore di esecuzione dei container: questo è storicamente associato a Docker, anche se spesso si tratta di Podman e comunque può essere una delle tante tecnologie conformi alla OCI. L'ultimo componente nei nodi computazionali è `kube-proxy`: questo va citato anche se non strettamente necessario all'esempio, ma perché parte del funzionamento tra nodi: si occupa infatti di gestire la presenza di attività che coinvolgono più nodi compute all'interno del cluster.

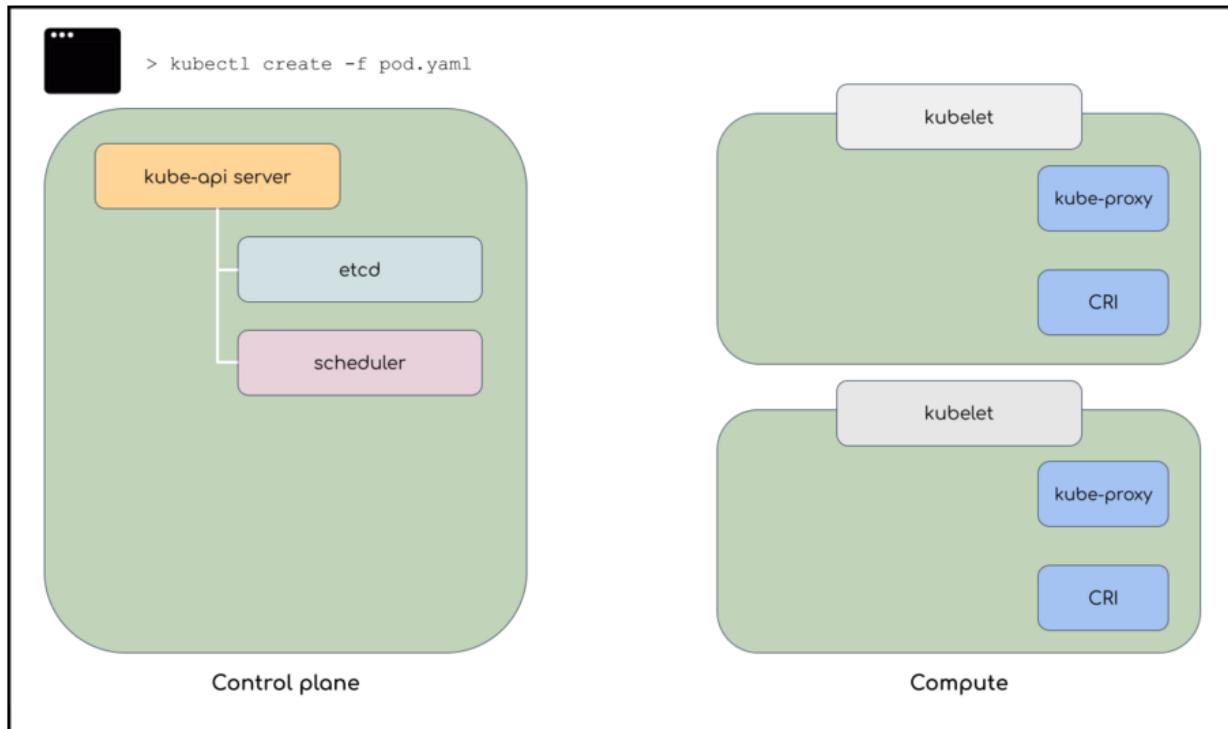


Figura 5.10 Dettaglio sui componenti dei nodi worker, che serviranno a distribuire i Pod sui nodi e renderli accessibili per la comunicazione con l'interno e l'esterno del cluster.

Tornando allo `scheduler`, questo andrà a valutare quali nodi sono disponibili, escludendo quelli che non hanno abbastanza risorse per il Pod che deve essere creato, o che magari sono stati esclusi dall'amministratore del cluster per diversi motivi.

Una volta scelto il nodo su cui eseguire l'attività, comunica la sua decisione a `kube-api`: questo, a sua volta, inserirà l'informazione su `etcd` e, una volta che lo stato desiderato sarà quindi parte del datastore, si proseguirà per far sì che questo diventi lo stato attuale.

Il server `kube-api` comunicherà quindi con il `kubelet` del nodo che si occuperà della creazione del Pod secondo la scelta eseguita dallo `scheduler`: a questo punto, `kubelet` e `CRI` (che sta per *Container Runtime Interface*) lavoreranno congiuntamente per creare il Pod.

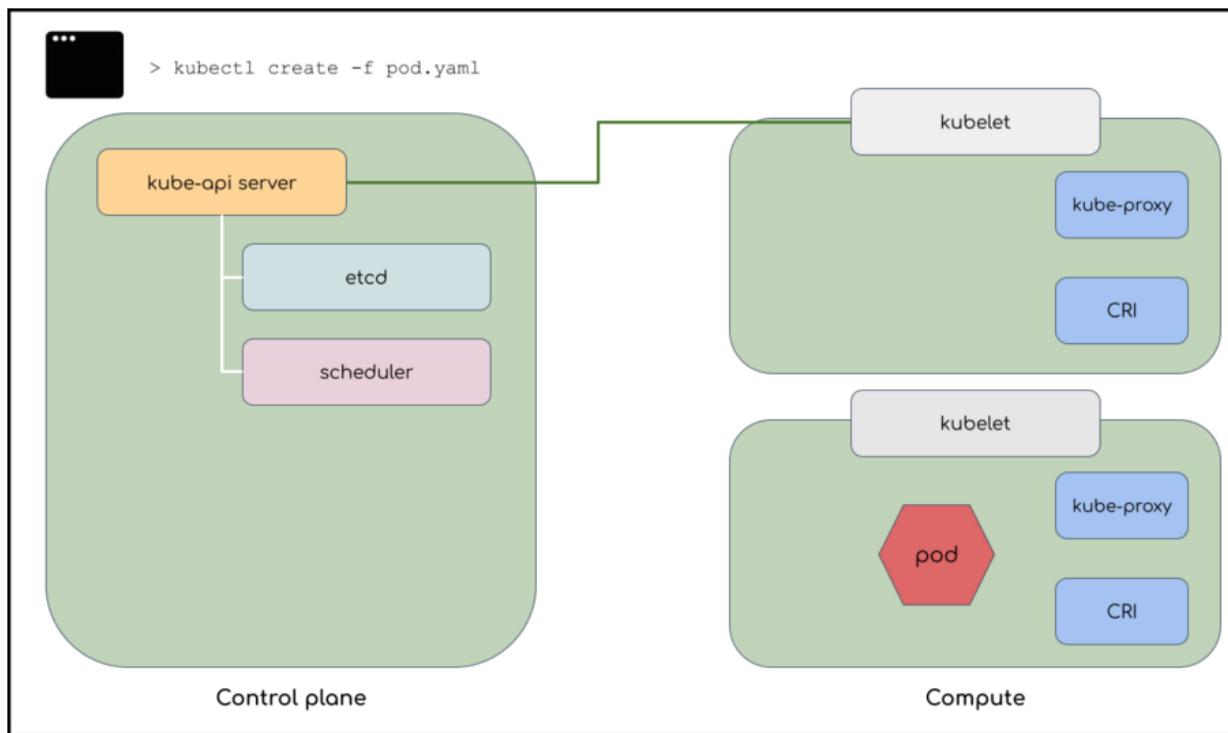


Figura 5.11 Avvio del Pod su uno dei nodi worker.

Che cosa succede se il Pod va in errore e la sua politica di riavvio prevede che venga avviato automaticamente? Entra in gioco l'ultimo componente di Kubernetes, ossia il `controller-manager`: questo è composto dall'unione dei controller, il quale, come lo scheduler, comunica con `kube-api` per tenere sotto controllo il numero di repliche presenti all'interno del cluster.

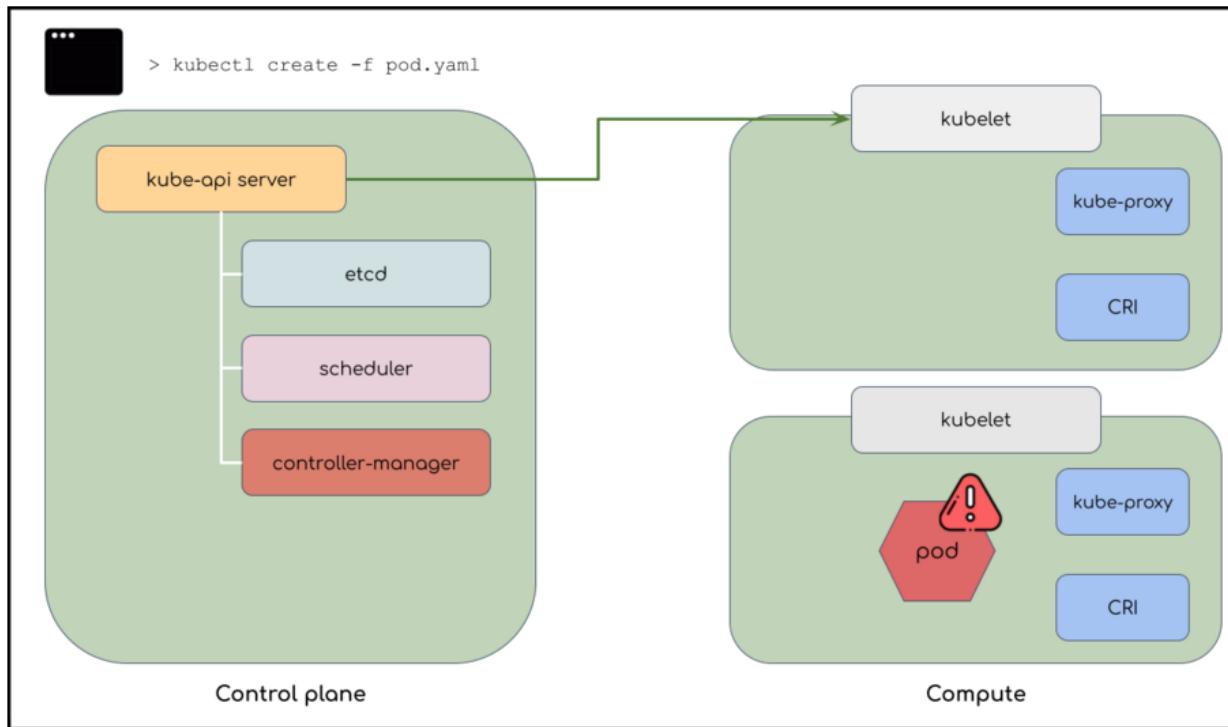


Figura 5.12 Intervento del controller-manager per gestire lo stato del Pod in errore.

In tal caso, verificando che uno dei Pod è andato in errore e che lo stato attuale e quello desiderato non corrispondono -grazie ai dati presenti in `etcd`, andrà a fare richiesta per la creazione di un nuovo Pod con le caratteristiche di quello

andato in errore.

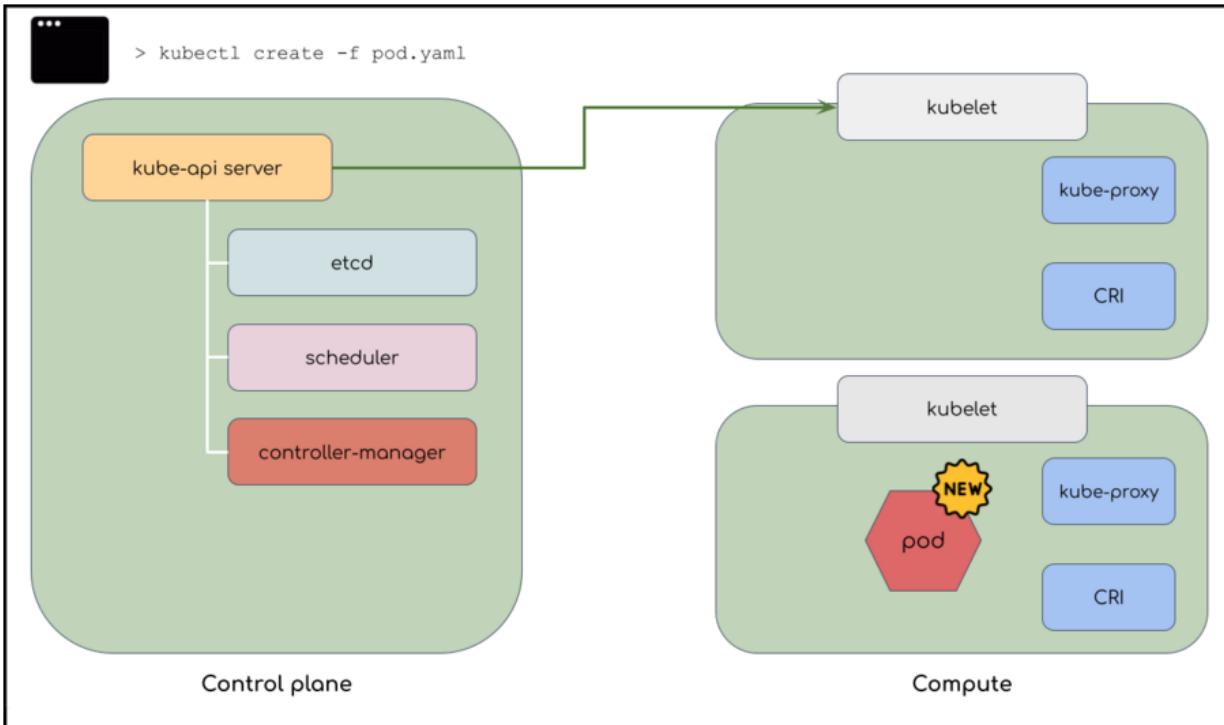


Figura 5.13 Sostituzione del Pod in errore con uno nuovo.

NOTA

In questo capitolo avremo a che fare con tantissime risorse diverse, che potrebbe essere difficile visualizzare: per facilitare la comprensione, è possibile immaginare alcune di queste come i singoli pezzi di una matrioska che andiamo a costruire pezzo per pezzo in ogni sezione. All'inizio di ogni sezione noterai anche una figura che rappresenta la dipendenza da eventuali altre entità: presto attenzione, ti aiuterà a memorizzarne meglio il ruolo.

Quelli che abbiamo menzionato prima sono infatti Pod *non gestiti*, per cui, alla loro creazione, viene selezionato un nodo del cluster per eseguire il Pod così come i suoi container; ma se il nodo subisce un errore, i Pod su quel nodo vengono persi e non verranno sostituiti con dei nuovi, a meno che tali Pod non siano in carico a una risorsa come quelle menzionate. Queste ci permettono infatti di verificare se un container è ancora attivo e riavviarlo, in caso contrario.

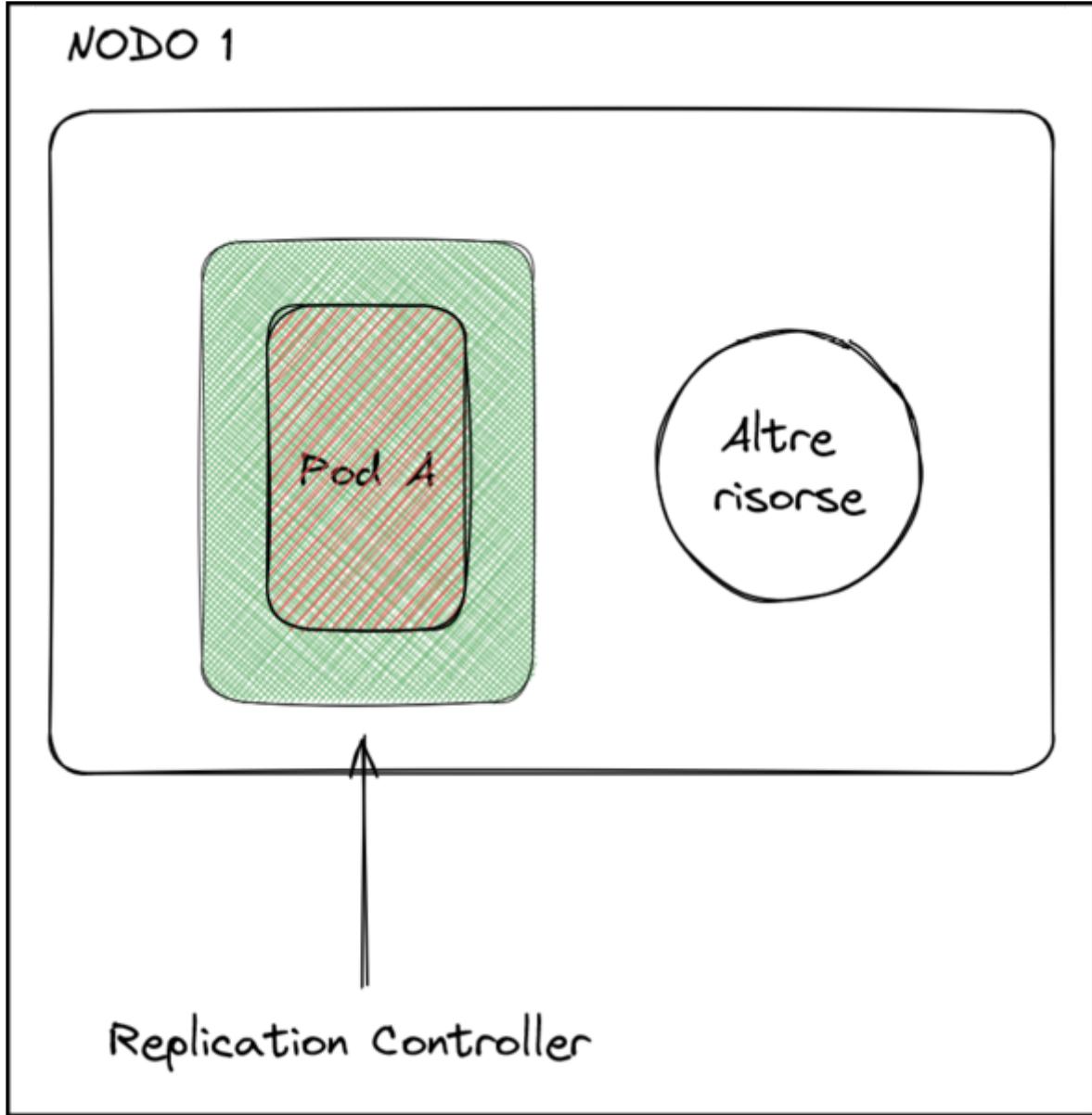


Figura 5.14 Esempio di ReplicationController in relazione a un Pod.

Cominciamo dai ReplicationController: si tratta di una risorsa il cui compito principale è quello di assicurarsi che i Pod che sono in gestione continuino a funzionare. Se uno dei Pod con cui ha a che fare si arresta per qualsiasi motivo, questo si occuperà di rimpiazzare il Pod con uno nuovo. Nella Figura 5.15 vediamo che cosa succede in uno scenario in cui uno dei nodi del cluster subisce un guasto e i Pod ospitati al suo interno devono essere riavviati dalle entità che le gestiscono: in questo caso, se il Pod A viene arrestato a causa del nodo 1 e del suo errore, il ReplicationController si occuperà di crearlo all'interno di un altro nodo.

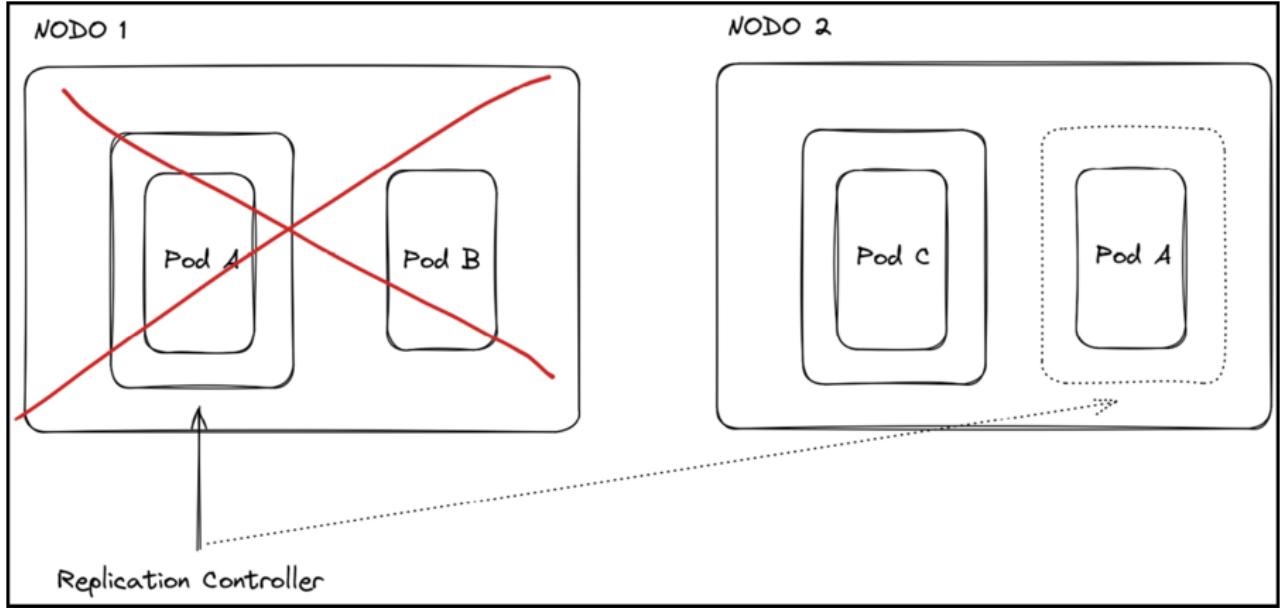


Figura 5.15 Funzionamento e responsabilità di un ReplicationController.

Un ReplicationController monitora costantemente i Pod in esecuzione per assicurarsi che il loro numero reale corrisponda a quello desiderato: se ci sono dei Pod “mancanti”, allora dovrà occuparsi di crearne di nuovi, così come gestirà la rimozione di quelli non necessari. Può venire spontaneo chiedersi: perché dovrei avere più Pod del necessario? Questo può accadere in realtà per diverse ragioni: qualcuno crea manualmente un Pod dello stesso tipo, oppure cambia la definizione di un Pod già esistente; quest’ultimo caso può avvenire per via dell’utilizzo di una label su più risorse, che va a creare una correlazione tra esse. Cerchiamo di capire in che modo: abbiamo già definito qual è il compito principale del ReplicationController, ma senza soffermarci sulle modalità con cui questo opera. In realtà, il suo lavoro è quello di trovare tutti i Pod che corrispondono a una certa label, contarli, e poi agire secondo quanto raccontato in precedenza. Potremmo quindi descrivere i diversi step che compie questa risorsa con uno schema come il seguente.

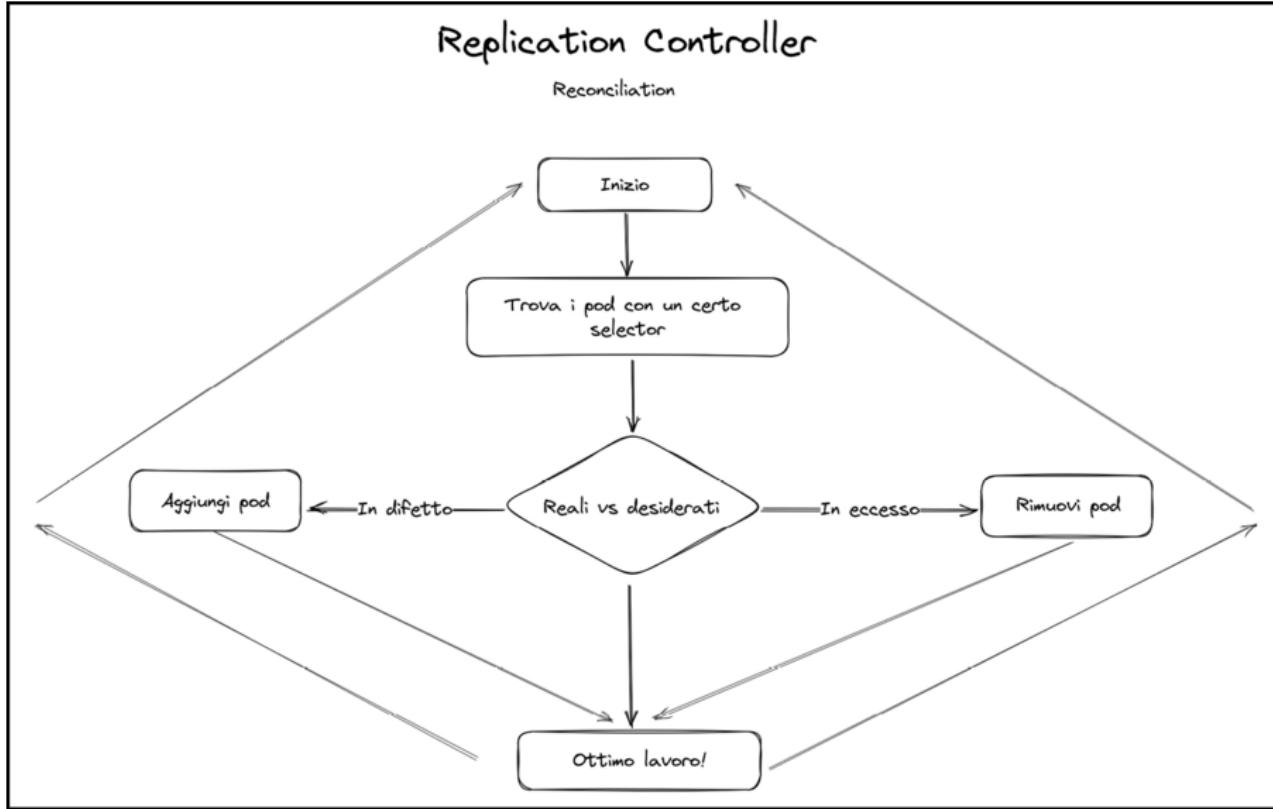


Figura 5.16 Diagramma semplificato dell'algoritmo di riconciliazione del ReplicationController.

Un esempio di ReplicationController è il seguente.

Listato 5.11 ReplicationController per Nginx

```

apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx
        ports:
          - containerPort: 80
  
```

In questo caso, abbiamo definito una risorsa che si occuperà di creare 3 repliche dell'immagine di Nginx, il cui tipo è ovviamente ReplicationController; la cosa interessante è l'utilizzo del `selector` con chiave `name` e valore `nginx`: questa farà sì che i Pod creati tramite questo oggetto avranno la stessa label e potranno quindi essere gestiti dal ReplicationController corrispondente. Che cosa succede se viene rimossa la label da uno dei Pod appena creati? Le modifiche non hanno effetto sui Pod esistenti: qualunque cosa avvenga dopo la loro creazione, come può essere il cambio della label che li

accomuna, fa sì che i Pod esistenti escano dall'ambito del ReplicationController, che quindi smette di occuparsene.

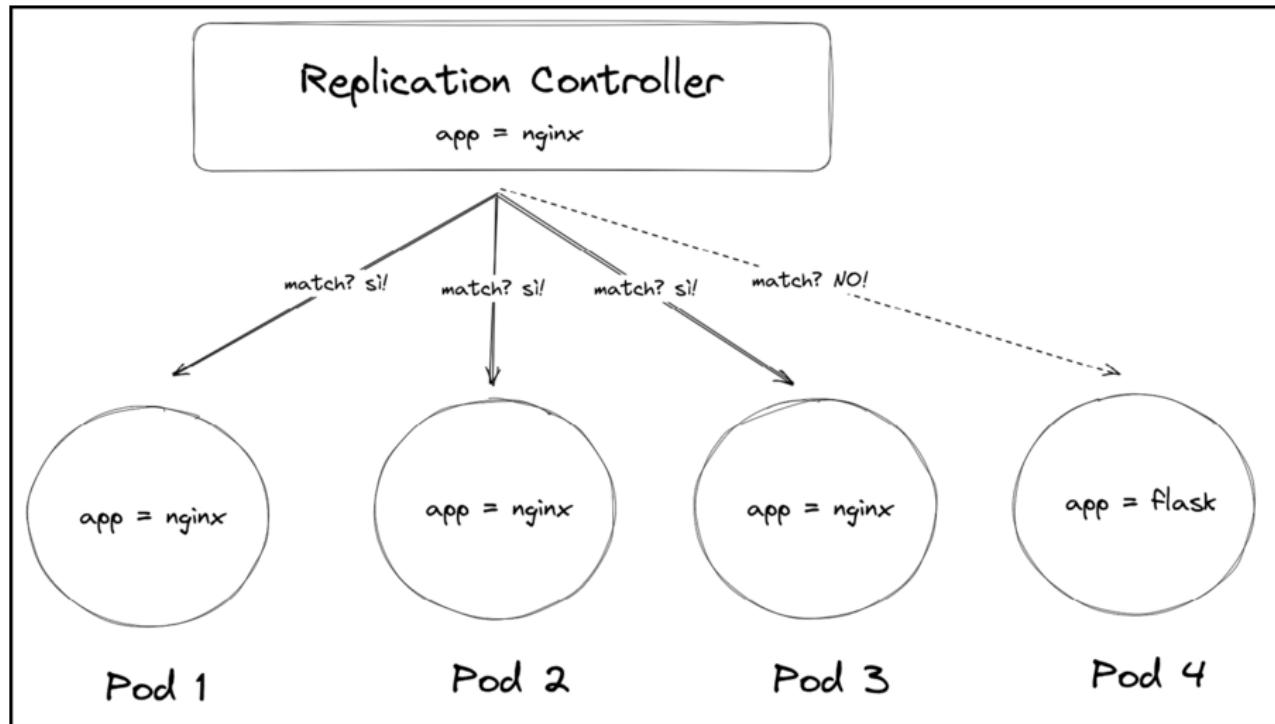


Figura 5.17 Come il ReplicationController “riconosce” i suoi Pod.

Sebbene un Pod non sia legato a un ReplicationController in alcun modo (piuttosto, è il ReplicationController che conosce i suoi Pod), il Pod vi fa riferimento nel campo `metadata.ownerReferences`, che è possibile utilizzare per trovare facilmente a quale ReplicationController appartiene un Pod. Questo porta anche a un'altra conseguenza: se cambiamo il valore di un `selector` del ReplicationController, tutti i Pod in esecuzione a esso associati vengono “svincolati” e non sono più sotto la sua gestione.

Listato 5.12 Esempio di Pod con il riferimento al ReplicationController

```
kind: Pod
apiVersion: v1
metadata:
  name: nginx-11-xxxx
...
  ownerReferences:
    - apiVersion: v1
      kind: ReplicationController
      name: nginx
      uid: d7ce9080-c65b-4ab3-89db-f6d992ef4c41
...
```

Come molte cose in Kubernetes, un ReplicationController, sebbene sia un concetto incredibilmente semplice, fornisce le seguenti potenti funzionalità: si assicura che un Pod (o più Pod) sia sempre in esecuzione avviandone uno nuovo quando ne manca uno esistente e, quando un nodo del cluster va in errore, crea delle repliche per sostituire tutti i Pod in esecuzione sul nodo in errore (quelli che erano sotto il suo controllo, ovviamente!).

Una volta creato l'oggetto presente nell'esempio riportato in precedenza, possiamo riportare l'elenco dei Pod creati.

Listato 5.13 Elenco dei Pod

```
kubectl get pods  
>>>  
NAME READY STATUS RESTARTS AGE  
nginx-33thy 0/1 ContainerCreating 0 2s  
nginx-b0xz9 0/1 ContainerCreating 0 2s  
nginx-q1vkg 0/1 ContainerCreating 0 2s
```

Se proviamo a cancellarne uno, vedremo il ReplicationController entrare in azione e creare immediatamente un altro:

Listato 5.14 Cancellazione di un Pod

```
kubectl delete pod nginx-33thy  
>>>  
pod "nginx-33thy" deleted
```

Listato 5.15 Elenco dei Pod aggiornato

```
kubectl get pods  
>>>  
NAME READY STATUS RESTARTS AGE  
nginx-33thy 1/1 Terminating 0 17s  
nginx-o19tm 0/1 ContainerCreating 0 4s  
nginx-b0xz9 1/1 Running 0 17s  
nginx-q1vkg 1/1 Running 0 17s
```

Il ReplicationController ha svolto egregiamente il suo lavoro: è un ottimo aiutante per il cluster!

Per ottenere l'elenco dei ReplicationController, possiamo usare il comando seguente, che ci mostrerà non solo le repliche dei Pod desiderate, ma anche il numero attuale e quello reale:

Listato 5.16 Elenco dei ReplicationController

```
kubectl get rc  
>>>  
NAME DESIRED CURRENT READY AGE  
nginx 3 2 3m
```

Abbreviazioni

Nel comando precedente abbiamo usato `rc`, che sta per `ReplicationController`: avremmo potuto scrivere il nome del controller per intero, ma per risparmiare tempo prenderemo confidenza con queste abbreviazioni, soprattutto quando avremo a che fare con nomi particolarmente lunghi.

Immaginiamo di voler modificare il numero di repliche impostandolo a 4: come fare? Possiamo utilizzare il comando `kubectl edit` per modificare qualsiasi informazione relativa al ReplicationController. Questo comando aprirà un editor di testo che ci permetterà di eseguire tutte le modifiche che vogliamo, salvare il nostro lavoro e poi applicarlo. Attenzione, però: l'editor di testo che apre è in effetti `vi`: per chi non lo conoscesse, si tratta di uno degli strumenti più amati (e odiati) dalle persone che lavorano con Unix, perché richiede diverse conoscenze sui comandi che ci permettono di entrare, salvare e uscire dall'editor.

Se invece volete utilizzare un editor diverso da `vi`, è possibile specificare con questo comando quello da utilizzare:

Listato 5.17 Utilizzare nano al posto di vi

```
export KUBE_EDITOR="/usr/bin/nano"
```

Torniamo alla modifica del file: oltre a poter manipolare il ReplicationController tramite il suo file YAML, possiamo cambiare il numero di repliche usando un semplice comando: `kubectl scale rc`.

Listato 5.18 Modifica del numero di repliche del ReplicationController a 4 - comando

```
kubectl scale rc nginx --replicas=4
```

Questo comando equivale alla modifica del file YAML, dove apriamo il file e, nella proprietà `replicas`, cambiamo il numero da 3 a 4:

Listato 5.19 Modifica del numero di repliche del ReplicationController a 4 - file

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 4
  selector:
    app: kubia
```

Grazie a quanto visto finora, sappiamo che, quando elimini un ReplicationController tramite `kubectl delete`, vengono eliminati anche i Pod. Tuttavia, poiché i Pod creati da un ReplicationController non sono parte integrante di esso e sono solo gestiti da esso, è possibile eliminare il ReplicationController e lasciare comunque i Pod in esecuzione. Questo può essere utile quando hai inizialmente un insieme di Pod gestito da un ReplicationController, e poi decidi di sostituire il ReplicationController con un ReplicaSet, per esempio!

Questo può essere fatto utilizzando l'opzione `--cascade=false`, che rappresenta appunto l'applicazione degli effetti sul cambio di stato del ReplicationController anche ai suoi Pod:

Listato 5.20 Cancellazione del ReplicationController, ma non dei Pod

```
kubectl delete rc nginx --cascade=false
```

ReplicaSet

Inizialmente, i ReplicationController erano l'unico componente Kubernetes creato per replicare i Pod e gestirli in caso di errore sui nodi. Successivamente, è stata introdotta una risorsa simile chiamata ReplicaSet, che rappresenta una nuova generazione di ReplicationController, che (probabilmente) presto sarà deprecata. So cosa starai pensando: avresti potuto iniziare a lavorare direttamente creando un ReplicaSet invece di un ReplicationController; la realtà è che questi oggetti a oggi sono ancora molto utilizzati, per cui è bene che tu li conosca. Detto questo, la cosa migliore è preferire sempre i ReplicaSet d'ora in poi, in quanto sono quasi identici, quindi non dovresti avere problemi a usarli.

Un ReplicaSet si comporta infatti nello stesso modo di un ReplicationController, ma permette una gestione dei selettori dei Pod più potente: mentre in un ReplicationController si può avere solo una corrispondenza esatta di una certa label, in un ReplicaSet è possibile anche dei Pod la cui label non ha un valore specifico, ma sfruttare la chiave.

Per esempio, un ReplicationController non può gestire contemporaneamente i Pod con l'etichetta `env=production` e quelli con l'etichetta `env=dev`, ma solo uno dei due; un ReplicaSet può invece specificare a entrambi i selettori e trattarli come un unico gruppo. Come vediamo nel seguente esempio, le differenze nel template tra i due non esistono, mentre la differenza principale sta nella gestione dei selettori:

Listato 5.21 Esempio di ReplicaSet

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx
```

```

labels:
  app: my-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
template:
  metadata:
    labels:
      app: my-app
  spec:
    containers:
      - name: nginx
        image: nginx
        ports:
          - containerPort: 80

```

Abbiamo detto che la differenza sta nella gestione dei selettori: in questo caso, abbiamo utilizzato `matchLabels` per poter specificare chiave e valore con cui fare il match; è possibile però anche utilizzare delle espressioni che ci permettano di raggruppare più Pod con label diverse, utilizzando `matchExpressions` insieme ad alcune proprietà speciali:

Listato 5.22 Esempio di ReplicaSet

```

apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx
  labels:
    app: my-app
spec:
  replicas: 3
  selector:
    matchExpressions:
    - key: app
      operator: In
      values:
        - my-app
  template:
    metadata:
      labels:
        app: my-app
  spec:
    containers:
      - name: nginx
        image: nginx
        ports:
          - containerPort: 80

```

È possibile intuire che, tramite questa proprietà, potremo esprimere anche altre tipologie di condizioni, come l'esistenza di una chiave o di un valore:

Listato 5.23 Esempio di ReplicaSet che matcha un insieme di valori riferiti a una chiave

```

apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: frontend
spec:
  replicas: 3
  selector:
  matchExpressions:
    - {key: stage, operator: In, values: [frontend, backend]}
  template:

```

```
metadata:  
  labels:  
    app: my-webapp  
  stage: frontend
```

Listato 5.24 Esempio di ReplicaSet che NON deve matchare un insieme di valori riferiti a una chiave

```
apiVersion: extensions/v1beta1  
kind: ReplicaSet  
metadata:  
  name: frontend  
spec:  
  replicas: 3  
  selector:  
  matchExpressions:  
    - {key: stage, operator: NotIn, values: [dev]}  
template:  
  metadata:  
    labels:  
      app: my-webapp  
  stage: frontend
```

Listato 5.25 Esempio di ReplicaSet la cui chiave deve esistere

```
apiVersion: extensions/v1beta1  
kind: ReplicaSet  
metadata:  
  name: frontend  
spec:  
  replicas: 3  
  selector:  
  matchExpressions:  
    - {key: stage, operator: Exists}  
template:  
  metadata:  
    labels:  
      app: my-webapp  
  stage: frontend
```

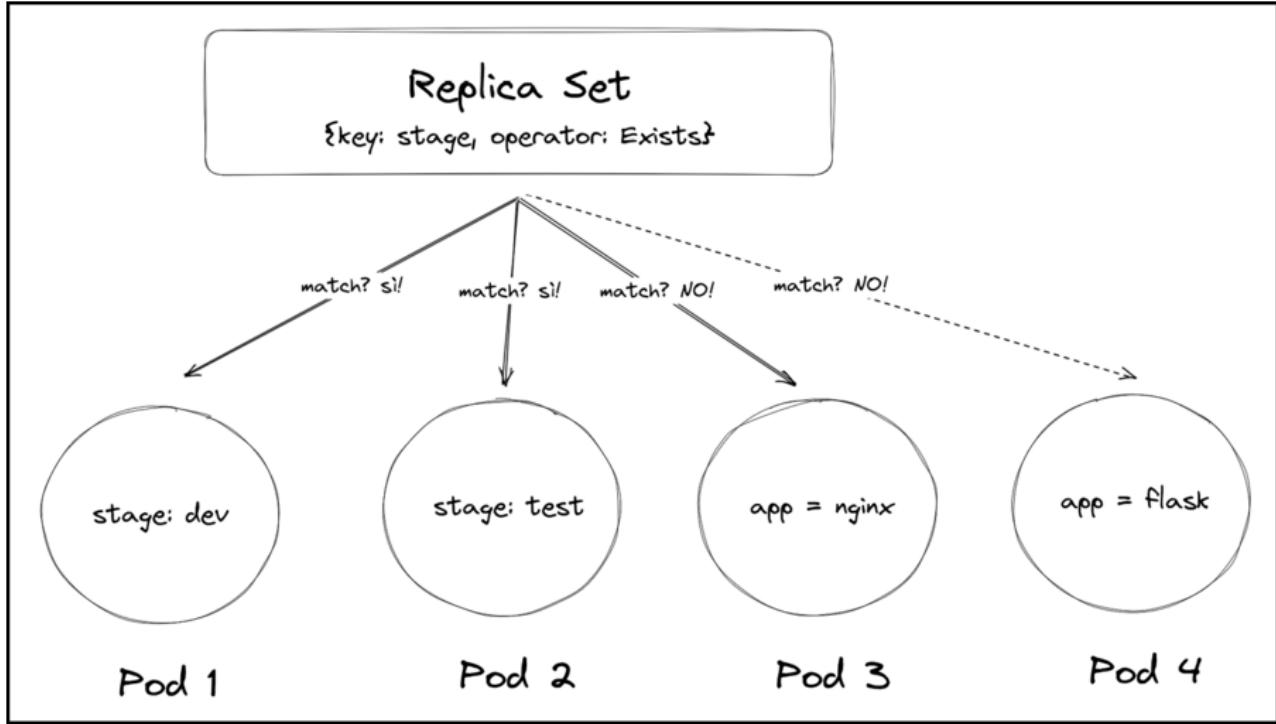


Figura 5.18 Funzionamento dell'operator “Exists” all'interno della definizione di un ReplicaSet.

Se vengono specificate più espressioni, tutte quelle espressioni devono essere soddisfatte affinché il selettor corrisponda a un Pod. Se vengono specificati sia `matchLabels` che `matchExpressions`, tutte le label devono corrispondere e tutte le espressioni devono restituire “vero” affinché il Pod sia gestito correttamente.

Deployment

Finora abbiamo visto alcuni esempi di come introdurre la nostra applicazione nel mondo di Kubernetes, creando un Pod con uno o più container, e come gestire la sua replicazione per garantire che il carico di lavoro in arrivo sia ripartito tra i Pod a disposizione. Come sottolineato in diverse occasioni, nel caso in cui un Pod venga rimosso o arrestato, eventuali modifiche verranno perse (a meno che non sia gestita la loro persistenza, ma è ancora presto per parlarne): tutti questi esempi non tengono conto del fatto che le immagini dei container potrebbero cambiare nel tempo e richiedere un aggiornamento.

Il Deployment è una risorsa la cui funzione principale è proprio quella di adattarsi al cambiamento e gestire quindi il rilascio di nuove versioni dell'applicazione, attuando una migrazione quanto più semplice possibile. I Deployment consentono di passare facilmente da una versione del codice alla versione successiva tramite una fase definita “rollout”, che si occupa dell'aggiornamento dei Pod in maniera trasparente, senza tempi di disservizi o errori. Prima di perderci nei dettagli, proviamo a seguirne il comportamento e le funzionalità attraverso degli esempi pratici e partiamo dicendo che, così come succede con i ReplicaSet, i Deployment sono risorse che gestiscono i Pod, e lo fanno proprio attraverso i ReplicaSet: se ritorniamo alla rappresentazione della matroska di risorse, la situazione attuale è la seguente:

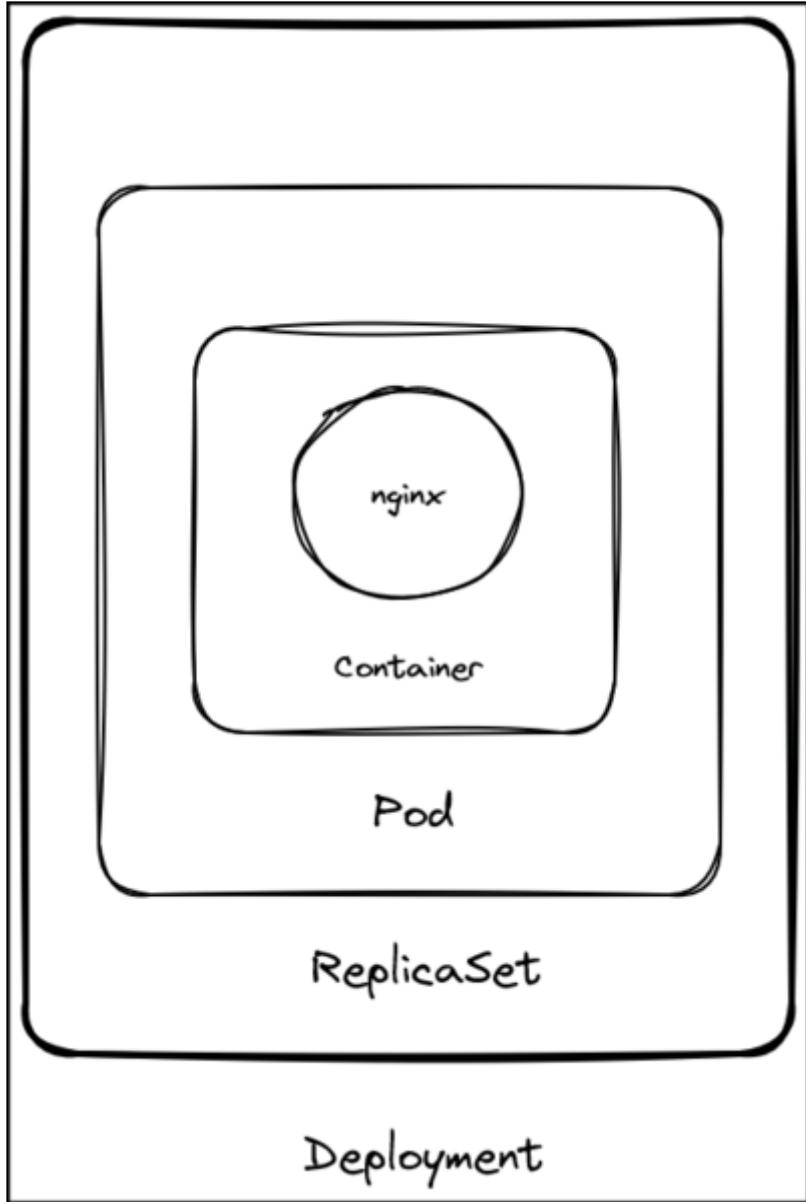


Figura 5.19 Rappresentazione di un Deployment.

Questo è l'ultimo strato della nostra astrazione, perché quello ancora superiore è rappresentato dal nodo che ospita le nostre applicazioni: questo significa che d'ora in poi, per le risorse che esploreremo, lavoreremo sempre su quest'ultimo strato, analizzando in che modo possiamo gestire i Pod e i relativi container con una serie di funzionalità che variano da oggetto a oggetto.

Un esempio di definizione di Deployment è il seguente: tornando a Nginx, vediamo come creare una risorsa che parta dalla sua immagine e crei il relativo Pod. Il Deployment avrà come nome `nginx-deployment`, mentre il numero di repliche impostato a 3 prevede quindi la creazione di tre Pod identici di Nginx; all'interno della sezione relativa ai container, andiamo a definire più dettagli: l'immagine utilizzata da Nginx sarà una delle ultime -il tag `latest` andrebbe sempre evitato, ma per questo esempio possiamo chiudere un occhio- e infine, per far sì che il web server sia raggiungibile, specifichiamo che il container è in ascolto sulla porta 80. La definizione di un Deployment ha una struttura molto simile alla specifica di ReplicaSet: esiste infatti il campo `template` che racchiude i dettagli relativi ai container descritti poco fa, come il numero di repliche o le porte esposte.

Listato 5.26 Esempio di definizione di Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80
```

Andiamo a creare il Deployment con il consueto comando, e poi passiamo a qualche osservazione:

Listato 5.27 Creazione del Deployment

```
kubectl create -f deployment.yaml
```

La prima è che, richiedendo l'elenco dei Deployment disponibili all'interno del namespace tramite il comando `kubectl get deployments`, questo dovrebbe essere il risultato:

Listato 5.28 Elenco dei Deployment

```
kubectl get deployments
>>>
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3/3     3           3           14s
```

Notiamo che nella seconda colonna sono presenti 3/3 Pod: questo vuol dire che ci saranno 3 Pod identici che girano all'interno del namespace. Ognuno di questi ha come prefisso il nome del Deployment (nell'esempio, `nginx-deployment`), più una stringa in comune tra tutti e 3 i Pod, con un suffisso rappresentato da 5 caratteri alfanumerici casuali. Quella stringa in comune non è altro che il nome assegnato al ReplicaSet, che sta gestendo al posto nostro il numero di Pod presenti nel namespace:

Listato 5.29 Elenco dei Deployment

```
kubectl get pods
>>>
NAME           READY   STATUS    RESTARTS   AGE
nginx-deployment-cd55c47f5-95hb5   1/1     Running   0          115s
nginx-deployment-cd55c47f5-cd668   1/1     Running   0          115s
nginx-deployment-cd55c47f5-d5f5t   1/1     Running   0          115s
```

Listato 5.30 Elenco dei ReplicaSets

```
kubectl get replicsets
>>>
NAME           DESIRED   CURRENT   READY   AGE
nginx-deployment-cd55c47f5   3         3         3       4m58s
```

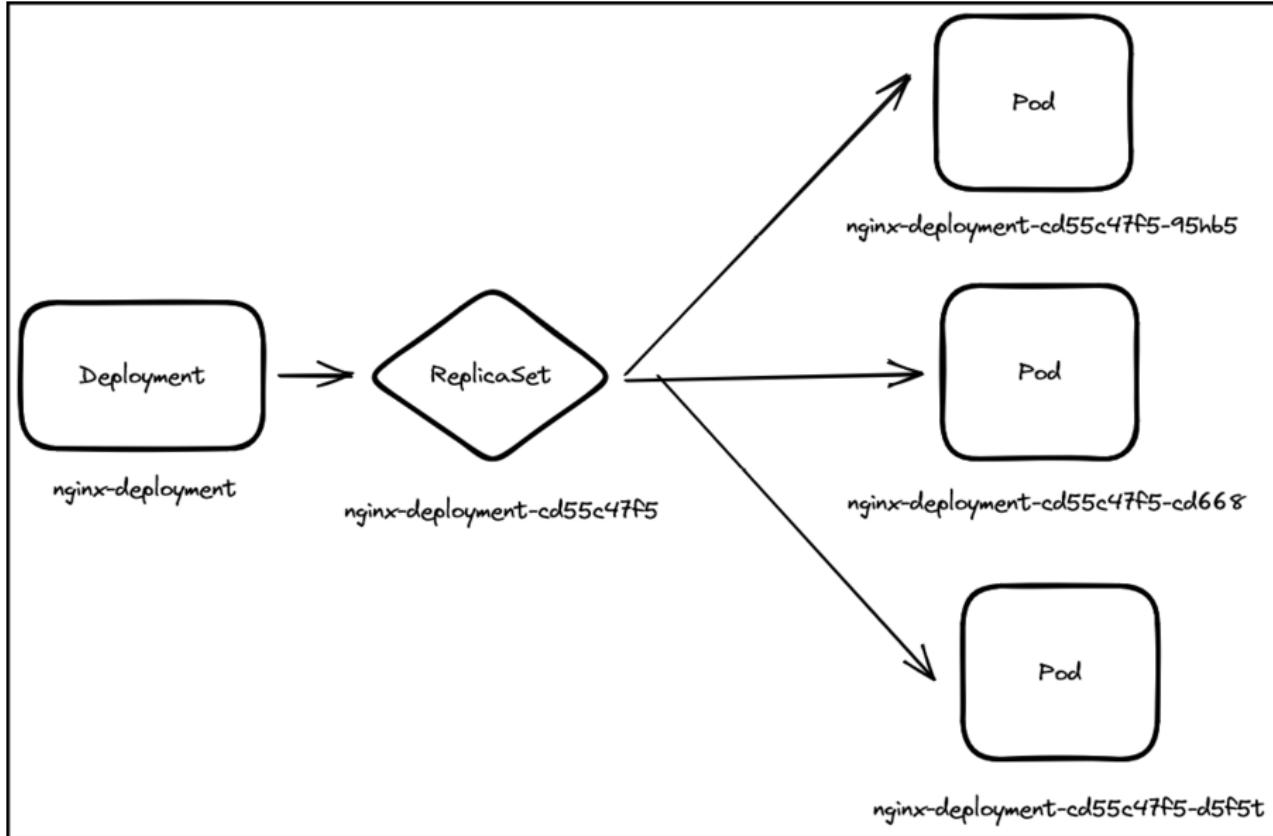


Figura 5.20 Relazione tra Deployment, ReplicaSet e Pod e tra i loro nomi.

Perché abbiamo bisogno di questi nomi casuali? Quando aggiorniamo il Deployment, il nuovo ReplicaSet verrà creato con un nuovo nome casuale, mantenendo quello vecchio per eventuali rollback. Inoltre, come visibile dalla figura, possiamo affermare che i Deployment controllano e gestiscono i ReplicaSet, e questa relazione di dipendenza è gestita tramite delle label e un selettore che accoppia queste risorse.

Listato 5.31 Selettore del ReplicaSet e del Pod

```

kubectl describe replicaset nginx-deployment-cd55c47f5
>>>
Name:           nginx-deployment-cd55c47f5
Namespace:      default
Selector:    app=nginx, pod-template-hash=cd55c47f5
Labels:         app=nginx
                pod-template-hash=cd55c47f5

---
kubectl describe pod nginx-deployment-cd55c47f5-95hb5
>>>
Name:           nginx-deployment-cd55c47f5-95hb5
Namespace:      default
Priority:      0
Service Account: default
Node:          docker-desktop/192.168.65.4
Start Time:    Sun, 05 Feb 2023 17:30:13 +0100
Labels:      app=nginx
                pod-template-hash=cd55c47f5

```

Per approfondire il rapporto tra Deployment e ReplicaSet, proviamo a scalare a 2 il numero di repliche tramite il seguente comando, e osserviamo cosa cambia nella descrizione del ReplicaSet:

Listato 5.32 Numero di repliche ridotto a 2

```
kubectl scale deployments nginx-deployment --replicas=2
>>>
deployment.apps/nginx-deployment scaled
```

Listato 5.33 Numero di repliche ridotto a 2 tramite Deployment

```
kubectl get replicsets --selector=app=nginx
>>>
NAME                DESIRED   CURRENT   READY
nginx-deployment-cd55c47f5  2         2         17m
```

Da questo esempio, puoi vedere che il Deployment sta gestendo un ReplicaSet proprio grazie alla label `app=nginx` e che questa può essere utilizzata anche all'interno di una query per specificare qual è il selettore di cui ci interessa recuperare le risorse associate. Dal momento che il Deployment è in grado di gestire tutte le risorse al suo “interno”, questo vuol dire che qualsiasi alterazione manuale di queste sarà comunque da lui gestita: un esempio è se proviamo a scalare direttamente il ReplicaSet a 1 replica, che porta al seguente risultato:

Listato 5.34 Numero di repliche ridotto a 1 tramite ReplicaSet

```
kubectl scale replicsets nginx-deployment-cd55c47f5 --replicas=1
>>>
replicaset.apps/nginx-deployment-cd55c47f5 scaled
---
kubectl get replicsets --selector=app=nginx
>>>
NAME                DESIRED   CURRENT   READY   AGE
nginx-deployment-cd55c47f5  2         2         19m
```

Quando proviamo a cambiare il numero di repliche su 1, questo non corrisponde più allo stato desiderato del Deployment, il quale ha il compito di mantenere traccia delle configurazioni dei Pod, e che ha un numero di repliche pari a 2. Non appena tentiamo di scalare a una sola replica il ReplicaSet, il Deployment se ne accorge e interviene per garantire che lo stato osservato corrisponda allo stato desiderato, in questo caso impostando nuovamente il numero di repliche a due. Per poter gestire in maniera autonoma il ReplicaSet, è chiaro che bisogna eliminare il Deployment, impostando a `false` il parametro `--cascade`, così che venga rimosso questo controller e rimanga la relazione tra ReplicaSet e Pod.

Per dare un'occhiata a quanto appena successo all'interno del Deployment, usiamo il comando `kubectl describe` e iniziamo a prendere confidenza con gli eventi:

Listato 5.35 Stato del Deployment nginx-deployment

```
kubectl describe deployments nginx-deployment
>>>
Name:                  nginx-deployment
Namespace:             default
CreationTimestamp:     Sun, 05 Feb 2023 17:30:13 +0100
Labels:                app=nginx
Annotations:           deployment.kubernetes.io/revision: 1
Selector:              app=nginx
Replicas:              2 desired | 2 updated | 2 total | 2 available | 0 unavailable
StrategyType:          RollingUpdate
MinReadySeconds:       0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
```

```

nginx:
  Image:          nginx:latest
  Port:           80/TCP
  Host Port:     0/TCP
  Environment:   <none>
Mounts:          <none>
Volumes:         <none>
Conditions:
  Type      Status  Reason
  ----      ----   -----
  Progressing    True   NewReplicaSetAvailable
  Available      True   MinimumReplicasAvailable
OldReplicaSets: <none>
NewReplicaSet:  nginx-deployment-cd55c47f5 (2/2 replicas created)
Events:
  Type      Reason     Age      From            Message
  ----      ----      ---      ----            -----
  Normal   ScalingReplicaSet 27m    deployment-controller  Scaled up replica set nginx-
deployment-cd55c47f5 to 3
  Normal   ScalingReplicaSet 10m    deployment-controller  Scaled down replica set
nginx-deployment-cd55c47f5 to 2 from 3
  Normal   ScalingReplicaSet 8m11s  deployment-controller  Scaled up replica set nginx-
deployment-cd55c47f5 to 2 from 1

```

Nell'ultima sezione riportata dall'output del comando, vediamo la sezione `Events`: questa descrive tutti i cambiamenti di stato del Deployment, compreso lo `scale down` del Deployment da 3 a 2 repliche (seconda riga) e del ReplicaSet (ultima riga), che ha attirato l'attenzione del controller e che ha fatto creare nuovamente una copia del Pod.

Ora che abbiamo visto come manipolare il numero di repliche del Deployment, vediamo un'altra delle operazioni estremamente comuni con un Deployment, ossia l'aggiornamento: supponiamo quindi di voler modificare l'immagine utilizzata dal container a una versione precisa (e non più *latest*), così da implementare una versione puntuale del web server. Riprendendo il file YAML che definisce il Deployment, modifichiamo il tag relativo all'immagine a 1.23.3:

Listato 5.36 Aggiornamento della definizione del Deployment nginx-deployment

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:1.23.3
        ports:
          - containerPort: 80

```

Prima di salvare il file, aggiungiamo anche un'annotazione all'interno della sezione `template`, questa ci permette di descrivere in poche parole quale modifica stiamo attuando all'interno del Deployment, per tenerne traccia in futuro.

Listato 5.37 Aggiunta dell'annotazione alla definizione del Deployment nginx-deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    annotations:
      kubernetes.io/change-cause: "Aggiornamento immagine alla versione 1.23.3"
...

```

Salviamo il file e applichiamo le modifiche con il comando `kubectl apply`: questo comando andrà a verificare le differenze tra i due oggetti, quello già istanziato e quello descritto nel file, per poi applicare la configurazione attuale.

Listato 5.38 Applicazione delle modifiche del Deployment nginx-deployment

```
kubectl apply -f deployment.yaml
>>>
deployment.apps/nginx-deployment configured
```

L'aspetto interessante riguarda il fatto che non dovremo distruggere e creare nulla, perché sarà il Deployment a occuparsi della creazione dei nuovi Pod e dello spengimento di quelli vecchi, portando così anche grossi benefici come l'assenza di disservizio (con un'opzione che vedremo a breve): i vecchi Pod rimarranno infatti in attività fintanto che i nuovi non saranno pronti per operare, come mostrato nella sezione relativa agli eventi. Con il comando `kubectl describe` possiamo infatti vedere che le modifiche riguardo l'immagine e l'annotazione aggiunta manualmente per descriverne gli effetti siano adesso disponibili nel Deployment appena aggiornato:

Listato 5.39 Verifica dell'aggiornamento delle modifiche del Deployment nginx-deployment

```
kubectl describe deployments nginx-deployment
>>>
Name:           nginx-deployment
Namespace:      default
CreationTimestamp: Sun, 05 Feb 2023 17:30:13 +0100
Labels:         app=nginx
Annotations:   deployment.kubernetes.io/revision: 2
Selector:       app=nginx
Replicas:      3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:  RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:       app=nginx
  Annotations: kubernetes.io/change-cause: Aggiornamento immagine alla versione 1.23.3
  Containers:
    nginx:
      Image:      nginx:1.23.3
      Port:       80/TCP
      Host Port:  0/TCP
      Environment: <none>
  Mounts:        <none>
  Volumes:      <none>
Conditions:
  Type     Status  Reason
```

```

----          ----- -----
Available      True   MinimumReplicasAvailable
Progressing    True   NewReplicaSetAvailable
OldReplicaSets: <none>
NewReplicaSet: nginx-deployment-8556cb945 (3/3 replicas created)
Events:
  Type      Reason     Age     From           Message
  ----      -----     ---     ----           -----
  Normal    ScalingReplicaSet 39m    deployment-controller  Scaled
up replica set nginx-deployment-cd55c47f5 to 3
  Normal    ScalingReplicaSet 19m    deployment-controller
Scaled up replica set nginx-deployment-cd55c47f5 to 2 from 1
  Normal    ScalingReplicaSet 82s    deployment-controller  Scaled up
replica set nginx-deployment-cd55c47f5 to 3 from 2
  Normal    ScalingReplicaSet 82s    deployment-controller  Scaled up
replica set nginx-deployment-8556cb945 to 1
  Normal    ScalingReplicaSet 76s (x2 over 22m) deployment-controller  Scaled down
replica set nginx-deployment-cd55c47f5 to 2 from 3
  Normal    ScalingReplicaSet 76s    deployment-controller  Scaled up
replica set nginx-deployment-8556cb945 to 2 from 1
  Normal    ScalingReplicaSet 73s    deployment-controller  Scaled down
replica set nginx-deployment-cd55c47f5 to 1 from 2
  Normal    ScalingReplicaSet 73s    deployment-controller  Scaled up
replica set nginx-deployment-8556cb945 to 3 from 2
  Normal    ScalingReplicaSet 70s    deployment-controller  Scaled down
replica set nginx-deployment-cd55c47f5 to 0 from 1

```

Questa modifica ha causato quindi un *rollout* del Deployment: questo vuol dire che adesso abbiamo uno storico che comprende due versioni del Deployment, le quali tengono traccia di tutti i cambiamenti avvenuti finora. Questo approccio ricorda molto il versionamento del codice, attraverso il quale possiamo salvare non solo le modifiche relative al nostro lavoro, ma anche il *delta*: qualora cambiassimo idea e volessimo tornare indietro, possiamo farlo in qualsiasi momento. Il comando `kubectl rollout` mostra infatti le diverse versioni del Deployment, con la relativa “causa”:

Listato 5.40 Cronologia delle versioni del Deployment

```

kubectl rollout history deployments nginx-deployment
>>>
deployment.apps/nginx-deployment
REVISION  CHANGE-CAUSE
1          <none>
2          Aggiornamento immagine alla versione 1.23.3

```

La cronologia delle versioni è data dal più vecchio al più recente, e un numero di versione univoco viene incrementato per ogni nuova implementazione. Finora ne abbiamo due: il Deployment iniziale, e poi l’aggiornamento dell’immagine a `nginx:1.23.3`. Quello che abbiamo attuato è un singolo cambiamento all’interno della risorsa `nginx-deployment`, che ha portato il controller ad aggiornare una serie di informazioni relative ai Pod che gestisce: qualora volessimo però attuare più modifiche e volessimo mettere “in pausa” il versionamento, fintanto che non avremo terminato le attività, potremmo utilizzare il comando `kubectl rollout pause` e successivamente `kubectl rollout resume` per riprendere con il versionamento:

Listato 5.41 Rollout in pausa

```

kubectl rollout pause deployments nginx-deployment
>>>
deployment "nginx-deployment" paused

```

Listato 5.42 Rollout ripristinato

```
kubectl rollout resume deployments nginx-deployment
>>>
deployment "nginx-deployment" resumed
```

Queste versioni ci aiutano, come abbiamo detto, a tenere traccia della storia del Deployment: questo può essere utile per due scopi, ossia la possibilità di analizzare le differenze tra una versione e l'altra, e anche la possibilità di tornare a una versione specifica. Queste due operazioni possono essere eseguite tramite i seguenti comandi:

Listato 5.43 Descrizione di una versione specifica del rollout di nginx-deployment

```
kubectl rollout history deployment nginx-deployment --revision=2
>>>
deployment.apps/nginx-deployment with revision #2
Pod Template:
  Labels:    app=nginx
             pod-template-hash=8556cb945
  Annotations: kubernetes.io/change-cause: Aggiornamento immagine alla versione 1.23.3
  Containers:
    nginx:
      Image:      nginx:1.23.3
      Port:       80/TCP
      Host Port:  0/TCP
      Environment: <none>
      Mounts:     <none>
  Volumes:   <none>
```

Listato 5.44 Ripristino di una versione precedente

```
kubectl rollout undo deployments nginx-deployment
>>>
deployment "nginx-deployment" rolled back
```

Se eseguiamo nuovamente il comando che riporta l'elenco delle versioni, noteremo che manca la revisione 1: questo perché quando si esegue il *rollback* (si torna indietro) a una versione precedente, il Deployment riutilizza semplicemente il *template* e lo ricontra in modo che sia l'ultima versione disponibile. Ciò che prima era la versione 1 è ora riordinato nella versione 3.

Listato 5.45 Elenco delle versioni dopo il rollback

```
kubectl rollout history deployments nginx-deployment
>>>
deployment.apps/nginx-deployment
REVISION  CHANGE-CAUSE
2          Aggiornamento immagine alla versione 1.23.3
3          <none>
```

La domanda sorge spontanea: e se volessi tornare a una versione specifica? Basterà aggiungere al comando utilizzato in precedenza per ripristinare una versione il parametro *--to-revision*:

Listato 5.46 Ripristino di una versione specifica

```
kubectl rollout undo deployments nginx-deployment --to-revision=2
>>>
deployment "nginx-deployment" rolled back
```

Finora abbiamo parlato di come gestire versioni, repliche, senza avere la preoccupazione di cosa succede quando si passa da una versione all'altra: ogni Deployment può infatti definire una strategia per gestire la creazione dei nuovi Pod quando c'è un aggiornamento in corso. Le strategie che Kubernetes mette a disposizione sono due: `Recreate` e `RollingUpdate`, e sono molto diverse tra loro. Per prima cosa, vediamo come inserire una strategia all'interno della definizione di un Deployment: la sezione evidenziata nel seguente esempio va inserita all'interno delle specifiche della risorsa, dove

possiamo specificarne anche i dettagli implementativi. In questo caso, stiamo utilizzando una strategia `RollingUpdate`, ossia la creazione dei Pod aggiornati “un po’ alla volta”, così che non ci sia alcun tempo di disservizio: questa metodologia è perfetta per qualsiasi servizio sia attivamente richiesto dagli utenti, di modo che i Pod in transizione continuino a ricevere il traffico fintanto che i nuovi non saranno attivi.

Listato 5.47 Definizione della strategia RollingUpdate

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 2
      maxUnavailable: 1
...
...
```

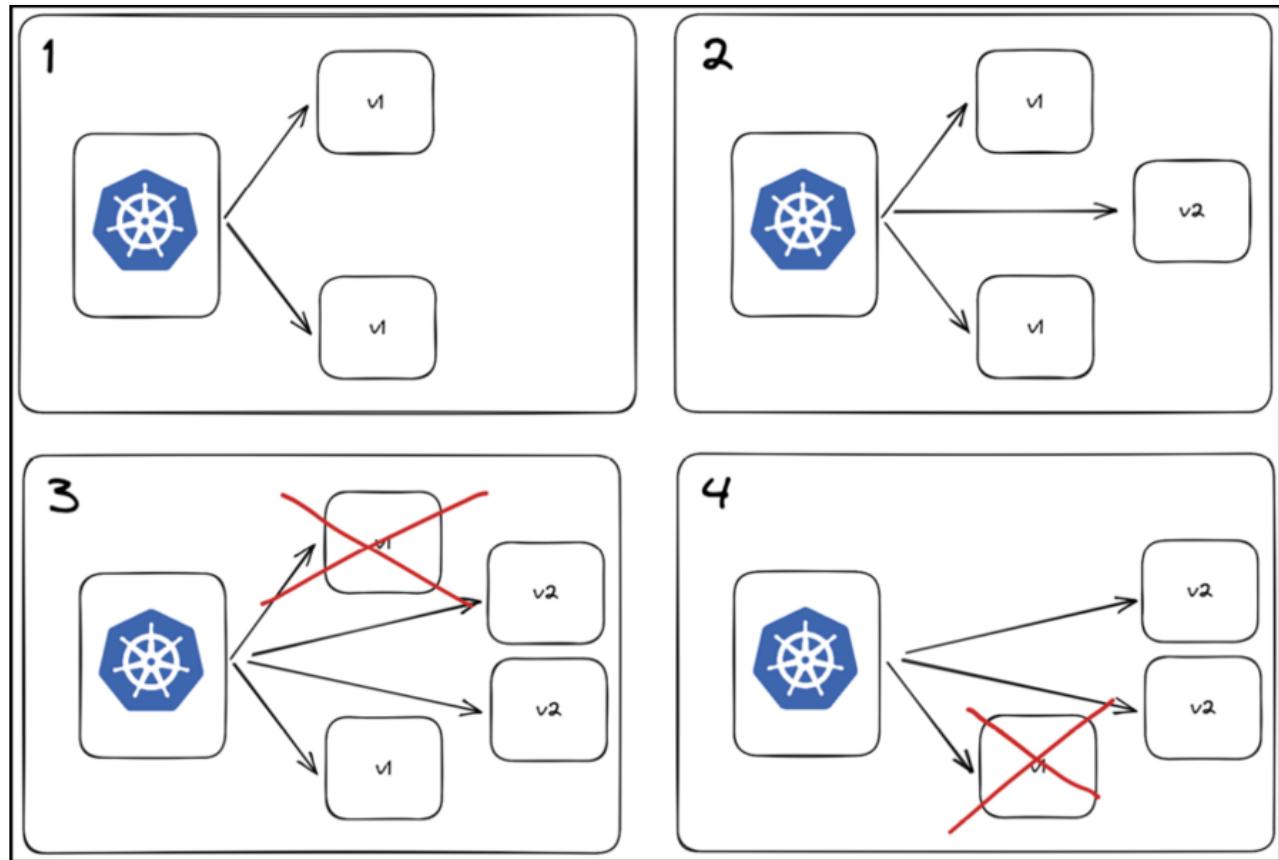


Figura 5.21 Esempio di “rolling” come strategia per un Pod.

Quando si utilizza la strategia `RollingUpdate`, sono disponibili altre due opzioni che ci consentono di perfezionare il processo di aggiornamento.

- `maxSurge`: il numero di Pod che possono essere creati oltre la quantità desiderata di Pod durante un *rollout*. Può trattarsi di un numero assoluto o di una percentuale del conteggio delle repliche. Il valore predefinito è 25%.

- `maxUnavailable`: il numero di Pod che possono non essere disponibili durante il processo di aggiornamento. Come prima, può essere un numero assoluto o una percentuale del conteggio delle repliche; il valore predefinito è 25%.

L'altra alternativa che abbiamo a disposizione per un Deployment è la strategia `Recreate`: è la più semplice delle due strategie, in quanto aggiorna il ReplicaSet terminando tutti i Pod associati al Deployment e creando successivamente i nuovi. Questo avviene perché i Pod vengono rimossi e il ReplicaSet nota che non ha più repliche, per cui procede a ricreare tutti i Pod utilizzando, nel nostro caso, la nuova immagine. Sebbene questa strategia sia veloce e semplice, ha un grosso svantaggio: quasi certamente comporterà alcuni tempi di inattività del servizio. Per questo motivo, questa strategia deve essere utilizzata solo per Deployment di test in cui l'applicazione non è rivolta all'utente ed è accettabile riscontrare del tempo di inattività. Non a caso, la strategia `RollingUpdate` è la predefinita!

Un esempio con la strategia `Recreate` è il Listato 5.46.

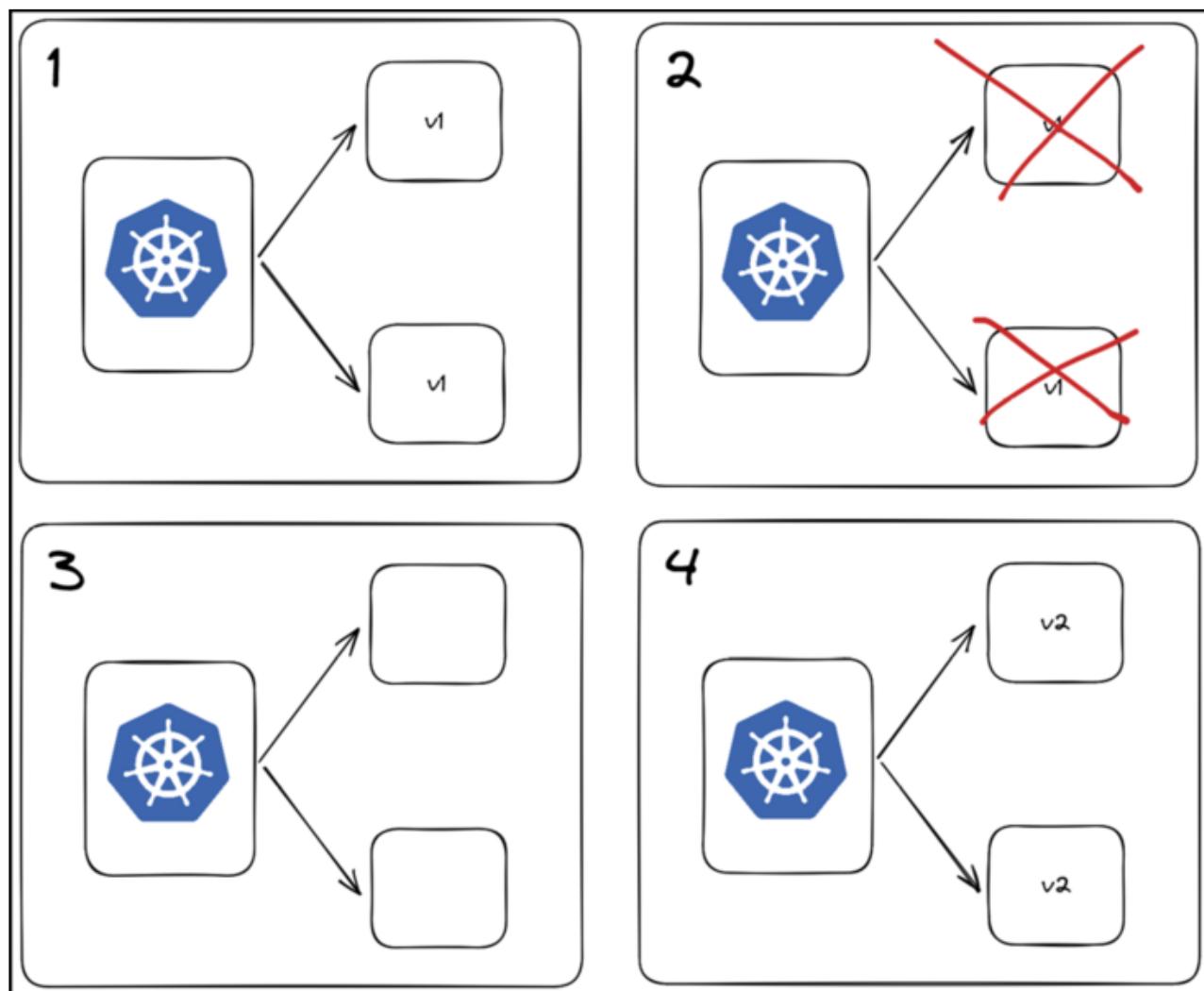


Figura 5.22 Esempio di Recreate come strategia per un Pod.

Listato 5.48 Definizione della strategia Recreate

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
```

```
spec:  
  strategy:  
    type: Recreate  
  ...
```

Abbiamo fatto dei passi da giganti finora con l'esplorazione dei Deployment, ma c'è molto, molto di più: per poter proseguire con i prossimi step, partiremo da diversi esempi pratici e andremo a vedere come eseguire il deploy di un'applicazione tramite un Deployment che ci permetterà di arricchire la gestione del suo operato, tramite strumenti come gli `init container`, gli `healthcheck` e la definizione delle risorse che i Pod potranno richiedere, in termini di CPU e memoria.

Init Container

Una premessa è doverosa: gli Init Container, così come i prossimi argomenti, sono applicabili anche al solo oggetto Pod: finora non sono state trattati in quanto, grazie al lavoro fatto con i controller, i Pod sono oggetti con cui difficilmente avremo a che fare direttamente: l'approccio più comune è quello che prevede la creazione di queste risorse e la relativa gestione tramite un Deployment (o altre tipologie di controller disponibili) per risparmiarci gran parte della fatica, e affidare a qualcuno la salvaguardia della nostra applicazione.

Detto ciò, passiamo al significato di Init Container: si tratta di un tipo speciale di container che viene eseguito prima di quelli "applicativi"; questi devono essere eseguiti e terminati correttamente prima che si avvii l'esecuzione dei container dell'applicazione principale. Sono diversi dai container dell'applicazione per diverse ragioni: se il Pod ha molti Init Container, ciascuno di essi viene eseguito a partire dal completamento del precedente, finché non saranno esauriti e si potrà passare al container finale. Inoltre, se l'Init Container riportasse un errore, ci sono due strade: è possibile definire una policy per cui il Pod verrà riavviato fintanto che questo container "speciale" non avrà esito positivo oppure il Pod verrà segnato come `failed` (in riferimento al ciclo di vita del Pod).

Per questa ragione, gli Init Container sono considerate entità separate dai container delle applicazioni e usano immagini diverse con diverse configurazioni. Sono utili per molteplici scopi: vengono utilizzati per l'inizializzazione dell'applicazione, come può essere il recupero di dati necessari per la configurazione dell'applicazione, o anche per ritardare la creazione del Pod finché non vengano soddisfatte una serie di precondizioni, come la connessione a un servizio.

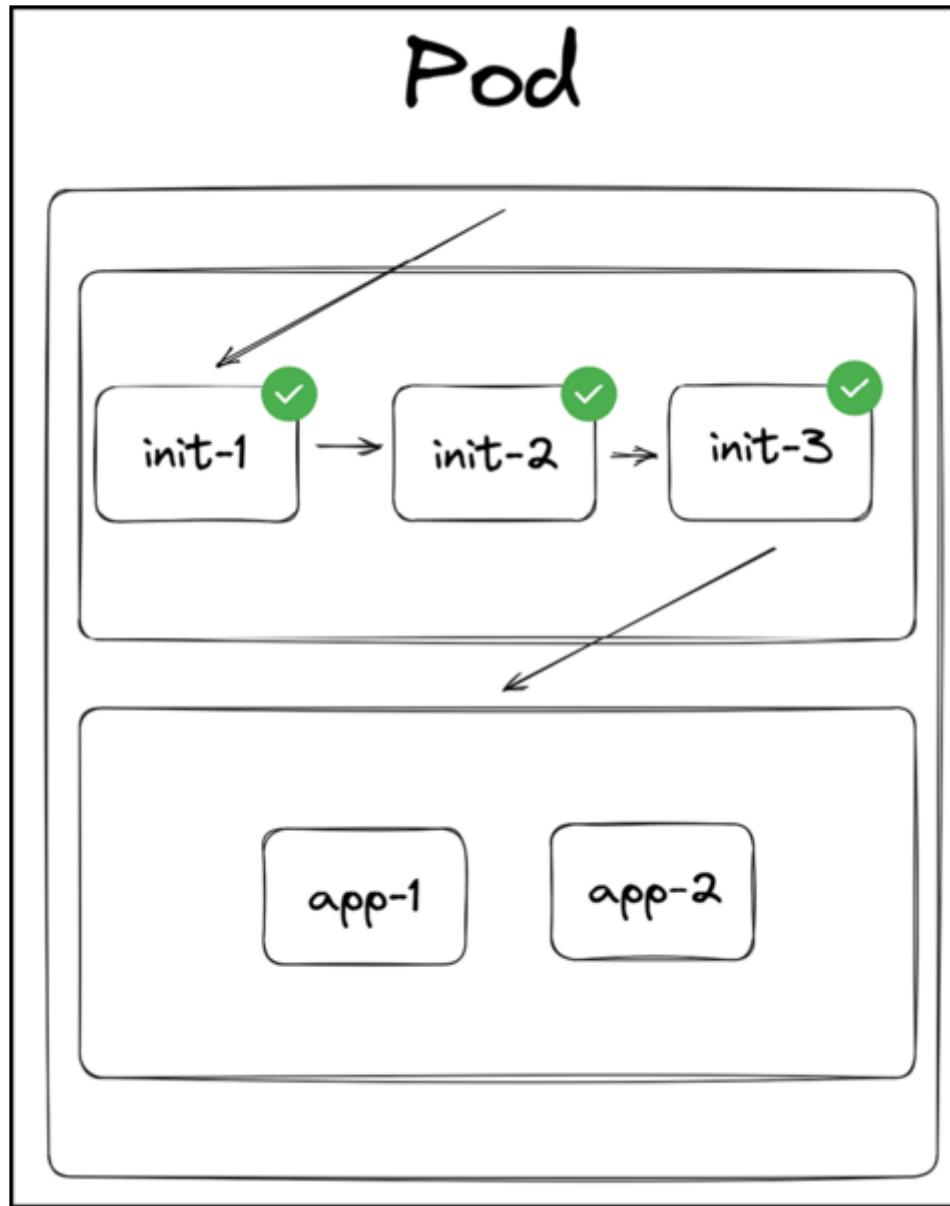


Figura 5.23 Ordine di esecuzione dei container di un Pod.

Per creare un Init Container, possiamo utilizzare sempre la definizione YAML del Deployment, e aggiungere un campo `initContainer` all'interno delle specifiche dell'oggetto:

Listato 5.49 Definizione degli initContainer

```

apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx-server
    name: nginx-deploy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-server
  template:

```

```

metadata:
  labels:
    app: nginx-server
spec:
  volumes:
    - name: common-disk
      emptyDir: {}
initContainers:
...
  containers:
...

```

L'aspetto interessante è che gli Init Container possono contenere utility o codice *ad hoc* per l'installazione di risorse che non sono presenti in un'immagine dell'applicazione, senza che sia necessario avere un'immagine di base da utilizzare; è possibile sfruttare i campi `command` e `args` per utilizzare strumenti come `sed`, `awk` o Python ed eseguire una serie di controlli prima che il container applicativo salga.

Procediamo con un esempio semplice: creiamo un Deployment per Nginx che utilizzi un Init Container per poter creare una prima pagina da mostrare all'utente: per farlo, sfrutteremo un'immagine estremamente semplice, ossia quella di `busybox`, e tramite questa risorsa aggiungeremo un file `index.html` con una prima riga di HTML. Nel container principale, aggiungiamo invece il container vero e proprio: Nginx ospiterà la nostra pagina e la mostrerà agli utenti.

Per farlo, di seguito viene riportato un esempio: prima definiamo il Deployment chiamato `nginx` e in seguito il campo `template`.

Listato 5.50 Esempio di Deployment con Init Container – parte 1

```

apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx-server
    name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-server
  template:
...

```

All'interno di questo campo, inseriamo come di consueto i metadata dei Pod e poi i dettagli degli Init Container: così come un “normale” container, dobbiamo specificare l'immagine da utilizzare (nel nostro caso `busybox`) e il nome del container. Sarà necessario aggiungere anche una cartella dove inserire il file che creeremo, per cui usiamo il campo `volumeMounts` con un nome e un path dove inserire questo file. Creiamo poi, tramite il comando `echo`, il file `index.html` e posizioniamolo nella cartella del volume.

Listato 5.51 Esempio di Deployment con Init Container – parte 2

```

...
  template:
    metadata:
      labels:
        app: nginx-server
    spec:
      initContainers:
        - name: busybox
          image: busybox
      volumeMounts:

```

```

    - name: shared-space
      mountPath: /web-files
    command: ["/bin/sh"]
      args: ["-c", "echo '<h2>Init container added this line!</h2>' > /web-
files/index.html && sleep 10"]
...

```

Completato questo step, possiamo passare alla definizione del container: questo utilizzerà l'immagine di Nginx (come prima, non è importante il tag in questo momento) e monteremo la stessa cartella utilizzata dall'Init Container per poter recuperare il file creato in precedenza. Da notare che questo container non sarà avviato fintanto che il comando specificato nell'Init Container non sarà completato con successo.

Listato 5.52 Esempio di Deployment con Init Container – parte 3

```

...
  template:
metadata:
  labels:
    app: nginx-server
spec:
  volumes:
    - name: common-disk
      emptyDir: {}
  initContainers:
    - name: busybox
      image: busybox
  volumeMounts:
    - name: shared-space
      mountPath: /web-files
    command: ["/bin/sh"]
      args: ["-c", "echo '<h2>Init container added this line!</h2>' > /web-
files/index.html && sleep 10"]
  containers:
    - image: nginx
      name: nginx
      volumeMounts:
        - name: shared-space
          mountPath: /usr/share/nginx/html

```

Questo il risultato finale:

Listato 5.53 Esempio di Deployment con Init Container – completo

```

apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx-server
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-server
  template:
    metadata:
      labels:
        app: nginx-server
  spec:
    volumes:
      - name: common-disk
        emptyDir: {}
  initContainers:

```

```

- name: busybox
  image: busybox
  volumeMounts:
  - name: common-disk
    mountPath: /web-files
    command: ["/bin/sh"]
    args: ["-c", "echo '<h2>Init container added this line!</h2>' > /web-
files/index.html && sleep 20"]
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - name: common-disk
      mountPath: /usr/share/nginx/html

```

Utilizzando `kubectl apply`, possiamo creare il Deployment e quindi accedere alla shell del Pod, dove possiamo provare a eseguire una `curl` per vedere se la pagina è stata creata con successo. Non possiamo farlo ancora tramite browser perché non abbiamo “esposto” la nostra applicazione all'esterno, ma lo faremo in seguito!

Listato 5.54 Test del Deployment di Nginx

```

kubectl apply -f deployment.yaml
>>>
deployment.apps/nginx created

kubectl get pods
>>>
NAME                  READY   STATUS    RESTARTS   AGE
nginx-deploy-964b87b86-c5k5j   1/1     Running   0          32s

kubectl exec -it nginx-deploy-55b5965c98-c5k5j -- /bin/sh
>>>
# curl localhost
<h2>Init container added this line!</h2>

```

Con questo esempio così semplice possiamo già iniziare ad apprezzare il potenziale degli `init container`: pensiamo a come utilizzarli su un caso più grande, per esempio una situazione in cui abbiamo bisogno di creare delle tabelle di base su un database prima che l'applicazione possa partire e iniziare a comunicare con esso, o anche a un check che tutti i servizi da cui dipende l'applicazione principale esposta dal container siano attivi e in esecuzione, pronti alle prossime attività... A proposito di “attivi” e “in esecuzione”, diamo un’occhiata alla prossima sezione e scopriamo che cosa vuol dire verificare lo stato di integrità di un'applicazione.

Healthcheck

Uno dei principali vantaggi dell'utilizzo di Kubernetes è la possibilità di fornirgli un elenco di container e lasciare che mantenga tali container in esecuzione da qualche parte nel cluster al posto nostro, utilizzando una risorsa che li gestisce come un semplice Pod, o anche un controller. Ma che cosa succede se uno di quei container “muore”? Che cosa succede se tutti i container di un Pod muoiono?

Non appena un Pod viene “schedulato” (se ne pianifica l'esecuzione) su un nodo, sarà `kubelet` a eseguire su quel nodo i suoi container e, da quel momento in poi, li manterrà in esecuzione finché il Pod esiste. Se il processo principale del container si arresta in modo anomalo, `kubelet` riavvierà il container; se l'applicazione ha un bug che ne causa l'arresto di tanto in tanto, Kubernetes la riavvierà automaticamente.

Questi sono casi molto semplici, che tengono conto di quando le applicazioni smettono di funzionare senza che il loro processo principale si arresti in modo anomalo. Per esempio, un'app Java che abbia

un *memory leak* (in estrema sintesi, un consumo di memoria non previsto perché non si libera lo spazio da ciò che non è utilizzato attivamente dal processo) inizierà a lanciare una raffica di `OutOfMemoryErrors` nei log, ma il processo principale della JVM continuerà a funzionare, per cui potremmo potenzialmente non accorgercene mai. Sarebbe fantastico avere un modo per consentire a un'applicazione di segnalare a Kubernetes che non sta funzionando più correttamente e che gli permettesse di riavviarla. Per assicurarci che l'applicazione venga riavviata in casi come quello descritto in precedenza, è necessario controllare l'integrità di un'applicazione dall'esterno e non dipendere dall'applicazione stessa. Per questo, esistono le cosiddette `probes`: si tratta di test, di diversa natura, che permettono di controllare lo stato del container.

In Kubernetes ne esistono tre versioni, e ognuna di esse assume un significato diverso: parliamo di *liveness probe*, *readiness probe* e *startup probe*. Iniziamo con la prima: la liveness probe permette di valutare se un container è ancora "vivo" e verrà (come le altre) ripetuta periodicamente per poter agire per tempo qualora il container riscontrasse un errore. Per il momento, immaginiamo che grazie a Kubernetes ci sia un modo di "pingare" continuamente l'applicazione per verificare che sia tutto ok e che le richieste in arrivo possano essere gestite, come rappresentato in maniera schematica nella Figura 5.24.

Qui possono già sorgere delle domande: come poter implementare nel pratico questi test? Kubernetes mette a disposizione diversi meccanismi: il primo è quello di eseguire una richiesta HTTP GET al container e, se il codice di risposta è positivo, dichiarare il successo del test, altrimenti il container verrà riavviato. Un altro metodo è quello di sfruttare una *socket*, e quindi tentare l'apertura di una connessione TCP a una specifica porta del container: se la connessione avviene con successo, allora il test avrà esito positivo, altrimenti (indovina) il container verrà riavviato. L'ultimo metodo è quello di eseguire un comando che ci permetta di verificare lo stato del container, e che ritorni un codice pari a 0 se la risposta è affermativa, un numero qualsiasi altrimenti. Tutti i metodi elencati sono validi e non c'è una preferenza particolare: la scelta dipende molto dallo stack tecnologico con cui abbiamo a che fare, e che ci permette già di fare una prima scrematura; nel caso di un'applicazione web, sarà più semplice fare una richiesta a un path specifico per ottenere una risposta, mentre con alcuni tipi di servizi dovremo lavorare con un comando personalizzato e adatto al caso d'uso.

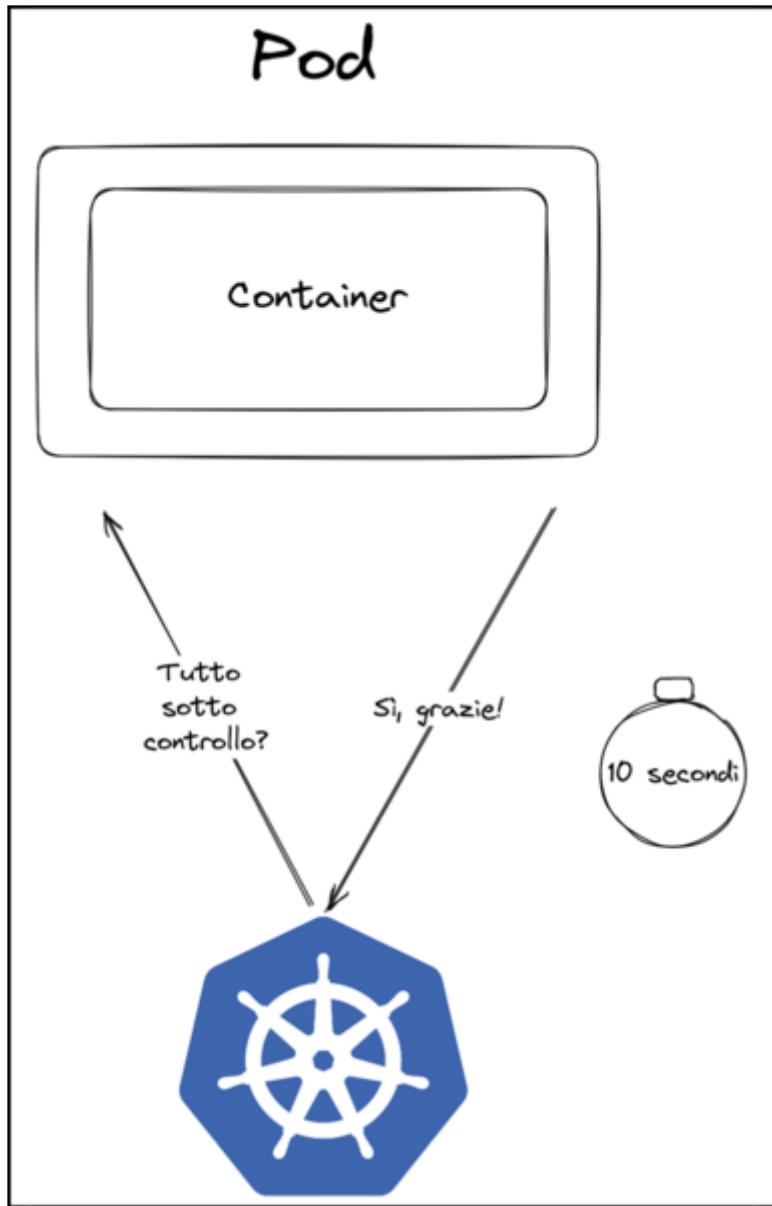


Figura 5.24 Rappresentazione del funzionamento dell'algoritmo di control loop.

Facciamo un esempio, per vedere in pratica come funziona: questo ci permetterà di trattare in maniera semplice anche le altre due che ci mancano. Prendiamo un'applicazione Node.js molto semplice, che permetta di eseguire una richiesta a due contesti: `healthcheck` risponderà con un codice positivo (per intenderci, un codice di stato tra il 200 e il 399, secondo il protocollo HTTP), se l'applicazione è già in esecuzione. Qualsiasi altra pagina (esclusa quella principale) risponderà con un codice negativo, quindi un valore uguale o superiore a 400. L'applicazione Node.js avrà un aspetto simile a questo (ma puoi usare anche una tua applicazione):

Listato 5.55 Esempio di applicazione Node.js - file app.js

```
'use strict';
const express = require('express');

const PORT = 8080;
const HOST = '0.0.0.0'; // App
const app = express();

app.get('/', (req, res) => {
```

```

    res.send('Hello world');
});

app.get('/healthcheck', (req, res)=> {
    res.send ("Health check passed");
});

app.listen(PORT, HOST);

console.log(Running on http://${HOST}:${PORT});

```

In questo file, abbiamo configurato tre *route* del server Express.js (un framework di Node.js) che rispondono alle richieste GET in arrivo: il primo serve le richieste al percorso principale (per intenderci, quelle che arrivano su localhost o un indirizzo IP, senza alcun contesto a seguito) che invia un messaggio "Hello world" come risposta. Il secondo percorso denominato /healthcheck restituisce uno stato HTTP pari a 200, il che indica al nostro test che l'applicazione è integra e in esecuzione. A questo punto, andiamo a creare un Deployment che, avendo a disposizione l'immagine di Node.js già pronta all'uso, possa eseguire la liveness probe: le informazioni su come eseguire questo test vengono aggiunte sempre all'interno del campo `template`, all'interno del singolo container; ogni container all'interno del Pod dovrà infatti rispondere con successo a questi test, per poter affermare con certezza che l'intero Pod è in funzione.

Listato 5.56 Esempio di Deployment per testare la liveness con Node.js

```

apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nodejs
    name: nodejs
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nodejs
  template:
    metadata:
      labels:
        app: nodejs
    spec:
      containers:
        - image: my-nodejs-app
          name: nodejs
          ports:
            - containerPort: 8080
      livenessProbe:
        httpGet:
          path: /healthcheck
          port: 8080
        initialDelaySeconds: 3
        periodSeconds: 3
        failureThreshold: 3

```

In primis, aggiungiamo all'interno della sezione `container` un campo `ports`: questo ci permetterà di rendere raggiungibile l'applicazione sulla porta 8080; nella sezione `livenessProbe` andiamo invece a definire le specifiche di questo test, dove vediamo diversi campi da compilare. Abbiamo la metodologia scelta che, nel nostro caso, corrisponde a una richiesta HTTP: per permettere a Kubernetes di eseguirla correttamente, specifichiamo il `path` che è stato indicato nel file principale dell'applicazione Node.js e la porta da utilizzare, e poi indichiamo alcuni dei dettagli sul come questa richiesta debba essere eseguita. Per esempio, indichiamo un ritardo iniziale di 3 secondi (campo

`initialDelaySeconds`): questo perché, quando il nostro container verrà avviato, non è detto che sia pronto a rispondere al test immediatamente, per cui vogliamo dargli modo di prepararsi; con il campo `periodSeconds`, andiamo invece a specificare ogni quanto è necessario testare l'applicazione (di default, 10 secondi): più stretto è l'intervallo, più saremo pronti a rispondere a eventuali guasti (questa è la regola generale), anche se molto dipende dal caso specifico. In ultimo, indichiamo il numero di tentativi da eseguire prima di dichiarare il fallimento del test e richiedere quindi il riavvio del Pod: come valore predefinito, c'è il numero 3.

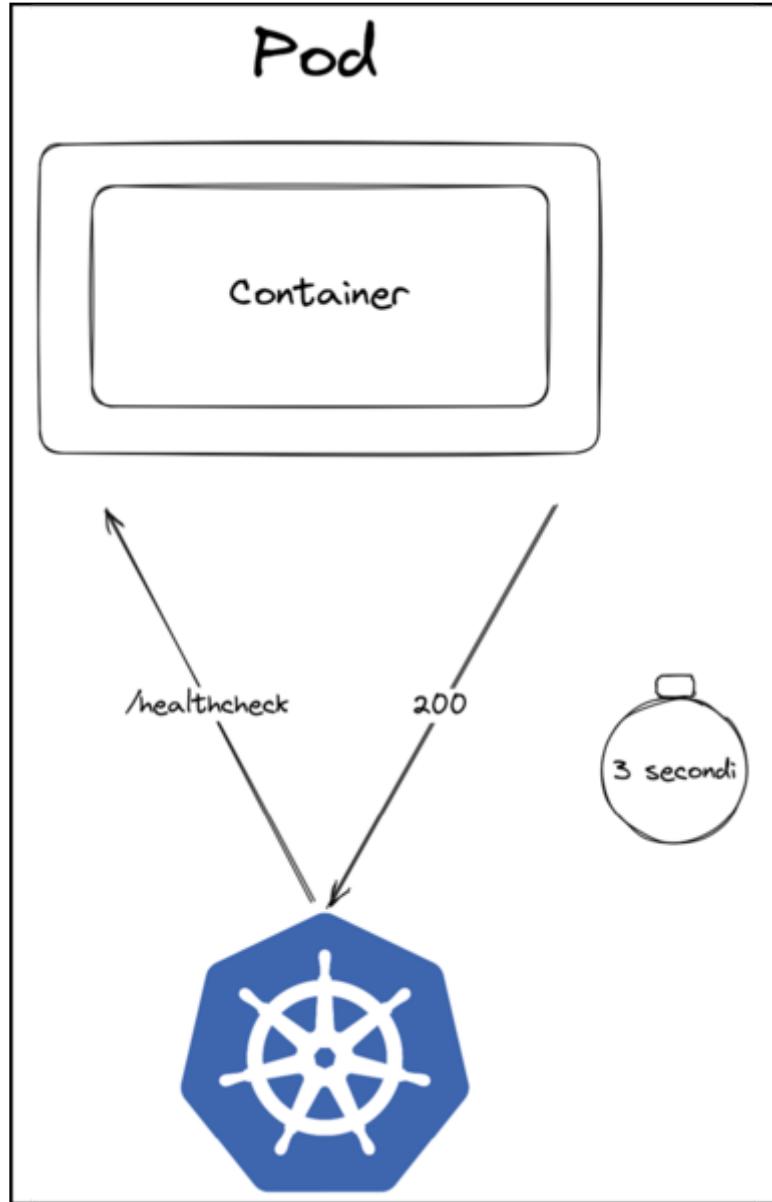


Figura 5.25 Esempio di funzionamento del controllo dello stato di integrità di un Pod tramite gli healthcheck.

Testiamo la nostra applicazione e creiamo il Deployment: vedremo che l'applicazione è in esecuzione correttamente; questo vuol dire che la liveness probe ha funzionato e che è in grado di verificare lo stato del container. Facciamo una prova artificiale, o del nove: proviamo a modificare il Deployment inserendo un endpoint errato, per verificare se il Pod viene riavviato come ci si aspetta: cambiamo il path come indicato di seguito, e attendiamo.

Listato 5.57 Modifica al path della liveness probe

```
...
  containers:
    - image: my-nodejs-app
      name: nodejs
      ports:
        - containerPort: 8080
      livenessProbe:
        httpGet:
          path: /health-check
          port: 8080
...

```

Utilizzando il comando `kubectl get Pods` con l'opzione `-w`, potremo osservare come cambia lo stato del Pod: l'avvio riesce con successo, ma dopo 10 secondi verrà eseguita la liveness probe, che fallirà e che comporterà altri due tentativi. Al terzo tentativo, il riavvio del Pod sarà necessario:

Listato 5.58 Modifica al path della liveness probe

```
kubectl get pods -w
>>>
NAME           READY   STATUS    RESTARTS   AGE
nodejs-xxx     1/1     Running   0          6s
nodejs-xxx     1/1     Running   1 (1s ago) 11s
nodejs-xxx     1/1     Running   2 (1s ago) 20s
nodejs-xxx     0/1     CrashLoopBackOff 2 (0s ago) 28s
nodejs-xxx     1/1     Running   3 (15s ago) 43s
nodejs-xxx     1/1     Running   4 (1s ago) 50s
nodejs-xxx     0/1     CrashLoopBackOff 4 (0s ago) 58s
```

Qui ci viene mostrato un errore molto importante: `CrashLoopBackOff`. Questo indica un errore apparentemente misterioso, ma che richiede un'attenta analisi: nel nostro caso l'errore è artificiale, e il Pod è in errore per via dell'endpoint sbagliato su cui la liveness probe sta tentando di recuperare lo stato del container; che cosa succederebbe se non avessimo questa informazione? Prima di tutto, in presenza di questo errore, si parla di un problema applicativo: c'è qualcosa nella nostra applicazione (non nell'infrastruttura) che non sta funzionando come dovrebbe. Per indagare più a fondo, possiamo sfruttare gli eventi del Pod, ma anche i log applicativi. Partiamo dagli eventi, e usiamo `kubectl` per recuperarli: come vediamo nelle prime righe (quelle più vecchie), l'immagine è stata scaricata correttamente e il Pod è stato schedulato su un nodo specifico; dalla quarta riga in poi, le cose iniziano a mettersi male e il test fallisce: questo viene esplicitato in una delle ultime righe, in cui `kubelet` indica chiaramente che il problema è relativo alla `probe` che fallisce, perché tenta di chiamare il `path` indicato nella definizione YAML e ottiene un codice 404.

Listato 5.59 Eventi del Pod

```
...
Events:
  Type  Reason  Age   From            Message
  ----  -----  --   ----            -----
  Normal Scheduled 3m40s default-scheduler  Successfully assigned k8s-training/nodejs-xxx to XXX
  Normal Pulled   3m38s kubelet         Successfully pulled image "my-nodejs-app" in 749.552488ms
  Normal Pulled   3m29s kubelet         Successfully pulled image "my-nodejs-app" in 747.612197ms
  Normal Created   3m20s (x3 over 3m38s) kubelet       Created container nodejs
  Normal Started  3m20s (x3 over 3m38s) kubelet       Started container nodejs
  Normal Pulled   3m20s kubelet         Successfully pulled image "my-nodejs-app" in 727.932775ms
```

```

Warning  Unhealthy  3m12s (x6 over 3m33s)  kubelet      Liveness probe failed:
HTTP probe failed with statuscode: 404
Normal   Killing   3m12s (x3 over 3m30s)  kubelet      Container nodejs failed
liveness probe, will be restarted
Warning  BackOff   3m12s (x2 over 3m12s)  kubelet      Back-off restarting
failed container
...

```

Come appena visto, compiere questo tipo di analisi è estremamente utile per permetterci di arrivare in fondo al problema, e per fortuna tutto ciò che avviene all'interno del cluster viene riportato nei cambi di stato del ciclo di vita delle risorse con cui lavoriamo. Se volessimo indagare ancora un po' più a fondo, potremmo aver bisogno di leggere i log del Pod: per farlo, esiste il comando `kubectl logs`, che ci permette di leggere quanto riportato dal container. Il problema è il seguente: quando questo viene riavviato, a causa di un errore come quello precedente, i log vengono “persi”, a meno che non si vogliano recuperare esplicitamente; questo è possibile aggiungendo il parametro `--previous`, che ci permette di leggere quelli relativi al container immediatamente precedente.

Listato 5.60 Esempio di comando

```

kubectl logs my-nodejs-app --previous

> node-app@1.0.0 start /usr/src/app
> node server.js

Running on http://0.0.0.0:8080

```

Ora che la liveness probe ha visto la sua prima applicazione, passiamo alle altre due, e iniziamo dalla *readiness probe*: questo test viene eseguito periodicamente e determina se il Pod specifico può ricevere o meno le richieste da parte di un agente esterno; quando restituisce esito positivo, segnala che il container è pronto ad accettare le richieste in ingresso. Come per la liveness probe, è possibile attuare tre diverse strategie per testare lo stato dell'applicazione, e le stesse dipendono sempre da cosa si vuole testare; anche i parametri di configurazione, come il ritardo nell'esecuzione dei test o la periodicità con cui questo viene eseguito sono configurabili allo stesso modo. Cambia il significato che assegniamo alle due e la relativa esecuzione: quando un container viene avviato, Kubernetes può essere configurato per attendere un certo intervallo di tempo configurabile prima di eseguire la readiness probe. Successivamente, utilizza in maniera periodica questa probe e agisce in base al risultato del test; qualora questa non risultasse “pronta” per comunicare con altri servizi, questa non sarà più raggiungibile dalle altre applicazioni.

Listato 5.61 Esempio di readiness probe

```

...
ports:
- containerPort: 8080

livenessProbe:
  httpGet:
    path: /ready
    port: 8080
  failureThreshold: 1
  periodSeconds: 10
...

```

Per spiegare un po' meglio questa differenza, proviamo ad aggiungere un tassello che verrà spiegato nel dettaglio più avanti: immaginiamo che per ogni Pod venga messo a disposizione uno strato di comunicazione chiamato Service: si tratta di una risorsa Kubernetes che, tramite diverse tecniche, gestisce il traffico in ingresso e in uscita da un Pod, occupandosi della comunicazione con altre applicazioni. Se vogliamo usare una metafora, lavora come un vigile urbano in mezzo al traffico:

quando il semaforo è verde, e quindi la readiness probe è stata eseguita con successo, le richieste possono passare, ma quando è rosso, si va a chiudere la strada, così che le richieste passino altrove (se possibile). Quindi, a differenza della liveness probe, se un container non supera questo test, non verrà riavviato e questa è una distinzione importante tra le due: mentre la liveness si occupa di tenere le applicazioni sempre attente e attive, la readiness si preoccupa che solo i Pod che sono in grado di rispondere siano contattati dall'esterno.

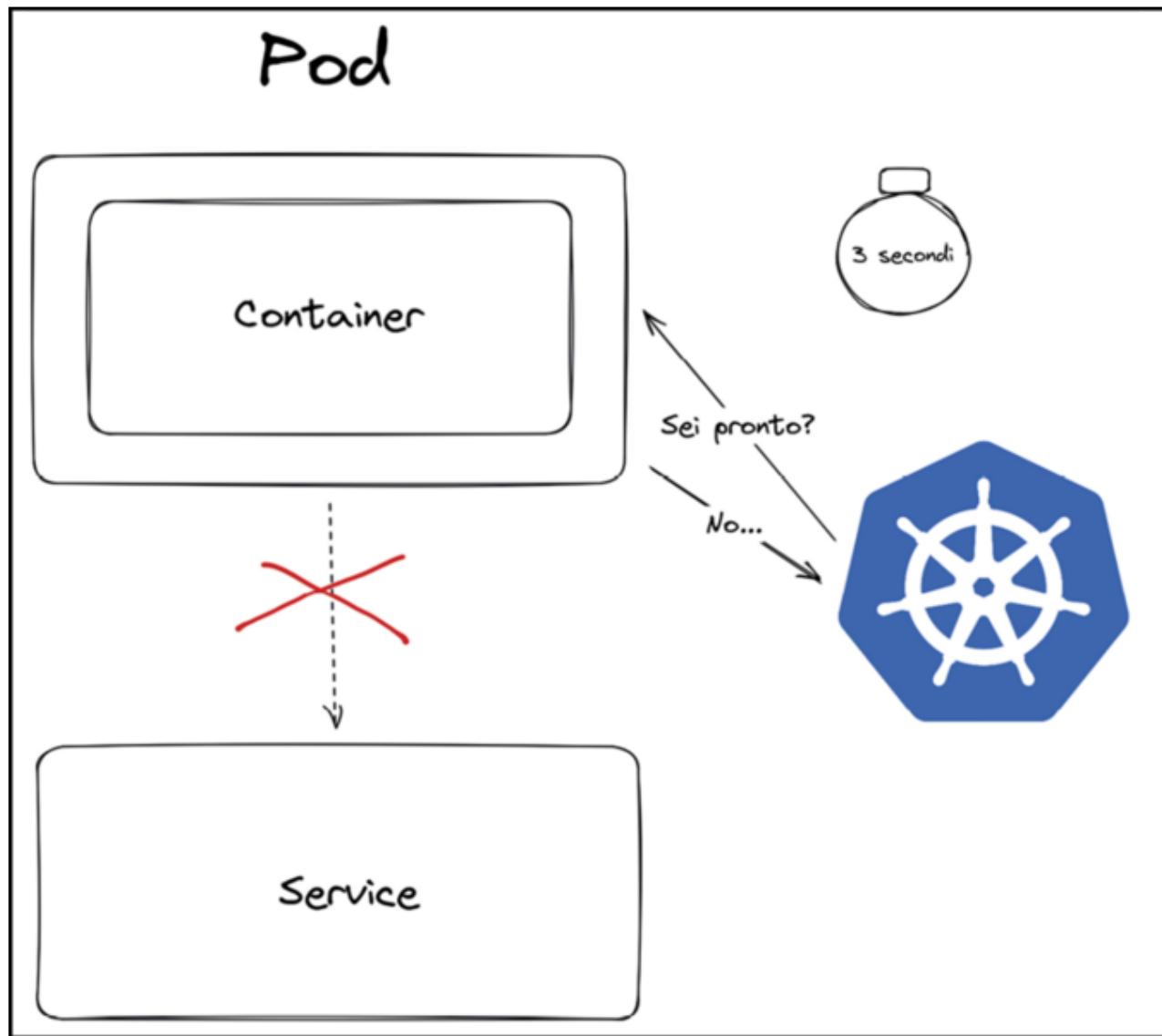


Figura 5.26 Che cosa avviene quando un Pod non supera la readiness probe.

Ultima, ma non ultima, la *startup probe*: questa viene utilizzata quando, a volte, devi gestire applicazioni legacy che potrebbero richiedere un tempo di avvio aggiuntivo alla loro prima inizializzazione. In tali casi, può essere complicato impostare i parametri delle *probes* “tradizionali” senza compromettere la risposta rapida di questi test. Il trucco è impostare la startup probe con lo stesso comando, richiesta HTTP o TCP socket che sia, con una soglia di tentativi falliti o di periodicità abbastanza lunga da coprire il tempo di avvio del caso peggiore.

Listato 5.62 Esempio di startup probe

```
...  
ports:
```

```

- containerPort: 8080

livenessProbe:
  httpGet:
    path: /healthcheck
  port: 8080
  failureThreshold: 1
  periodSeconds: 10

  startupProbe:
    httpGet:
      path: /healthcheck
    port: 8080
    failureThreshold: 30
    periodSeconds: 10
...

```

Abbiamo visto diversi esempi, e potrebbe sorgere la domanda: ho veramente bisogno di una liveness probe, o di una readiness probe? Partiamo dalla prima: per i Pod in esecuzione in produzione, devi sempre definire una liveness probe: senza questa, Kubernetes non ha modo di sapere se la tua applicazione è ancora attiva o meno, e finché il processo è ancora in esecuzione, Kubernetes considererà il container “integro”. Una liveness probe dovrebbe, banalmente, controllare se il server, o più genericamente l’applicazione, risponde: per quanto possa sembrare semplice, anche un test così può comunque dare modo a `kubelet` di riavviare il container se questo smette di rispondere alle richieste HTTP. Se volessimo dare una marcia in più a questa probe, potremmo utilizzare un endpoint custom (come quelli visti in precedenza, per esempio `/healthcheck`) per richiamare un servizio interno che verifichi che tutti i componenti vitali e interni all’applicazione siano responsivi. È importante specificare in questo caso come questo tipo di contesto non debba richiedere autenticazione: il rischio è che il test fallisca sempre, rendendo il container inutilizzabile. Un altro aspetto importante è quello di controllare solamente ciò che concerne la logica dell’applicazione che stiamo esponendo tramite container, prestando attenzione al fatto che questa non sia influenzata da fattori esterni. Per esempio, un test di liveness di un server di front-end non dovrebbe ritornare un errore se il server non riesce a collegarsi al database: questo perché se il database stesso sta riscontrando dei problemi, non sarà sufficiente riavviare il server per risolvere il problema. Questo porterebbe la liveness probe a fallire ancora e ancora, finché il database non tornerà accessibile nuovamente.

Un esempio in cui una liveness probe potrebbe non essere necessaria è quando, per esempio abbiamo, a che fare con un server che espone dei file statici: l’applicazione si avvierà velocemente e uscirà solamente nel caso in cui riscontri un errore che gli impedisca di mostrare le pagine. Dal momento che il processo di Nginx produce un codice di errore ogni volta riscontra un problema di configurazione o di esposizione delle pagine, il container sarà riavviato naturalmente come avverrebbe per qualunque altra situazione analoga. Al contrario, la readiness probe è necessaria: questo perché il server andrà a gestire del traffico di rete in entrata e in uscita e, ogni volta che abbiamo a che fare con questa tipologia di applicazioni, un test del genere ci permette di evitare problemi all’avvio del container o magari nel momento in cui dovessimo aumentare le risorse a disposizione del cluster con un flusso di lavoro che cambia. Per fare un ultimo paragone, nel caso in cui avessimo a che fare con un job che si occupa di compiere delle attività ripetitive o programmate, sarà certamente utile a configurare una liveness probe per verificare che il processo sia in esecuzione correttamente e che, nel caso in cui il container riscontrasse un problema, possa essere gestito in maniera opportuna, ma non sarà necessario configurare una `readiness probe`: abbiamo detto che questa serve a mandare un segnale a Kubernetes per comunicare che il container è pronto a gestire del traffico di rete e che quindi la comunicazione con l’esterno e con altre applicazioni può essere attivata.

Requests e Limits

Quando Kubernetes vuole eseguire un Pod, è importante che i container dispongano di risorse sufficienti per essere effettivamente lanciati; infatti, se provi a eseguire un'applicazione di grandi dimensioni su un nodo del cluster con risorse limitate, è possibile che il nodo esaurisca la memoria o la CPU e anche altre applicazioni risentano di questi cambiamenti. Requests e limits sono due meccanismi utilizzati da Kubernetes per controllare risorse come CPU e memoria: le prime servono a definire ciò che Kubernetes dovrà garantire al container per potersi avviare; se un container richiede una risorsa, Kubernetes la pianificherà solo su un nodo che può fornirgli tali risorse, che quindi rappresentano un valore *minimo* che permette all'applicazione di partire correttamente. I limits, d'altra parte, servono ad assicurarsi che un container non superi mai un certo valore: in altre parole, al container è consentito utilizzare le risorse solo fino al limite specificato, che è quindi limitato.

È importante sottolineare che il *limite* non può mai essere inferiore alla *richiesta*: se provi a impostare questi valori violando tale regola, Kubernetes genererà un errore e non consentirà di eseguire il container. D'altronde, non avrebbe senso impostare un limite di risorse inferiore a quello strettamente necessario affinché il container si avvii!

Requests e limits sono definiti *per container*: sebbene i Pod di solito contengano un singolo container, abbiamo detto che è possibile incontrare anche Pod con più container. Ogni container nel Pod dovrà quindi avere il proprio *limite* e la propria *richiesta* individuale, tenendo presente che, poiché i Pod sono sempre eseguiti come un gruppo di container, è necessario sommare i limiti e le richieste per ogni container per ottenere un valore aggregato per il Pod. Nell'esempio in Figura 5.28, il Pod contiene due container e ognuno ha requests e limits definiti.

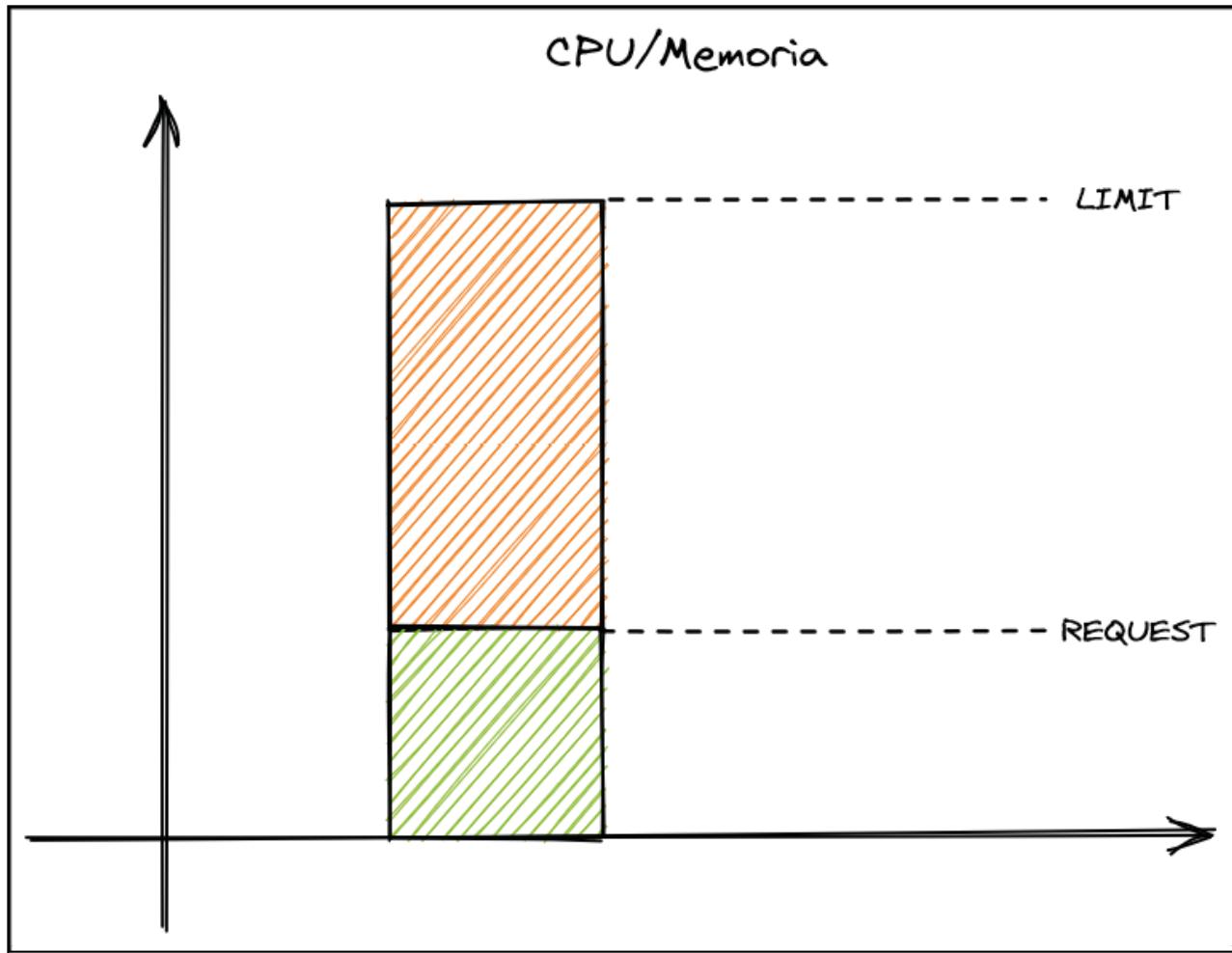


Figura 5.27 Rappresentazione di request e limit per CPU e memoria.

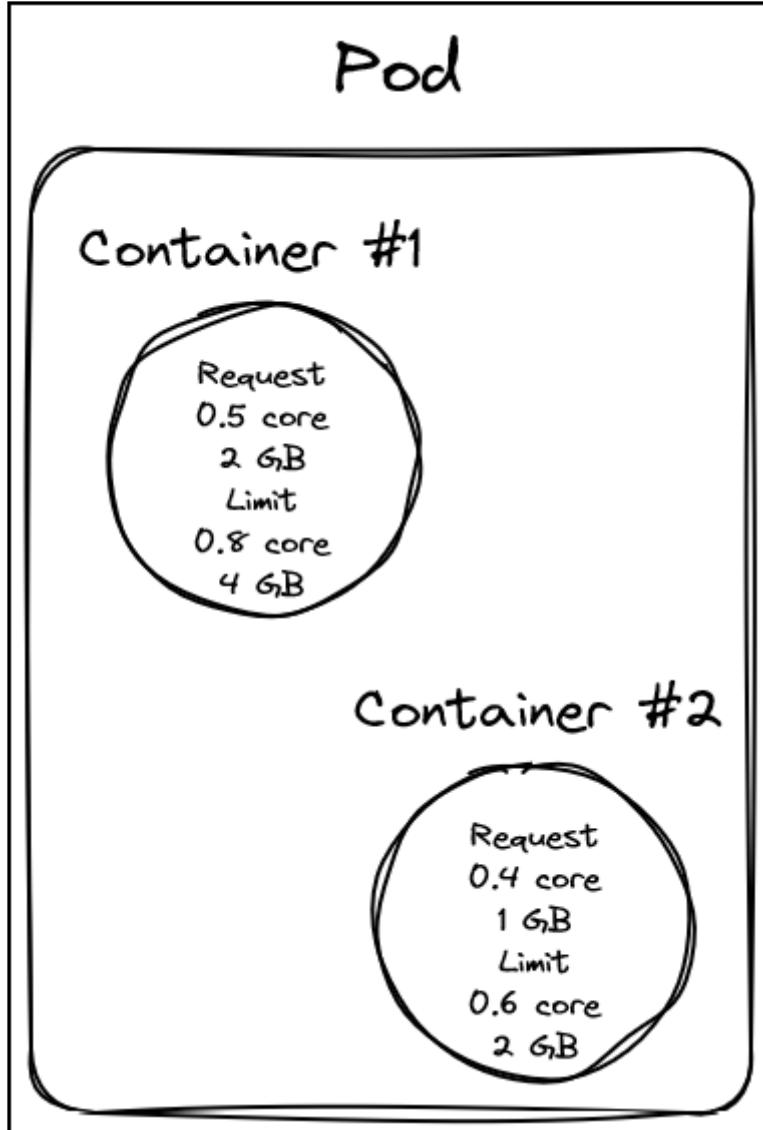


Figura 5.28 Esempio di definizione di request e limit per i container presenti all'interno del Pod.

A questo punto sarà evidente che esistono due tipi di risorse che possono essere configurate tramite requests e limits: CPU e memoria (intesa come memoria volatile e non come storage), che lo scheduler Kubernetes utilizza per capire qual è il nodo più appropriato per eseguire i Pod. Una tipica specifica per le risorse del Pod potrebbe essere simile a quella che segue.

Listato 5.63 Esempio di definizione di requests e limits

```
...
containers:
- name: container1
  image: busybox
  resources:
    requests:
      memory: "2Gi"
      cpu: "500m"
    limits :
      memory: "4Gi"
      cpu: "800m"
- name: container2
  image: busybox
```

```

resources:
  requests :
    memory: "1Gi"
    cpu: "400m"
  limits :
    memory: "2Gi"
    cpu: "600m"
...

```

Questo Pod di esempio ha due container: ciascuno di essi ha quindi le proprie richieste e limiti specificati all'interno della proprietà `resources`. Quindi, nell'esempio precedente, il Pod avrà quindi come `request` complessiva 0.9 core di CPU e 3 GiB di memoria, mentre come `limits` avrà 1.4 core e 6 GiB di memoria. Parliamo delle unità di misura: le risorse della CPU sono definite in *millicore*, che corrisponde a un'uguaglianza dove un core corrisponde a 1000 millicores. In altre parole, se il tuo container ha bisogno di due core completi per funzionare, dovresti inserire il valore `2000m`; se il tuo container ha bisogno solo di $\frac{1}{4}$ di un core, dovresti inserire un valore pari a `250m`. Le risorse di memoria sono definite in byte. Normalmente, dai un valore che sfrutti i *mebibyte* per la memoria (sappi che è fondamentalmente la stessa cosa rispetto a quando si usano i megabyte), ma puoi dare qualsiasi cosa, dai byte ai petabyte. Nel nostro caso, abbiamo impostato 6 Gibibyte, che corrispondono a 6,44 Gigabyte.

Gibibyte, che maledista

Perché non avvalersi dei Gigabyte o dei MB? Secondo la documentazione, limiti e richieste di memoria sono misurati in byte e puoi esprimere il suo valore come numero intero semplice o come numero intero in virgola fissa utilizzando uno di questi suffissi: E, P, T, G, M, K. Puoi anche utilizzare gli equivalenti potenza di due: Ei, Pi, Ti, Gi, Mi, Ki. Il tutto nasce dal kibibyte: questo è stato progettato per sostituire il kilobyte in quei contesti informatici in cui il termine kilobyte è usato per indicare 1024 byte, e dove l'interpretazione di kilobyte per denotare 1024 byte era in conflitto con la definizione SI del prefisso kilo (1000).

Una cosa da tenere a mente sulle `requests` della CPU è che se inserisci un valore maggiore del numero di core del tuo nodo più grande, il tuo Pod non verrà mai pianificato, come detto prima anche per la differenza tra i valori di `requests` e `limits`. Supponiamo che tu abbia un Pod che necessita di quattro core, ma il tuo cluster Kubernetes è composto da VM *dual core*: il tuo Pod non sarà mai pianificato. A meno che la tua app non sia specificamente progettata per sfruttare più core (mi vengono in mente il calcolo scientifico e alcuni database), di solito è consigliabile mantenere la richiesta della CPU su un core o meno ed eseguire più repliche per ridimensionarla; ciò conferisce al sistema maggiore flessibilità e affidabilità. Viceversa, proprio come la CPU, se inserisci una `request` di memoria maggiore della quantità di memoria sui tuoi nodi, il Pod non verrà mai eseguito.

Torniamo un secondo a parlare della CPU: è quando si tratta dei limiti di questa risorsa che le cose si fanno interessanti. La CPU è considerata una risorsa “comprimibile”. Se la tua app inizia a raggiungere i limiti della CPU, Kubernetes inizia a limitare il tuo container. Ciò significa che la CPU sarà limitata artificialmente, dando alla tua applicazione prestazioni potenzialmente peggiori; tuttavia, non sarà terminata o eliminata e potrai utilizzare le `probes` descritte in precedenza per assicurarti che le prestazioni non siano state influenzate.

A differenza delle risorse della CPU, la memoria non può essere compressa, per cui, poiché non è possibile limitare l'utilizzo della memoria, se un container supera il limite di memoria verrà terminato. Se il tuo Pod è gestito da un Deployment, StatefulSet, DaemonSet o un altro tipo di controller, questo si occuperà di avviare un nuovo Pod quando il vecchio sarà stato eliminato.

Come decidere quali sono i valori “giusti” da assegnare a queste risorse? Abbiamo detto che esistono due diversi tipi di configurazioni delle risorse che possono essere impostate su ciascun container di un Pod e che le richieste definiscono la quantità minima di risorse necessarie ai container.

Se ritieni che la tua applicazione richieda almeno 256 MB di memoria per funzionare, questo sarà il valore della request per la memoria. La request, infatti, stabilisce il valore minimo, ma l'applicazione potrà utilizzare più di 256 MB; Kubernetes farà solo in modo che si possa garantire un minimo di 256 MB al container. D'altra parte, i limiti definiscono la quantità massima di risorse che il container può consumare; se vuoi avere la certezza che la tua applicazione non consumi più di 1 GB di memoria, questo sarà il tuo limite. Questo vuol dire che il suo consumo nel tempo potrà crescere fino a questo limite superiore, fino a essere terminato da Kubernetes al superamento di tale soglia. L'impostazione dei limiti è utile per interrompere *l'overcommit* delle risorse e proteggere altre applicazioni dall'esaurimento delle risorse; questo serve anche per impedire a una singola applicazione di utilizzare tutte le risorse disponibili e lasciare solo *briciole di pane* al resto del cluster. In altre parole, per calcolare correttamente questi valori, l'ideale è quello di eseguire dei test di carico attraverso strumenti come Apache JMeter e raccogliere tutte le metriche in base a delle specifiche funzionalità o transazioni e poi prendere la media, il massimo e il minimo della CPU e della memoria per estrapolare requests e limits.

A onor del vero, c'è anche un altro metodo, che però viene illustrato più avanti: si tratta dell'utilizzo di una risorsa chiamata Vertical Pod Autoscaler, che è un componente che è possibile installare nel cluster e che serve a stimare le requests e limits per i diversi Pod. Grazie a questa risorsa, non devi pensare a un algoritmo per estrarre limiti e richieste, perché il Vertical Pod Autoscaler applica un modello statistico ai dati raccolti dal server delle metriche. Sembra uno strumento dai poteri magici, ma attenzione: come tutti gli automatismi, è importante conoscerne a fondo la definizione e quando utilizzarlo. Infatti, in un mondo ideale, la definizione di request e limits del container di Kubernetes sarebbero più che sufficienti per gestire le risorse del cluster nella sua interezza, ma il mondo è un luogo oscuro e terribile e le persone possono facilmente dimenticare di impostare queste risorse o un team può impostare richieste e limiti molto più alti del necessario e occupare più CPU e memoria della loro giusta "fetta" del cluster. Per evitare questi scenari, ci sono dei componenti aggiuntivi che puoi configurare, come `ResourceQuotas` e `LimitRanges`, che lavorano a livello di namespace: parleremo di questo nel Capitolo 14 dedicato alle *best practice* in un contesto enterprise.

Che cosa abbiamo imparato

- All'interno di un cluster, ci sono tantissime risorse che ci permettono di eseguire la nostra applicazione; è importante conoscere ognuna di esse e poter comprendere a pieno le differenze e le funzionalità che offrono.
- Abbiamo esplorato i Pod e il loro utilizzo di base, così come il loro ciclo di vita e la loro definizione.
- Abbiamo anche approfondito la definizione di ReplicaSet e ReplicationController, così da evidenziarne le differenze e il loro funzionamento.
- Il concetto di controller è poi fondamentale per poter gestire in maniera appropriata i diversi oggetti che gravitano all'interno del cluster: in questo capitolo ne abbiamo visto un accenno con diversi esempi tramite risorse come i Deployment, anche se questi verranno approfonditi ed esplorati successivamente nei prossimi capitoli.

Configurazione

[L'industria tecnologica] è favorevole al rischio, quindi è meglio cogliere l'occasione o prendere l'iniziativa se pensi che qualcosa possa funzionare. Provaci e mettiti davvero in gioco, perché è un settore che risponde bene al cambiamento. Quando corri dei rischi e ci provi, tendi a essere ricompensato, vedrai che le cose accadono.

– Megan Berry, Vice President of Product at Octane AI

Come avrai modo di apprezzare, in ogni capitolo di questo libro vestiremo diversi cappelli a seconda del ruolo e del componente che dovremo spiegare. Questo capitolo, in particolare, è pensato per tutte quelle persone che sviluppano e che hanno bisogno di capire profondamente il significato di ognuno degli oggetti che questa tecnologia ci mette a disposizione: ogni sezione è pensata per spiegare, attraverso moltissimi esempi, in che modo queste risorse differiscano e come possono adattarsi ai nostri casi d'uso sotto diversi punti di vista. Per il momento, potrebbe sembrare che ognuno di questi oggetti sia slegato dagli altri e che non ci sia un vero punto di connessione tra tutto ciò che vedremo: teniamo allora a mente che questa vuole essere un'introduzione agli strumenti utili nella nostra cassetta degli attrezzi, prima di dedicarci interamente a ciò che significa trasformare un'applicazione tradizionale in un'applicazione che può essere eseguita in un cluster con Kubernetes.

Risorse

Finora le applicazioni che abbiamo visto avevano dei dati “scolpiti” al loro interno tramite variabili di ambiente che permettevano, in un qualche modo, di parametrizzarne il comportamento. La realtà è che la dinamicità delle applicazioni a seconda del contesto di installazione è fondamentale ed è estremamente utile poter esternalizzare le sue configurazioni, di modo che le modifiche da apportare siano centralizzate quanto più possibile all'interno di oggetti specifici la cui funzione è proprio quella di conservare il contesto dell'applicazione. Per questa ragione ci vengono in soccorso due risorse molto utili che ci permetteranno di passare delle opzioni di configurazione alle applicazioni quando vengono eseguite in Kubernetes: parliamo di ConfigMaps e Secrets, due oggetti molto simili, ma anche molto diversi.

Facciamo un passo indietro: prima di vedere come passare dei dati di configurazione alle applicazioni in esecuzione su Kubernetes, diamo un'occhiata a come vengono solitamente configurati i container. Se si ignora il fatto che è possibile inserire la configurazione nell'applicazione stessa, quando si avvia l'installazione di un nuovo servizio, di solito si inizia configurando l'applicazione passando degli argomenti tramite riga di comando. Chiaramente, man mano che l'elenco dei parametri di configurazione cresce, diventa difficile poter eseguire un comando lungo righe ed è necessario spostare la configurazione in un file apposito.

Un altro modo molto diffuso per passare questi dati di configurazione a un'applicazione all'interno dei container è attraverso le variabili di ambiente. Invece di far leggere un file di configurazione o degli argomenti della riga di comando, l'app cerca il valore di una determinata variabile di ambiente. L'immagine del container MySQL ufficiale, per esempio, utilizza una variabile di ambiente denominata `MYSQL_ROOT_PASSWORD` per l'impostazione della password per l'account dell'utente di amministrazione

root, così come usa molte altre variabili di ambiente che permettono di cambiare la configurazione del database.

Listato 6.1 Esempio di esecuzione di un container tramite Docker passando diversi parametri

```
docker run -it --rm mysql:tag --name some-mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw -e  
MYSQL_DATABASE=mydb -e MYSQL_USER=myuser -e MYSQL_PASSWORD=mypassword
```

Ma perché le variabili d'ambiente sono così popolari nei container? L'utilizzo dei file di configurazione all'interno dei container Docker è un po' complicato, perché dovresti inserire il file di configurazione nell'immagine del container stesso o montare un volume contenente il file nel container e poi modificarlo. Inserire dei file con i dati scolpiti nell'immagine è simile alla configurazione hardcoded nel codice sorgente dell'applicazione, perché richiede di ricostruire l'immagine ogni volta che si desidera modificare la configurazione, e non rientra nelle buone pratiche. Inoltre, chiunque abbia accesso all'immagine può vedere la configurazione, incluse tutte le informazioni che devono essere mantenute segrete, come credenziali o chiavi di crittografia.

Una premessa è doverosa: le risorse che vedremo adesso sono perfette quando abbiamo a che fare con dati che possono cambiare a seconda del contesto, come le opzioni della JVM (per chi sviluppa in Java), oppure opzioni che cambiano a seconda dell'ambiente, come i dati di connessione a un database: questo non vuol dire che le variabili di ambiente o i comandi aggiuntivi sono banditi, ma piuttosto che abbiamo la possibilità di rendere più modulare l'applicazione che installeremo su Kubernetes.

ConfigMap

Abbiamo detto che il punto centrale della configurazione di un'applicazione è mantenere i dati di configurazione, che variano tra gli ambienti o che cambiano frequentemente, separati dal codice sorgente dell'applicazione. Kubernetes consente di separare le opzioni di configurazione in un oggetto separato chiamato ConfigMap, che è di base una mappa (o un dizionario, a seconda della scuola di programmazione da cui vieni) contenente coppie chiave/valore con valori che possono essere sia letterali brevi sia file di configurazione completi. Questo tipo di oggetto è perfetto per memorizzare informazioni generiche di configurazione e non "sensibili", come file di properties, configurazioni applicative o argomenti utili ai processi in esecuzione sul container.

Il vantaggio sta nel fatto che un'applicazione non ha bisogno di leggere direttamente la ConfigMap o addirittura sapere che esiste, perché il contenuto di questa risorsa viene invece passato ai container come variabili di ambiente o come file tramite un volume; poiché è anche possibile fare riferimento alle variabili di ambiente negli argomenti della riga di comando utilizzando la sintassi `$ (VARIABILE_AMBIENTE)`, è anche possibile passare i valori della ConfigMap ai processi come argomenti della riga di comando.

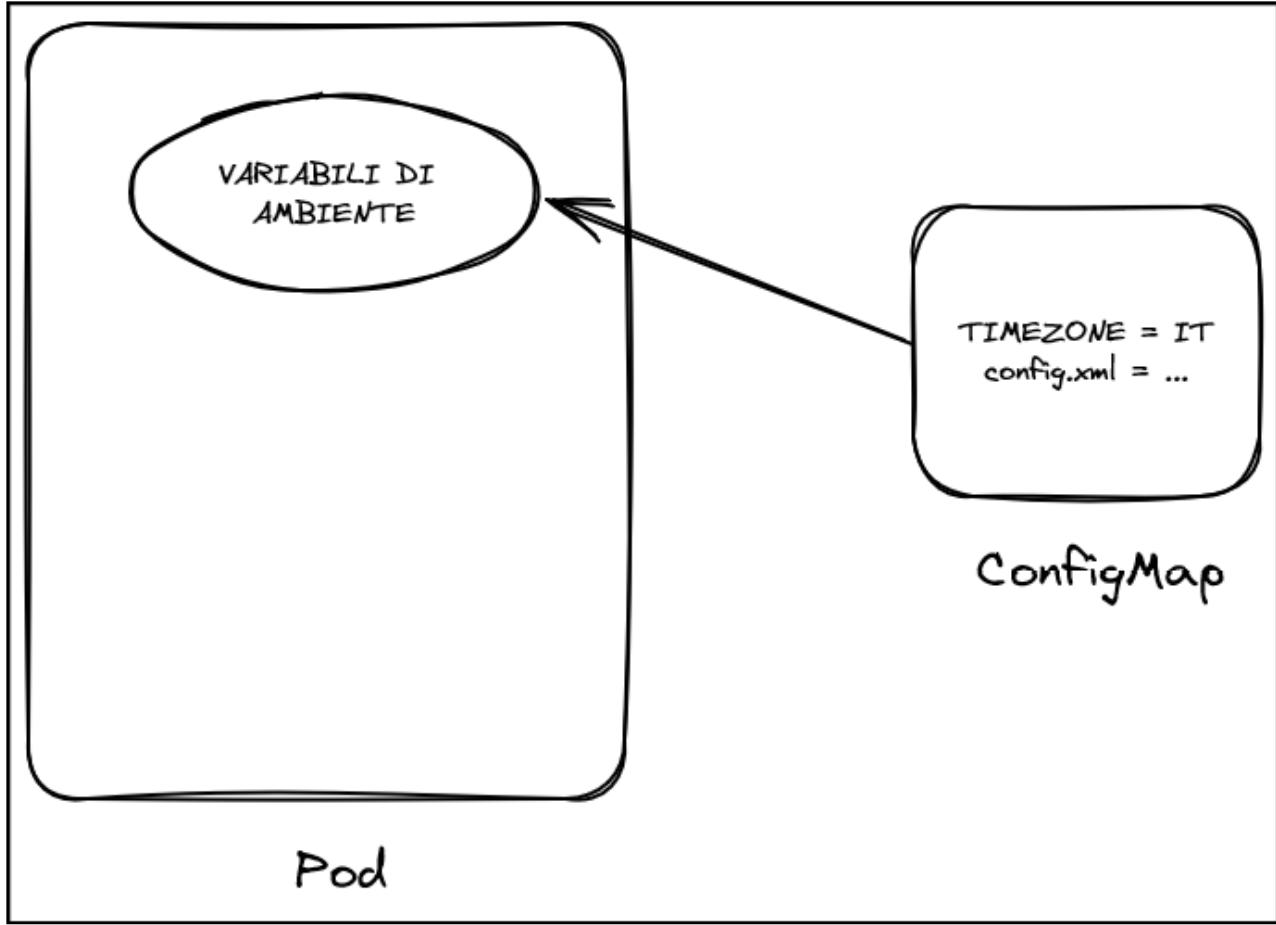


Figura 6.1 Esempio di relazione tra un Pod e una ConfigMap.

Indipendentemente da come un'applicazione utilizza una ConfigMap, avere la configurazione in un oggetto separato come questo ti consente di mantenere più configurazioni diverse in ConfigMap con lo stesso nome, ciascuna per un ambiente diverso, come sviluppo, test, QA, produzione e così via. Questo vuol dire che per esempio potremo creare, in due namespace diversi, due Pod a partire dallo stesso file che leggono da due ConfigMap con due configurazioni diverse, a seconda del contesto:

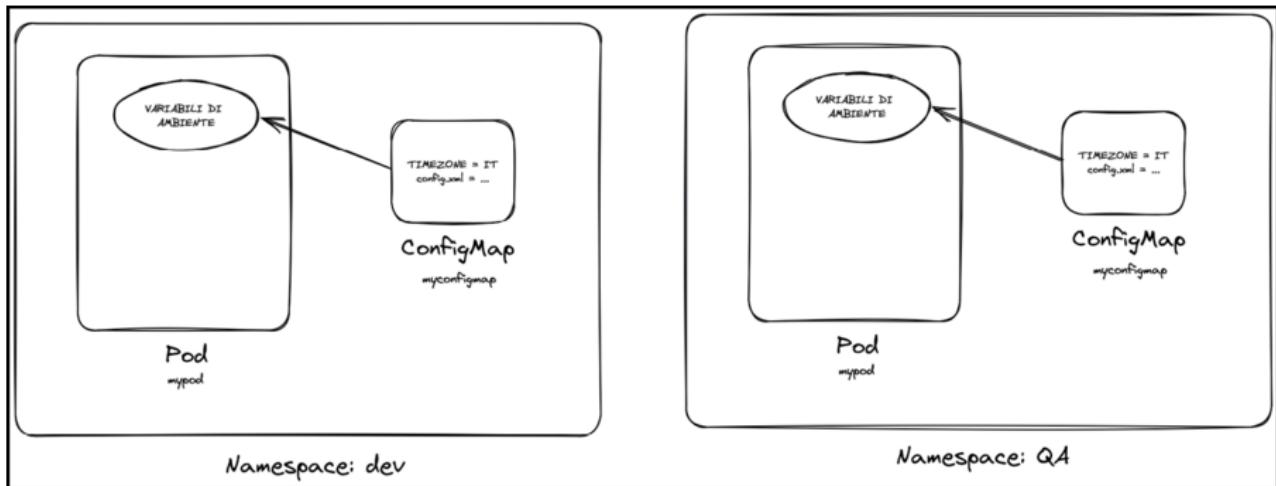


Figura 6.2 Esempio di funzionamento di una ConfigMap su namespace diversi, con specifiche diverse.

Vediamo un esempio di ConfigMap: come sempre, sfruttiamo un file YAML per definirla e ne analizziamo la struttura. Il caso che utilizziamo è molto semplice, e serve a memorizzare la timezone utilizzata dall'applicazione in una variabile d'ambiente chiamata `TIMEZONE` con un valore corrispondente a una stringa.

Listato 6.2 Esempio di ConfigMap

```
apiVersion: v1
data:
  TIMEZONE: "IT"
kind: ConfigMap
metadata:
  name: myconfigmap
```

Come prima informazione, notiamo che l'API di riferimento è sempre `v1`: dopo aver inserito come valore del campo `kind` quello corrispondente a `ConfigMap`, notiamo che questa risorsa ha un nome assegnato all'interno dei metadati, che verrà poi utilizzato per avere un riferimento per l'oggetto applicativo che andrà a leggerne i valori. Il cuore della ConfigMap sta nel campo `data`: all'interno di questo potremo infatti definire tutte le coppie chiave/valore che ci servono per avere la configurazione pronta per il nostro Pod.

Naming convention

Questo tipo di risorsa deve avere un nome associato che segua alcune regole: tra queste, c'è l'utilizzo di caratteri alfanumerici, trattini (“-”), trattini bassi (“_”) e punti. Un nome valido è quindi `my-configmap`, ma non `my!configmap`.

Per salvare un file all'interno della ConfigMap, che sia un file di properties o anche un xml, possiamo utilizzare sempre il campo `data`, tenendo presente che per gestire il multi-riga avremo bisogno di utilizzare questo trucco: come chiave utilizzeremo il nome del file, mentre come valore inseriremo “`| -`” subito prima di inserire il contenuto del file:

Listato 6.3 Esempio di ConfigMap con file

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: myconfigmap-files
data:
  locale.properties: |-
    # ##### Locale properties compliant with the standard Java i18n mechanism
    #
    # see https://docs.oracle.com/javase/8/docs/api/java/util/Locale.html
    # for extensive explanations
    #
    # the file provides default empty values for each property
    # #####
language = @locale.properties.language@
country = @locale.properties.country@
variant = @locale.properties.variant@
...
```

Definire una ConfigMap è piuttosto facile: ma come crearla all'interno del namespace? Tramite il comando `kubectl create`:

Listato 6.4 Creazione di una ConfigMap a partire dal file YAML

```
kubectl create -f myconfigmap.yaml
```

Questi sono solo alcuni esempi di come scrivere da zero una ConfigMap, ma è possibile anche crearne una a partire da un file di configurazione già presente o da riga di comando: nel primo caso, possiamo per esempio prendere il file di configurazione riportato e utilizzare `kubectl create configmap` come comando per creare questa risorsa, come mostrato di seguito:

Listato 6.5 File di configurazione logrotate.conf

```
# see "man logrotate" for details
# rotate log files weekly
weekly

# use the adm group by default, since this is the owning group
# of /var/log/syslog.
su root adm

# keep 4 weeks worth of backlogs
rotate 4

# create new (empty) log files after rotating old ones
create

# use date as a suffix of the rotated file
#dateext

# uncomment this if you want your log files compressed
#compress

# packages drop log rotation information into this directory
include /etc/logrotate.d

# system-specific logs may be also be configured here.
```

Listato 6.6 Creazione di una ConfigMap a partire dal file logrotate.conf

```
kubectl create configmap myconfigmap-logrotate --from-file=logrotate.conf
```

Quando esegui il comando precedente, kubectl cerca il file `logrotate.conf` nella directory in cui stai eseguendo il comando e poi crea un qualcosa di simile a questo:

Listato 6.7 Esempio di ConfigMap con il file logrotate.conf

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: myconfigmap-files
data:
  logrotate.conf: |-
    # see "man logrotate" for details
    # rotate log files weekly
    weekly

    # use the adm group by default, since this is the owning group
    # of /var/log/syslog.
    su root adm

    # keep 4 weeks worth of backlogs
    rotate 4

    # create new (empty) log files after rotating old ones
    create

    # use date as a suffix of the rotated file
    #dateext
```

```

# uncomment this if you want your log files compressed
#compress

# packages drop log rotation information into this directory
include /etc/logrotate.d

# system-specific logs may be also be configured here.

```

In maniera analoga, è possibile creare una ConfigMap a partire da coppie chiave/valore singole, passate da riga di comando, in questo modo:

Listato 6.8 Esempio di ConfigMap con delle variabili

```
kubectl create configmap myconfigmap --from-literal=foo=bar --from-literal=bar=baz --from-literal=one=two
```

Tutte le opzioni viste finora possono anche essere combinate: se volessimo creare un'unica ConfigMap con dei file di configurazioni e delle variabili utili all'applicazione riutilizzando gli esempi precedenti, potremmo scrivere un comando come il seguente:

Listato 6.9 Esempio di ConfigMap con delle variabili

```
kubectl create configmap myconfigmap \
    --from-literal=foo=bar \
    --from-literal=bar=baz \
    --from-literal=one=two \
    --from-file=logrotate.conf
```

Ora che abbiamo visto come creare una ConfigMap, è il momento di passare all'azione con un esempio pratico: vediamo come utilizzarla all'interno di un'applicazione, per capire meglio come questo tipo di oggetto possa essere "consumato" e quali strategie abbiamo a disposizione. Infatti, per poter disporre dei valori contenuti in una ConfigMap, abbiamo a disposizione due metodi: uno prevede di fornire il contenuto della ConfigMap come elenco di variabili di ambiente a disposizione della risorsa che lo richiede, mentre l'altra strategia prevede di "montare" il contenuto come uno o più file. Se questo può sembrare difficile, proviamo con un esempio: definiamo una ConfigMap che contenga una variabile di ambiente per gestire i tipi di contesto che abbiamo a disposizione (chiamata environments) e una porta, in questo modo:

Listato 6.10 ConfigMap di esempio

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  port: "3000"
  environments: production, development, production

```

Listato 6.11 Creazione della ConfigMap

```
kubectl create configmap --from-file=app-config.yaml
```

Listato 6.12 Elenco delle ConfigMap presenti

```

kubectl get configmaps
>>>
NAME          DATA   AGE
kube-root-ca.crt  1     11d
my-files      2     27m

```

Creiamo ora il Pod di test sfruttando la semplice immagine di busybox che come unico compito abbia quello di elencare le variabili di ambiente a sua disposizione, tramite il comando `env`, espresso nel

campo `command`: le variabili di ambiente (campo `env`) PORT e ENVIRONMENTS andranno a leggere il valore relativo (campo `valueFrom`) nella ConfigMap creata in precedenza il cui nome è `app-config` (campo `name`), prendendo la coppia la cui chiave è, rispettivamente, `port` e `environments` (campo `key`).

Listato 6.13 Creazione del Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: PORT
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: port
        - name: ENVIRONMENTS
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: environments
  restartPolicy: Never
```

Creando questo Pod, e recuperandone i log, questo è il risultato che otteniamo:

Listato 6.14 Log del Pod

```
kubectl logs busybox
>>>
KUBERNETES_PORT=tcp://10.100.0.1:443
KUBERNETES_SERVICE_PORT=443
HOSTNAME=busybox
SHLVL=1
HOME=/root
PORT=3000
ENVIRONMENTS=production, development, production
KUBERNETES_PORT_443_TCP_ADDR=10.100.0.1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT_443_TCP=tcp://10.100.0.1:443
PWD=
KUBERNETES_SERVICE_HOST=10.100.0.1
```

Trattandosi di variabili di ambiente, come output del comando `env` otteniamo anche variabili che non abbiamo direttamente configurato tramite la ConfigMap, ma che comunque sono messe a disposizione per il Pod.

Questa è la prima strategia: per quanto riguarda invece l'utilizzo di uno o più file memorizzati all'interno di una ConfigMap, possiamo comunque sfruttare un esempio molto simile, dove però modifichiamo leggermente i due file YAML: nel caso della ConfigMap, inseriremo come dati il nome del file e il relativo contenuto, come visto anche negli esempi in precedenza.

Listato 6.15 ConfigMap di esempio con file

```
kind: ConfigMap
apiVersion: v1
metadata:
```

```

name: my-files
data:
  user.properties: >-
    myproperty=value
  simple-text.txt: >-
    Hello world!

```

Per quanto riguarda il Pod, andiamo invece a leggere la ConfigMap tramite il filesystem del container: come evidenziato nell'esempio seguente, è necessario prima specificare come questi file devono essere "montati" e aggiunti nel container (tramite `volumeMounts`), assegnando un nome al volume (campo `name`) e specificando in che cartella ci aspettiamo di trovarli. Andiamo poi a indicare nella definizione del Pod qual è la ConfigMap di riferimento per recuperare questi valori (campo `volumes`), specificando il nome del volume (pari al campo `name` visto poco fa) e il nome della ConfigMap relativa:

Listato 6.16 Pod con ConfigMap in cui sono presenti dei file

```

apiVersion: v1
kind: Pod
metadata:
  name: busybox
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ «/bin/sh», «-c», «ls -la /etc/config» ]
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: my-files
  restartPolicy: Never
>>>
total 0
drwxrwxrwx      3 root  root          106 Feb  1 15:54 .
drwxr-xr-x      1 root  root           20 Feb  1 15:54 ..
drwxr-xr-x      2 root  root           52 Feb  1 15:54 ..2023_02_01_15_54_14.3139671222
lrwxrwxrwx 1 root  root            32 Feb  1 15:54 ..data ->
..2023_02_01_15_54_14.3139671222
lrwxrwxrwx      1 root  root           22 Feb  1 15:54 simple-text.txt -> ..data/simple-
text.txt
lrwxrwxrwx      1 root  root           22 Feb  1 15:54 user.properties ->
..data/user.properties

```

Dove vengono memorizzate le ConfigMap (e anche i Secret)?

Kubernetes memorizza oggetti come ConfigMap e Secret all'interno di `etcd`: questo è essenzialmente il cervello di Kubernetes, poiché memorizza tutti gli oggetti valore-chiave richiesti da Kubernetes per orchestrare i container. Il problema con l'utilizzo di `etcd`, tuttavia, è che limita la dimensione dei dati che possono essere archiviati: infatti, la dimensione massima per questo tipo di oggetti è di 1 MB.

Quanto visto finora per la lettura delle ConfigMap non vale solo per i Pod, ma per tutte le risorse che ne devono disporre dal lato applicativo: Deployment, DaemonSet, StatefulSet e così via. La sintassi per il recupero dei valori rimane invariata, come mostrato nel seguente esempio: in questo caso, andiamo a leggere un file di configurazione presente in una ConfigMap chiamata `nginx-conf` che al suo interno ha diverse proprietà, tra cui un file di configurazione `nginx.conf`, che individuiamo singolarmente tramite il campo `items`. Quest'opzione ci permette infatti di sfruttare alcuni dei valori della ConfigMap come variabili di ambiente, e altri come file:

Listato 6.17 ConfigMap di esempio con parametri misti

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: my-files
data:
  nginx.conf: >-
    # The identifier Backend is internal to nginx, and used to name this specific
  upstream
  upstream Backend {
    # hello is the internal DNS name used by the backend Service inside Kubernetes
    server dsp;
  }

  server {
    listen 80;

    location / {
      # The following statement will proxy traffic to the upstream named Backend
      proxy_pass http://Backend;
    }
  }

  simple-text.txt: >-
Hello world!
```

Listato 6.18 Deployment con ConfigMap

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  namespace: sandbox
spec:
  selector:
    matchLabels:
      run: nginx
      app: dsp
      tier: frontend
  replicas: 2
  template:
    spec:
      containers:
        - name: nginx
          image: nginx
          volumeMounts:
            - mountPath: /etc/nginx
              name: nginx-conf
  volumes:
    - name: nginx-conf
      configMap:
        name: nginx-conf
        items:
          - key: nginx.conf
            path: nginx.conf
```

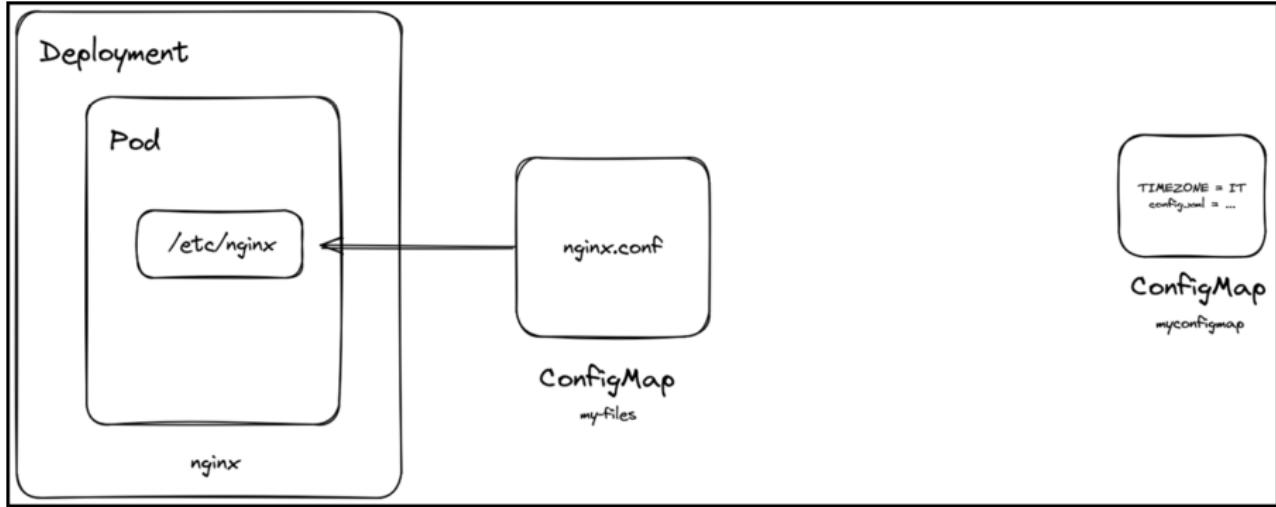


Figura 6.3 Rappresentazione di un Deployment con la relativa ConfigMap.

Finora potrebbe sembrare che questa risorsa sia più che altro un modo per strutturare bene il nostro progetto, ma che non porti alcun vantaggio significativo: in realtà, esternalizzando la configurazione in questo modo, abbiamo la possibilità di modificare qualsiasi valore contenuto al suo interno anche quando il Pod è già in esecuzione, senza doverlo riavviare manualmente. Se prendessimo l'esempio precedente, e andassimo a modificare la ConfigMap tramite il comando `kubectl edit`, dopo qualche istante, noteremmo che il file è stato aggiornato correttamente:

Listato 6.19 Modifica alla ConfigMap di esempio

```

kind: ConfigMap
apiVersion: v1
metadata:
  name: my-files
data:
  nginx.conf: >-
      # The identifier Backend is internal to nginx, and used to name this specific
      upstream
      upstream Backend {
          # hello is the internal DNS name used by the backend Service inside Kubernetes
          server dsp;
      }

      server {
          listen 443;

          location / {
              # The following statement will proxy traffic to the upstream named Backend
              proxy_pass http://Backend;
          }
      }

  simple-text.txt: >-
Hello world!

```

Listato 6.20 Recupero del contenuto del file aggiornato tramite il terminale del Pod

```

kubectl exec nginx -c nginx cat /etc/nginx/nginx.conf
>>>
# The identifier Backend is internal to nginx, and used to name this specific upstream
upstream Backend {
    # hello is the internal DNS name used by the backend Service inside Kubernetes

```

```

        server dsp;
    }

server {
    listen 443;

    location / {
        # The following statement will proxy traffic to the upstream named Backend
        proxy_pass http://Backend;
    }
}

```

Se non vedi immediatamente l'aggiornamento, attendi qualche istante e riprova, dal momento che richiede un po' di tempo. Nel caso di esempio è utile fare una precisazione: vedrai la modifica nel file di configurazione sulla porta esposta, ma questo non ha alcun effetto su Nginx, perché non guarda i file e li ricarica automaticamente.

Secret

Ora che abbiamo visto come memorizzare informazioni generiche di configurazione tramite una ConfigMap, vediamo dove mettere al sicuro i dati "sensibili": per questo, ci vengono in soccorso oggetti come i Secret, che ci permettono di salvare credenziali, certificati, password e altre informazioni che non devono essere condivise. I Secret sono molto simili alle ConfigMap: sono anche loro mappe che contengono delle coppie chiave-valore e possono essere utilizzati nello stesso modo di una ConfigMap, quindi è possibile passare i valori del Secret al container come variabili di ambiente o esporli come file in un volume. Al contrario delle ConfigMap, Kubernetes aiuta a proteggere i Secret assicurandosi che ciascuno di essi sia distribuito solo nei nodi che eseguono i Pod e che necessitano dell'accesso al Secret. Inoltre, sui nodi stessi, i Secret sono sempre archiviati in memoria e mai scritti su un archivio fisico; infatti, a partire dalla versione 1.7 di Kubernetes, etcd archivia i Secret in forma crittografata, rendendo il sistema molto più sicuro.

L'aspetto di un file che definisce un Secret è molto simile a quello di una ConfigMap, tranne per qualche dettaglio: prendiamo questo Secret di esempio.

Listato 6.21 Esempio di Secret

```

kind: Secret
apiVersion: v1
metadata:
  name: mysecret
data:
  https.cert: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURCeKNDQ...
type: Opaque

```

Noti la differenza? I contenuti delle chiavi del campo data di un Secret sono mostrati come stringhe codificate in Base64, mentre quelli di una ConfigMap sono mostrati in chiaro. Può sembrare complesso ma questa scelta deriva dal fatto che si possano memorizzare all'interno dei Secret anche dati binari: i dati che memorizziamo possono infatti contenere molti caratteri non riconosciuti, e codificarli con un formato così noto è conveniente. Questo non è dunque un modo per renderli più "sicuri", in quanto con Base64 si sta codificando, e non crittografando: questo consente di codificare le informazioni in modo più semplice.

In effetti, è possibile decodificare una stringa con un qualsiasi decoder (anche online), come visibile in questo esempio:

Listato 6.22 Decodifica di una stringa in formato Base64

```

echo ZXhwB3NlZC1wYXNzd29yZAo= | base64 --decode
> exposed-password

```

Nel creare una stringa, è tuttavia possibile inserire i dati utilizzando un formato semplice: questo può avvenire sostituendo, al posto del campo data, il campo `stringData` (Listato 6.23).

Listato 6.23 Esempio di Secret con stringData

```
kind: Secret
apiVersion: v1
metadata:
  name: my-secret
  stringData:
    foo: test
```

Il campo `stringData` è di sola scrittura, e questo vuol dire che può essere utilizzato solo per impostare i valori. Quando si recupera lo YAML del Secret con il comando `kubectl get secret`, il campo `stringData` non verrà mostrato. Invece, tutte le voci specificate nel campo `stringData` (come la voce `foo` nell'esempio precedente) verranno mostrate sotto data e saranno codificate in Base64 come tutte le altre voci.

Listato 6.24 Output del secret dopo la creazione

```
kubectl get secret my-secret -o yaml -n k8s-training
>>>
apiVersion: v1
kind: Secret
metadata:
...
  name: my-secret
  data:
    foo: dGVzdA==
type: Opaque
```

Per poter usare un Secret in una risorsa applicativa (un controller, o semplicemente un Pod), le operazioni sono del tutto simili a quanto visto per la ConfigMap: vediamo un esempio con la creazione di Postgres come database attraverso un Deployment per quanto riguarda il Pod, e un Secret che contiene le variabili necessarie per inizializzare un database, con relativo utente e password:

Listato 6.25 Secret per l'istanza di PostgreSQL

```
kind: Secret
apiVersion: v1
metadata:
  name: postgresql
data:
  database-name: c2FtcGx1ZGI=
  database-password: ZFRXYlZPZWtwTkpjZU5EZw==
  database-user: dXNlckdKTg==
type: Opaque
```

Listato 6.26 Deployment per l'istanza di PostgreSQL

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgresql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgresql
  template:
    metadata:
      labels:
        app: postgresql
```

```

spec:
  volumes:
    - name: postgresql-data
      emptyDir: {}
  containers:
    - env:
        - name: POSTGRES_DB
          valueFrom:
            secretKeyRef:
              name: postgresql
              key: database-name
        - name: POSTGRES_USER
          valueFrom:
            secretKeyRef:
              name: postgresql
              key: database-user
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: postgresql
              key: database-password
    ports:
      - containerPort: 5432
        protocol: TCP
  volumeMounts:
    - name: postgresql-data
      mountPath: /var/lib/pgsql/data
  image: postgres

```

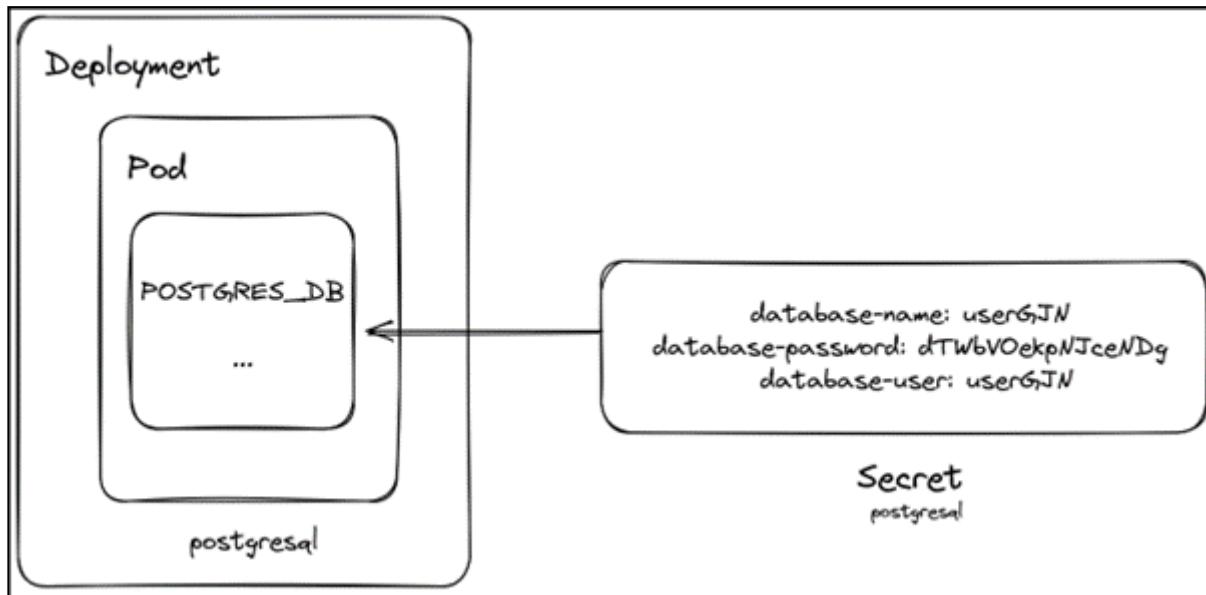


Figura 6.4 Esempio di relazione tra un Pod e un Secret.

L'applicazione verrà quindi istanziata sfruttando le variabili di ambiente messe a disposizione dall'immagine e personalizzate con i valori specificati: per fare un test e verificare che il database specificato esista e che sia accessibile all'utente, dopo aver creato il Deployment tramite `kubectl create`, eseguiamo, tramite terminale, il seguente comando:

Listato 6.27 Creazione delle risorse

```

kubectl create -f secret.yaml
kubectl create -f deployment.yaml

```

Listato 6.28 Connessione al database

```
kubectl exec postgresql-7bf4457f99-k8m95 -it /bin/sh -n k8s-training
>>>
# psql -U userGJN -d sampledb
psql (15.1 (Debian 15.1-1.pgdg110+1))
Type "help" for help.

sampledb=#
```

La connessione è avvenuta con successo! Ora che abbiamo aperto il terminale, facciamo un'altra verifica: elenchiamo tutte le variabili di ambiente presenti nel container e cerchiamo quelle relative a Postgres (dopo essere usciti da psql tramite il comando exit!):

Listato 6.29 Elenco variabili di ambiente

```
env | grep -i postgres_
>>>
POSTGRES_PASSWORD=dTWbVOekpNJceNDg
POSTGRES_USER=userGJN
POSTGRES_DB=sampledb
```

Al contrario di quanto avviene tramite il comando `kubectl get`, i valori impostati per queste variabili di ambiente non sono codificati in Base64 e abbiamo invece i valori attuali dei diversi campi.

Sssh, non dirlo a nessuno...

Potresti notare che all'interno del namespace esiste un Secret che inizia per `default-token`. Alcune risorse in Kubernetes hanno un Secret collegato che contiene tre voci: `ca.crt`, `namespace` e `token`; questi rappresentano tutto ciò di cui hai bisogno per comunicare in modo sicuro con il server API Kubernetes dall'interno dei tuoi pod.

Che cosa abbiamo imparato

- Che cosa rappresenta una configurazione per le risorse Kubernetes e come esternalizzarla tramite le apposite risorse.
- Quali sono le informazioni che una ConfigMap può ospitare, e quali sono i suoi limiti.
- Che cosa sono i Secret, come utilizzarle e quando, attraverso degli esempi rapidi.

Rete

È più facile chiedere perdono che ottenere il permesso.

– Grace Hopper, ufficiale della Marina degli Stati Uniti e una delle prime programmatrici

Questo è uno degli argomenti più difficili per qualsiasi persona provenga dal mondo dello sviluppo: comprendere in che modo diverse applicazioni comunicano, con che modalità e quali sono i protocolli a disposizione è sicuramente qualcosa di “estraneo” a chi finora si è occupato dello sviluppo di un’applicazione. Così, perché non toglierci il dente adesso? Prima di proseguire con i prossimi capitoli, che ci permetteranno di mettere a terra una serie di esempi pratici dove molti di questi concetti saranno dati per scontati, facciamo una panoramica piuttosto ampia delle modalità con cui Kubernetes ci permette di creare una connessione tra i diversi Pod all’interno del cluster, ma anche fuori da questo.

Come funziona: le basi

Prima di parlare di quali sono le risorse che Kubernetes ci mette a disposizione, parliamo di quelle che sono le tipologie di comunicazione che vogliamo poter implementare grazie al cluster, e queste sono essenzialmente quattro:

- comunicazione tra container nello stesso Pod;
- comunicazione tra Pod sullo stesso nodo;
- comunicazione tra Pod su nodi diversi;
- comunicazione tra Pod e l’esterno.

Partiamo dal primo scenario: Immaginiamo di avere a disposizione due container che sono in esecuzione nello stesso Pod. Come fanno a comunicare l’uno con l’altro? Questo avviene grazie all’interfaccia *localhost* e alle porte che sono state aperte. Non cambia molto rispetto alla situazione in cui abbiamo a che fare con diversi servizi o diverse applicazioni sul nostro laptop: questa comunicazione è possibile grazie al fatto che i container si trovano nello stesso Pod, e questo vuol dire che condividono anche la rete in termini di risorse, per cui parliamo di *network namespace*. Con questo termine intendiamo la connessione tra due servizi grazie alle interfacce di rete e alle tabelle di routing che contengono le istruzioni su dove inviare le request, dal momento che queste condividono lo stesso namespace. Abbiamo infatti accennato al fatto che un namespace all’interno di un cluster rappresenta una sorta di host separato dagli altri e che quindi ha una propria rete all’interno della quale le sue risorse riescono a comunicare in maniera sicura. Questo vuol dire che ogni Pod ha la sua rete di connessione e che quindi i container al suo interno condividono questa rete: possiamo rappresentare la situazione attuale come nella Figura 7.1.

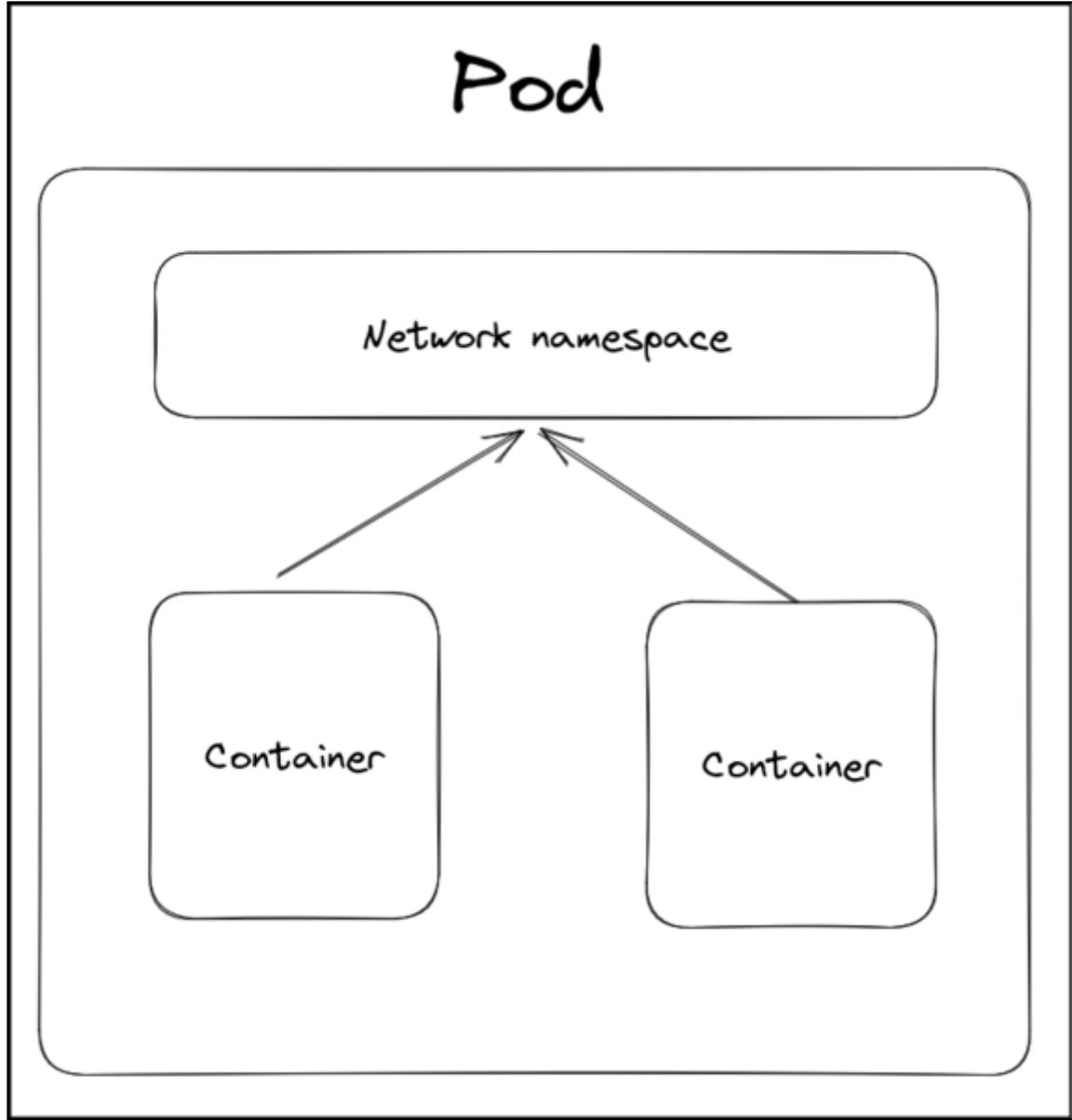


Figura 7.1 Scenario in cui i container condividono la stessa rete.

Passiamo al secondo scenario, e parliamo di due Pod che si trovano sullo stesso nodo del cluster: per quel che ci siamo detti prima, ogni Pod su ogni nodo al suo network namespace. Non solo: ogni Pod ha il suo indirizzo IP e quindi ogni Pod è collegato a una sua interfaccia di rete virtuale che gli permette di collegarsi al nodo che lo ospita. Questo vuol dire che quando un Pod esegue una richiesta all'indirizzo IP di un altro nuovo nodo utilizza la sua interfaccia di rete e quella del nodo. Proviamo a entrare un pochino di più nel dettaglio con qualche figura che ci aiuti a spiegare meglio questi concetti: immaginiamo infatti che ogni Pod abbia un'interfaccia di rete chiamata `eth0` e che quindi ogni Pod esegua le proprie request stia attraverso questa interfaccia: ogni Pod è connesso alla rete del nodo tramite un'altra interfaccia chiamata `vethX`, dove la `X` rappresenta ogni Pod. Per far sì che queste due interfacce di rete possano comunicare, viene utilizzato quello che si chiama *bridge*: un *network bridge* serve a collegare due reti insieme. quando una request arriva al bridge, questo chiede a tutti i suoi dispositivi connessi di poter indirizzare la richiesta al giusto indirizzo IP.

Prendiamo l'esempio in Figura 7.2 e immaginiamo che il Pod 1 abbia la sua network namespace e la sua interfaccia `eth0` il cui indirizzo IP è 10.65.0.1: quando avrà bisogno di comunicare con il Pod 2, il cui indirizzo è 10.65.0.2, il bridge andrà a cercare all'interno dell'elenco delle sue interfacce virtuali quella con l'indirizzo IP che il primo Pod sta cercando di contattare e a quel punto sarà in grado di indirizzare la richiesta verso il Pod corretto grazie all'interfaccia `eth0` del Pod 2.

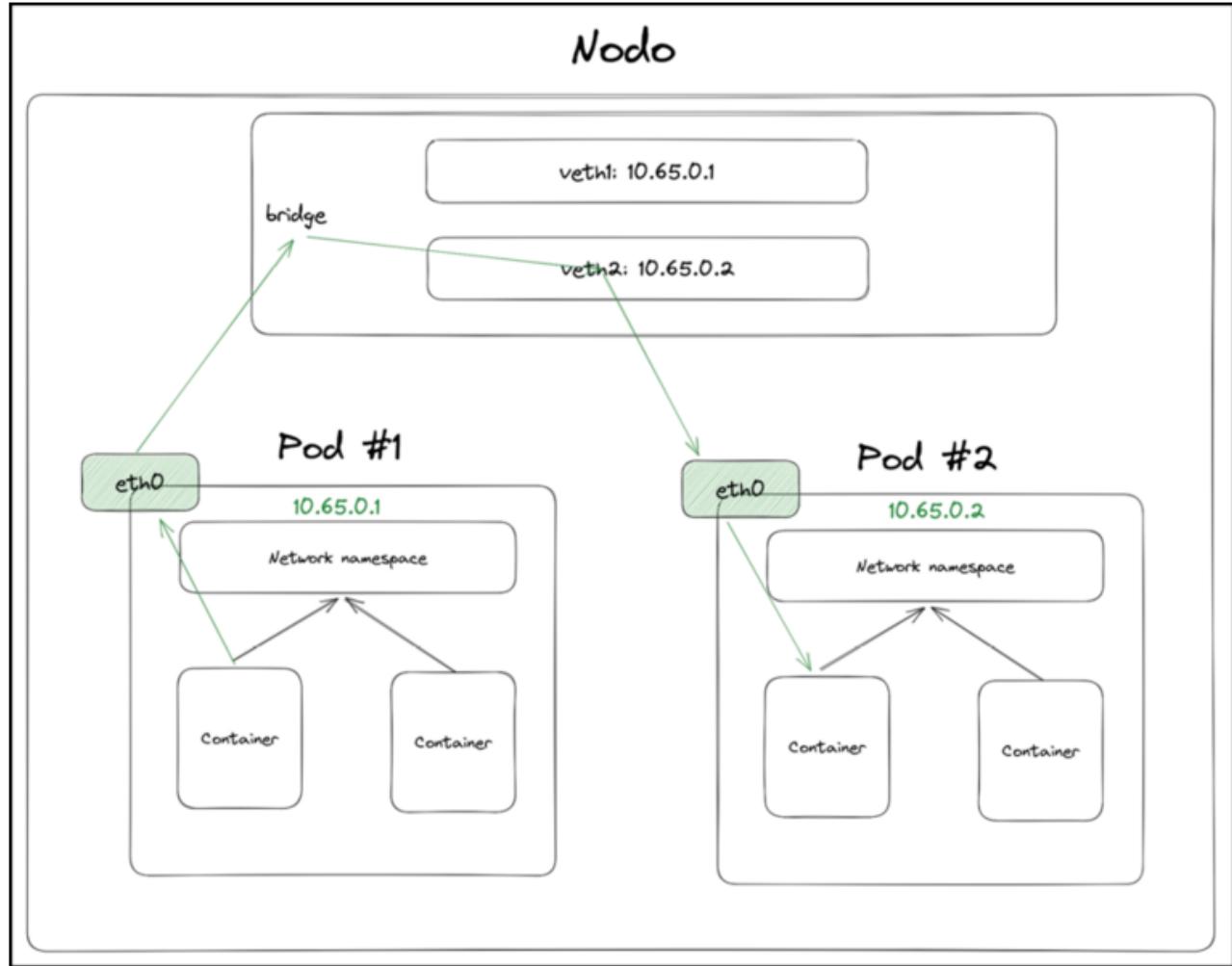


Figura 7.2 Funzionamento della comunicazione tra due container su due Pod diversi attraverso un semplice bridge.

Il prossimo scenario prevede invece che si abbiano due Pod su nodi diversi: se facciamo riferimento alla figura appena vista e immaginiamo il bridge alla ricerca di un'interfaccia che abbia il corretto indirizzo IP, ci possiamo immaginare che nessuna di queste corrisponderà a quelle presenti nell'elenco. A questo punto il bridge sarà costretto a rivolgersi al default gateway, il quale ha una visione più ampia della rete a livello di cluster e può cercare l'indirizzo IP giusto. Infatti, a livello di cluster esiste una tabella che è in grado di mappare ogni indirizzo IP ai diversi nodi presenti a seconda di alcuni range: non a caso, i Pod sullo stesso nodo avranno assegnato un indirizzo IP che appartiene allo stesso range.

Per esempio, Kubernetes potrebbe fornire ai Pod che si trovano sul nodo 1 indirizzi IP come 10.65.1.1, 10.65.1.2, e via dicendo, mentre quelli sul nodo 2 potrebbero avere 10.65.2.1, 10.65.2.2 e così via; quindi, la tabella di routing memorizzerà il fatto che le richieste verso gli indirizzi IP che assomigliano a 10.65.1.xxx dovranno andare al nodo 1 e gli indirizzi come 10.65.2.xxx devono andare al nodo 2.

Sistemista cercasi

Hey, tu! Sistemista: sappi che gli indirizzi IP mostrati qui sono puramente casuali e servono solo a scopo di esempio. Vedremo solo successivamente quando entreremo nel dettaglio delle risorse di rete di Kubernetes se ci sono dei range specifici e per quali risorse.

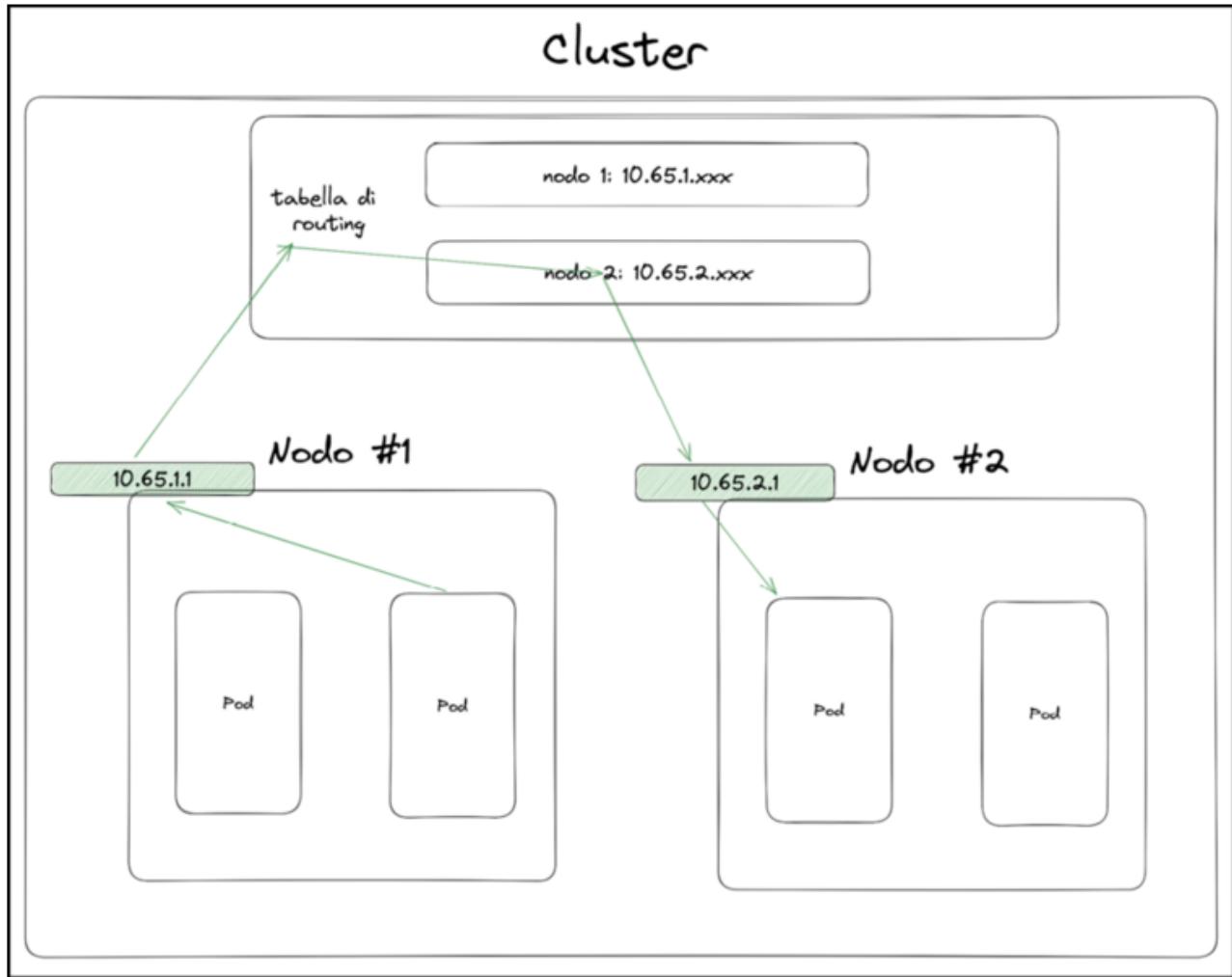


Figura 7.3 Comunicazione tra due Pod su due nodi diversi tramite le tabelle di routing.

L'ultimo tipo di scenario riguarda la comunicazione tra un Pod e alcune risorse che vedremo nel dettaglio a breve, come i Service e gli Ingress. Detto in poche parole, in Kubernetes, un Service ci permette di mappare un singolo indirizzo IP a un insieme di Pod, di modo che sia possibile eseguire una richiesta a un singolo endpoint (un singolo dominio) e sarà poi il Service a gestire la richiesta nel modo che ritiene più opportuno. Tutto questo avviene grazie al componente `kube-proxy`, un processo in esecuzione su ogni nodo del cluster che serve a mappare degli indirizzi IP virtuali a degli indirizzi IP reali associati ai Pod.

Tipologie di Service

Ora che abbiamo fatto una panoramica abbastanza ampia di come funziona la comunicazione, cerchiamo di vedere nel dettaglio quali sono le risorse che ci mette a disposizione questa tecnologia e in che modo differiscono le une dalle altre, e partiamo dal concetto di Service: ci sono quattro tipologie principali, dove *ClusterIP* rappresenta il Sacro Graal: ognuno di questi ha un significato ben specifico

all'interno della rete Kubernetes, anche se per il momento cercheremo di astrarci il più possibile per parlare dell'oggetto Service da un punto di vista più generale.

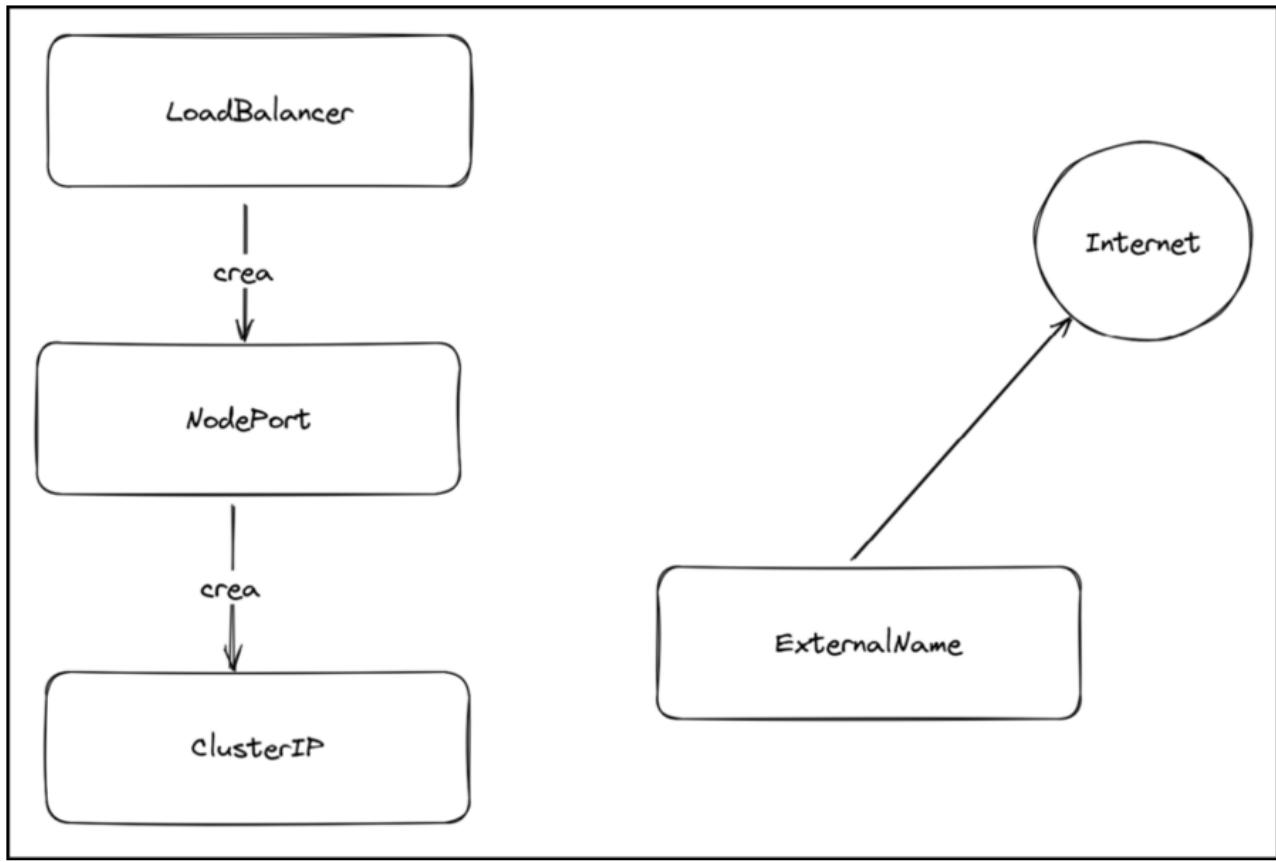


Figura 7.4 Tipologie di Service.

Per farlo, prendiamo un esempio molto semplice: immaginiamo di avere due Pod diversi che vogliono comunicare tra loro, e al momento non ci importa se siano o meno sullo stesso nodo: la cosa importante è che sono due entità diverse, con due indirizzi IP diversi. Come fanno a comunicare? Abbiamo detto più volte che i Pod sono oggetti transienti, e questo vuol dire che possono essere creati e distrutti per diverse ragioni: questo vuol dire che, nella maggior parte dei casi, il loro indirizzo IP potrà cambiare ed essere assegnato in maniera casuale; potremmo quindi avere uno scenario come il seguente, dove il primo Pod ha indirizzo IP 1.1.1.1 e il secondo 1.1.1.2. Per far comunicare i due, potremmo far riferimento a questi indirizzi, e quindi fornire a ognuno l'indirizzo IP dell'altro.

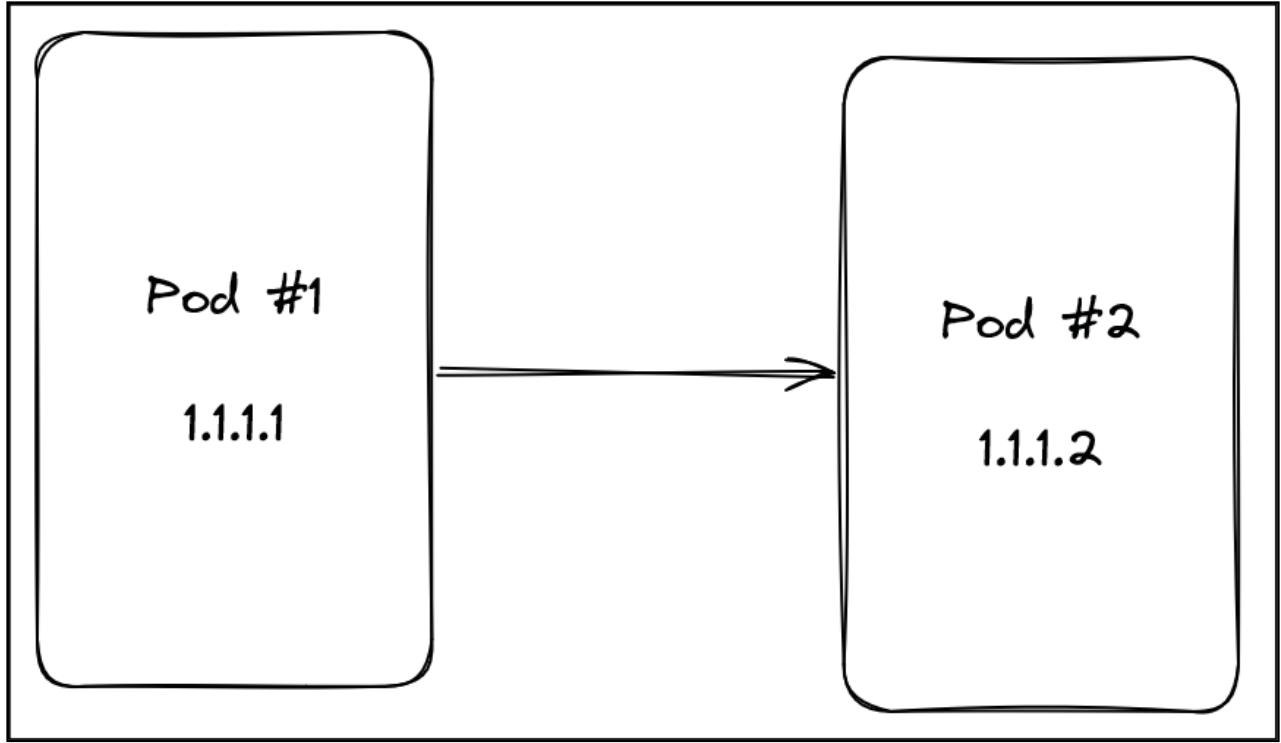


Figura 7.5 Esempio di Pod con assegnati due indirizzi IP.

Quando il secondo Pod viene riavviato, questo potrebbe avere indirizzo IP diverso, e questa connessione verrebbe a mancare e i due non sarebbero più in grado di comunicare. Che cosa succederebbe se il secondo Pod fosse replicato, e il primo dovesse raggiungerlo? Quale indirizzo dovrebbe utilizzare?

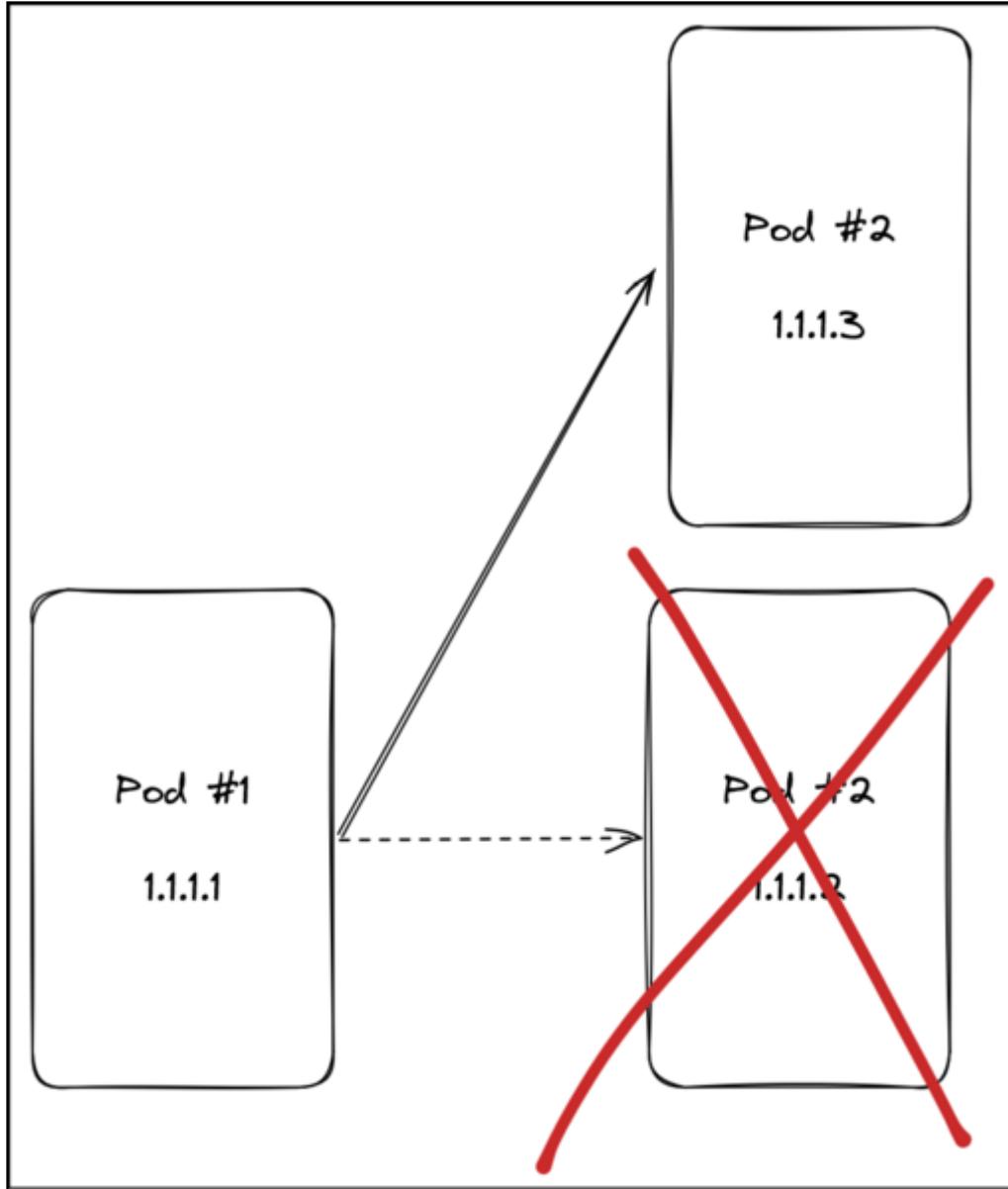


Figura 7.6 Scenario in cui il Pod #2 viene riavviato e il suo indirizzo IP cambia.

Ed è qui che entra in gioco il Service: questo oggetto lavora come uno strato aggiuntivo che è in grado di “gestire” il traffico in ingresso e in uscita e che è indipendente dal ciclo di vita del Pod. Genericamente parlando, in questo modo il primo Pod potrà raggiungere il secondo facendo riferimento al solo Service, senza doversi preoccupare del fatto che ci siano più repliche dello stesso Pod o che questo venga riavviato modificando il suo indirizzo IP: il primo Pod potrà sempre connettersi in tutta sicurezza all’indirizzo e alla porta del Service ed essere reindirizzato verso il Pod giusto. Piccola nota: i Service puntano ai Pod, non a Deployment o ReplicaSet; usano direttamente i Pod grazie alle label, con cui riescono a selezionare quelli corretti. Per quanto questo possa sembrare banale, in realtà questa modalità offre una grande flessibilità: non importa attraverso quali vari (forse anche personalizzati) modi in cui sono stati creati i Pod, il Service funzionerà lo stesso.

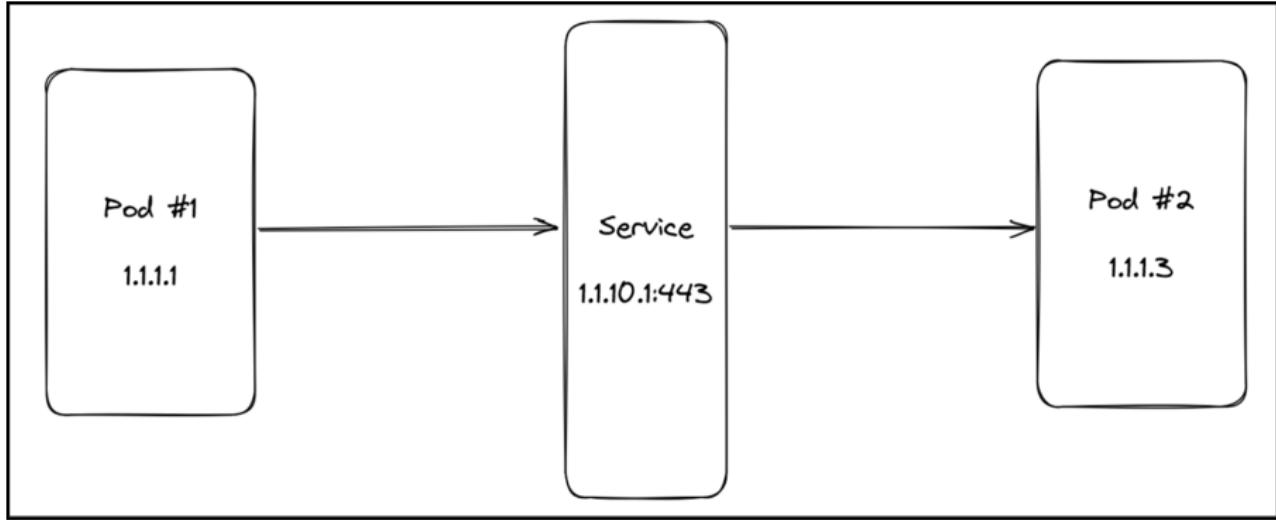


Figura 7.7 Rappresentazione di come funziona un Service per la comunicazione tra due Pod.

ClusterIP

Questo è il funzionamento al minimo di un Service, ma abbiamo visto che ne esistono più tipologie e quindi partiamo da quella più semplice, ossia il ClusterIP. Quanto descritto finora corrisponde proprio a questa tipologia, e permette a tutti i Pod interni al cluster (non solo del namespace dove il Pod si trova, come vedremo) di raggiungere qualsiasi applicazione specificando alcune semplici informazioni: si tratta del metodo perfetto per permettere la comunicazione tra Pod.

Vediamo un esempio di definizione di un Service, e analizziamo in che modo questo renda raggiungibile la nostra applicazione: all'interno del file YAML, definiamo un nome e un elenco di porte che vogliamo rendere accessibili dall'applicazione, e che corrispondono a quelle esposte. Nel caso di esempio, la porta (campo `port`) 3000 del Service sarà in ascolto per tutte le richieste che gli arriveranno e che girerà alla porta 443 (campo `targetPort`) utilizzando il protocollo TCP.

Listato 7.1 Definizione di un Service di tipo ClusterIP

```

apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
  - port: 3000
    protocol: TCP
    targetPort: 443
  selector:
    app: my-app
  type: ClusterIP

```

Soffermiamoci un secondo sul campo `targetPort`: il nome può essere fuorviante ed è bene chiarire adesso il suo funzionamento. Questa proprietà definisce la porta sul Pod a cui viene inviata la richiesta, ossia quella esposta dal container, al contrario di `port`, che espone una porta per gli altri Pod del cluster. Un Service può mappare qualsiasi porta in entrata su una `targetPort`, anche se, per impostazione predefinita e per comodità, spesso `targetPort` è impostato sullo stesso valore del campo `port`. Il Service ClusterIP distribuisce le richieste in base a un approccio casuale o *round-robin*, e lo fa rendendo disponibili i Pod all'interno del cluster tramite un nome e un IP.

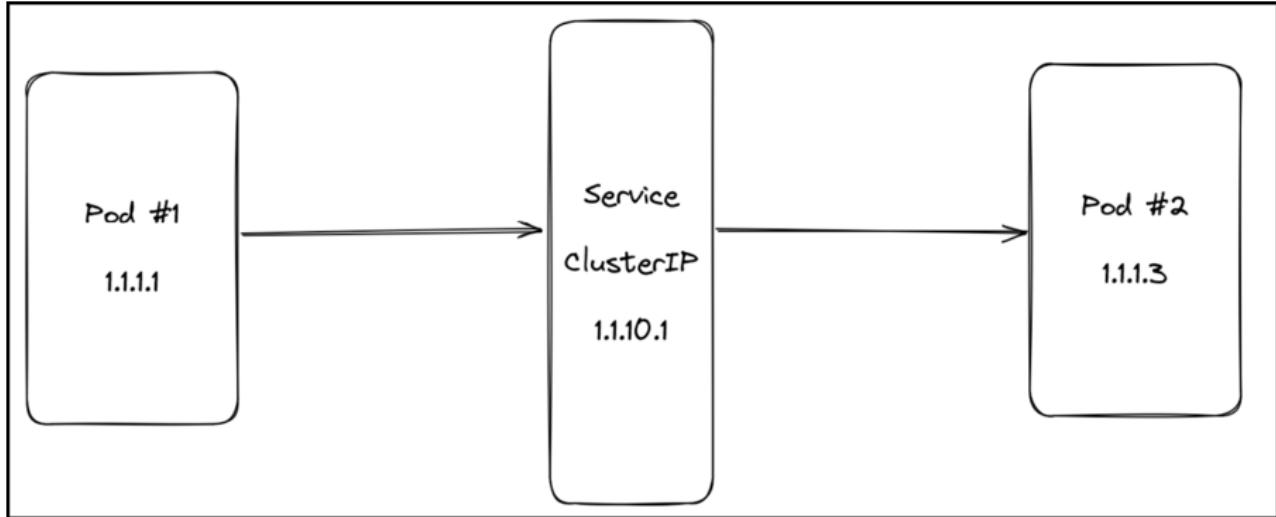


Figura 7.8 Rappresentazione di un Service di tipo ClusterIP

Alla creazione di questa risorsa, verranno aggiunte delle informazioni all'interno della definizione del Service, come l'indirizzo IP associato al cluster: visualizzarlo con il comando `kubectl get svc`:

Listato 7.2 Service con campo clusterIP valorizzato

```

apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  clusterIP: 10.100.71.132
  clusterIPs:
  - 10.100.71.132
  internalTrafficPolicy: Cluster
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - port: 3000
  protocol: TCP
    targetPort: 443
  selector:
    app: my-app
  sessionAffinity: None
  type: ClusterIP

```

Quando abbiamo parlato in precedenza delle tipologie di comunicazione che possiamo avere all'interno del cluster, gli scenari che prevedevano la comunicazione tra due Pod nello stesso nodo o su nodi diversi prevedono l'utilizzo di questo Service: questo perché, alla sua creazione, gli verrà associato un indirizzo IP valido all'interno del cluster, e il nome definito nello YAML potrà essere utilizzato per raggiungerlo in qualunque parte del cluster. Vediamo qualche esempio: se creiamo due Service come i seguenti (uno applicativo e uno per Postgres), questi saranno in grado di essere raggiunti tramite il *nome* scelto e la relativa porta. Per fare un test, utilizziamo il comando `curl` disponibile all'interno del Pod, e proviamo a contattare tramite `telnet` il database.

Listato 7.3 Definizione di un Service per l'applicazione

```

kind: Service
apiVersion: v1
metadata:

```

```

name: sample-webapp
labels:
    app: sample-webapp
spec:
  ports:
  - name: 8080-tcp
    protocol: TCP
    port: 8080
    targetPort: 8080
  type: ClusterIP
  selector:
    app: sample-webapp
  deploymentconfig: sample-webapp

```

Listato 7.4 Definizione di un Service per Postgres

```

kind: Service
apiVersion: v1
metadata:
  name: postgresql
spec:
  ports:
  - name: postgresql
    protocol: TCP
    port: 5432
    targetPort: 5432
  type: ClusterIP

```

Listato 7.5 Test di raggiungibilità verso Postgres

```

curl -vvvk telnet://postgresql:5432
>>>
* Rebuilt URL to: telnet://postgresql:5432/
*   Trying 10.30.192.228...
* TCP_NODELAY set
* Connected to postgresql (10.30.192.228) port 5432 (#0)

```

Listato 7.6 Test di raggiungibilità verso Postgres

```

curl -vvvk telnet://sample-webapp:8080
>>>
* Rebuilt URL to: telnet://sample-webapp:8080/
*   Trying 10.30.210.207...
* TCP_NODELAY set
* Connected to sample-webapp (10.30.210.207) port 8080 (#0)

```

Negli esempi riportati, abbiamo chiamato il Service semplicemente facendo riferimento al nome e alla porta, nulla di più: nell'output, ci è stato riportato che la connessione è avvenuta con successo e che siamo stati collegati al Pod specificato con l'indirizzo IP. E se volessimo raggiungere lo stesso Pod di Postgres, ma da un altro namespace? Possiamo utilizzare la seguente notazione, che specifica l'ambiente in cui il Pod è in esecuzione, più eventualmente l'FQDN del cluster:

Listato 7.7 Test di raggiungibilità verso Postgres da un namespace esterno

```

curl -vvvk telnet://postgresql.my-namespace.svc.cluster.local:5432
>>>
* Rebuilt URL to: telnet://postgresql.my-namespace.svc.cluster.local:5432/
*   Trying 10.30.192.228...
* TCP_NODELAY set
* Connected to postgresql.my-namespace.svc.cluster.local (10.30.192.228) port 5432 (#0)

# oppure senza .svc.cluster.local, che assume il valore di default
curl -vvvk telnet://postgresql.my-namespace:5432
>>>
...

```

Qui entra in gioco un componente che finora abbiamo ignorato: il DNS. In effetti, come fa Kubernetes a lavorare e risolvere i nomi dei Service rispetto agli indirizzi IP? Il DNS, genericamente parlando, è uno strumento che ci permette di salvare da qualche parte le informazioni per associare dei nomi di dominio con i relativi indirizzi IP. I cluster Kubernetes dispongono di un servizio responsabile della risoluzione DNS, motivo per cui a ogni Service in un cluster viene assegnato un nome di dominio il cui formato è quello che abbiamo visto nell'esempio. Pertanto, quando viene effettuata una richiesta a un Service tramite il suo nome di dominio, il servizio DNS la risolve verso l'indirizzo IP del Service di destinazione, ed è proprio `kube-proxy` a convertire l'indirizzo IP di quel Service in quello corrispondente al Pod che vogliamo raggiungere.

NodePort

Andiamo avanti con gli altri scenari: vogliamo raggiungere l'applicazione dall'esterno del cluster. In questo caso, modifichiamo il Service creato in precedenza, con alcune modifiche: introduciamo il tipo `NodePort` e assegnamo una porta del nodo al Service, come visibile di seguito.

Listato 7.8 Definizione di un Service di tipo NodePort

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
    - port: 3000
      protocol: TCP
      targetPort: 443
      nodePort: 30080
  selector:
    app: my-app
  type: NodePort
```

Questo vuol dire che, grazie a questo Service, l'applicazione sarà raggiungibile dall'esterno, tramite la porta 30080. Quando all'inizio del capitolo abbiamo parlato delle interfacce di rete e di come i Pod possano comunicare tra loro e verso l'esterno, abbiamo infatti ipotizzato lo scenario in cui ci fosse bisogno di raggiungere dei Pod dall'esterno del cluster: non ci basta più quindi far sì che il Service attivi un indirizzo IP valido al di fuori (in questo caso) del cluster, ma è necessario che si esponga anche una specifica porta. Questo vuol dire che, se possiamo raggiungere il nodo attraverso il suo indirizzo IP o hostname, potremo allora raggiungere l'applicazione.

Questo tipo di Service è quindi l'ideale quando vogliamo esporre un Pod verso l'esterno del cluster, tenendo però a mente alcuni aspetti: in primis, le porte che possiamo esporre per i nodi hanno un range ben preciso, che va da 30000 fino a 32767, e questo perché l'intervallo scelto vuole evitare conflitti con qualsiasi altra risorsa che sia presente sulla rete dei nodi che ospitano Kubernetes, visto che, tra le opzioni, abbiamo la possibilità di non specificare una porta per il Service e lasciare che questa venga assegnata dinamicamente. Per esempio, se avessimo l'intervallo 1-32767, il campo `nodePort` potrebbe essere in conflitto con la porta 22 dell'host, utilizzata per le comunicazioni tramite SSH. Di base, quindi, non si vuole che le porte di questa tipologia di Service vadano a sovrascrivere le porte reali utilizzate dal nodo. In ogni caso, è bene dire che questo range può essere eventualmente allargato oltre la porta 32767, ma non può mai essere ridotto.

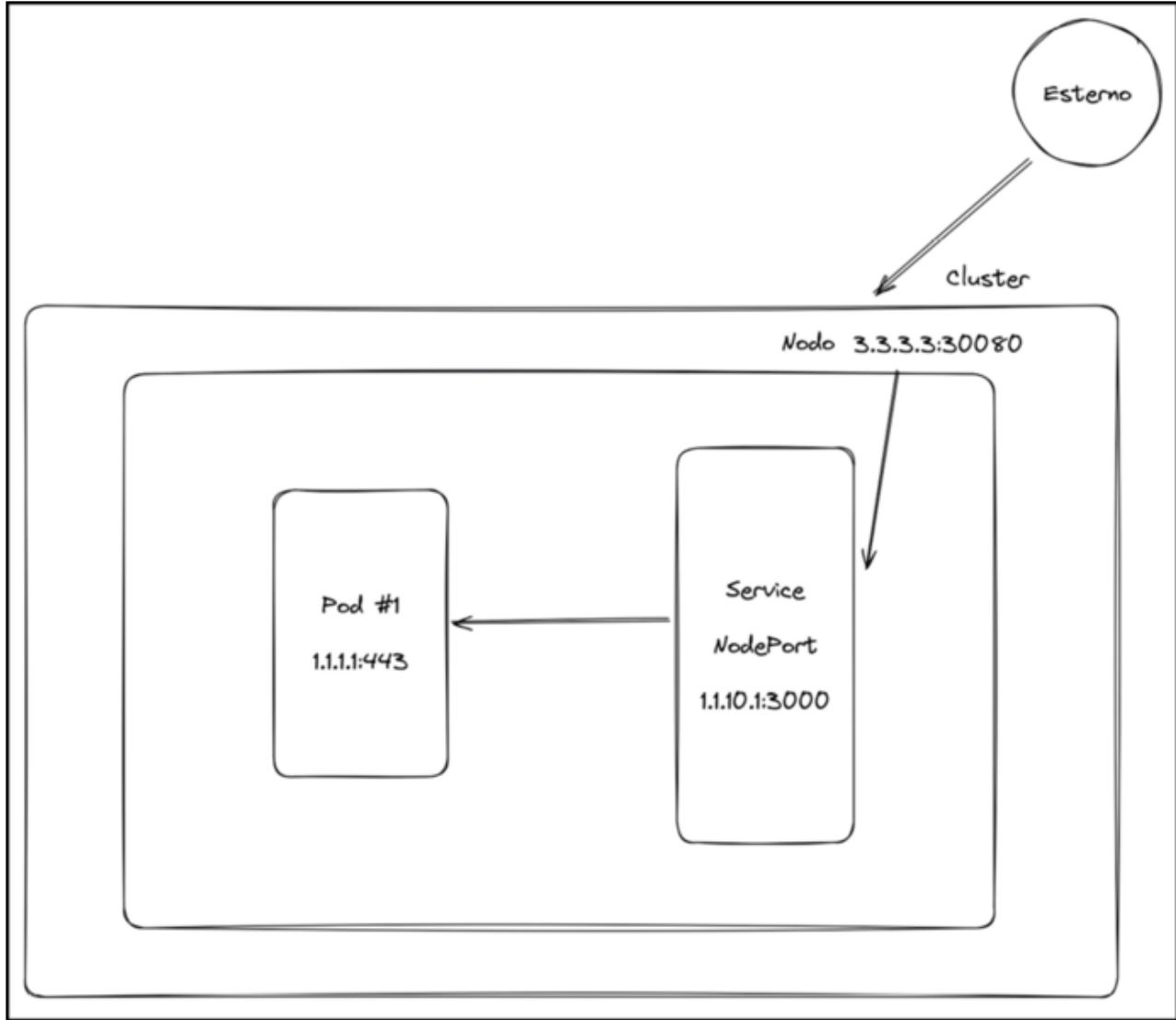


Figura 7.9 Rappresentazione di un Service di tipo NodePort.

LoadBalancer

Oltre al NodePort e al ClusterIP, esiste il *LoadBalancer*: un Service di questo tipo è il modo standard per esporre un servizio su Internet. Utilizziamo un Service LoadBalancer se desideriamo avere un singolo IP che distribuisce le richieste (utilizzando un metodo come *round-robin*) a tutti i nostri IP di nodi esterni. Per questo all'inizio abbiamo detto che il LoadBalancer è costruito sopra i Service di tipo NodePort; quindi, lo YAML modificato per LoadBalancer rispetto all'esempio di prima è semplicemente questo:

Listato 7.9 Definizione di un Service di tipo LoadBalancer

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
  - port: 3000
    protocol: TCP
```

```

targetPort: 443
nodePort: 30080
selector:
  app: my-app
type: LoadBalancer

```

Tutto ciò che fa un servizio LoadBalancer è creare un servizio NodePort e inviare un messaggio al provider che ospita il cluster Kubernetes chiedendo la configurazione di un load balancer che punti a tutti gli IP dei nodi esterni e a una specifica nodePort. Se il provider non supporta questo messaggio o una forma di automatismo per la creazione di questo oggetto, allora non succede nulla e il LoadBalancer diviene uguale a un servizio NodePort; questa risorsa è infatti molto utile quando lavoriamo su infrastrutture on cloud, dove entità di questo tipo sono facilmente configurabili.

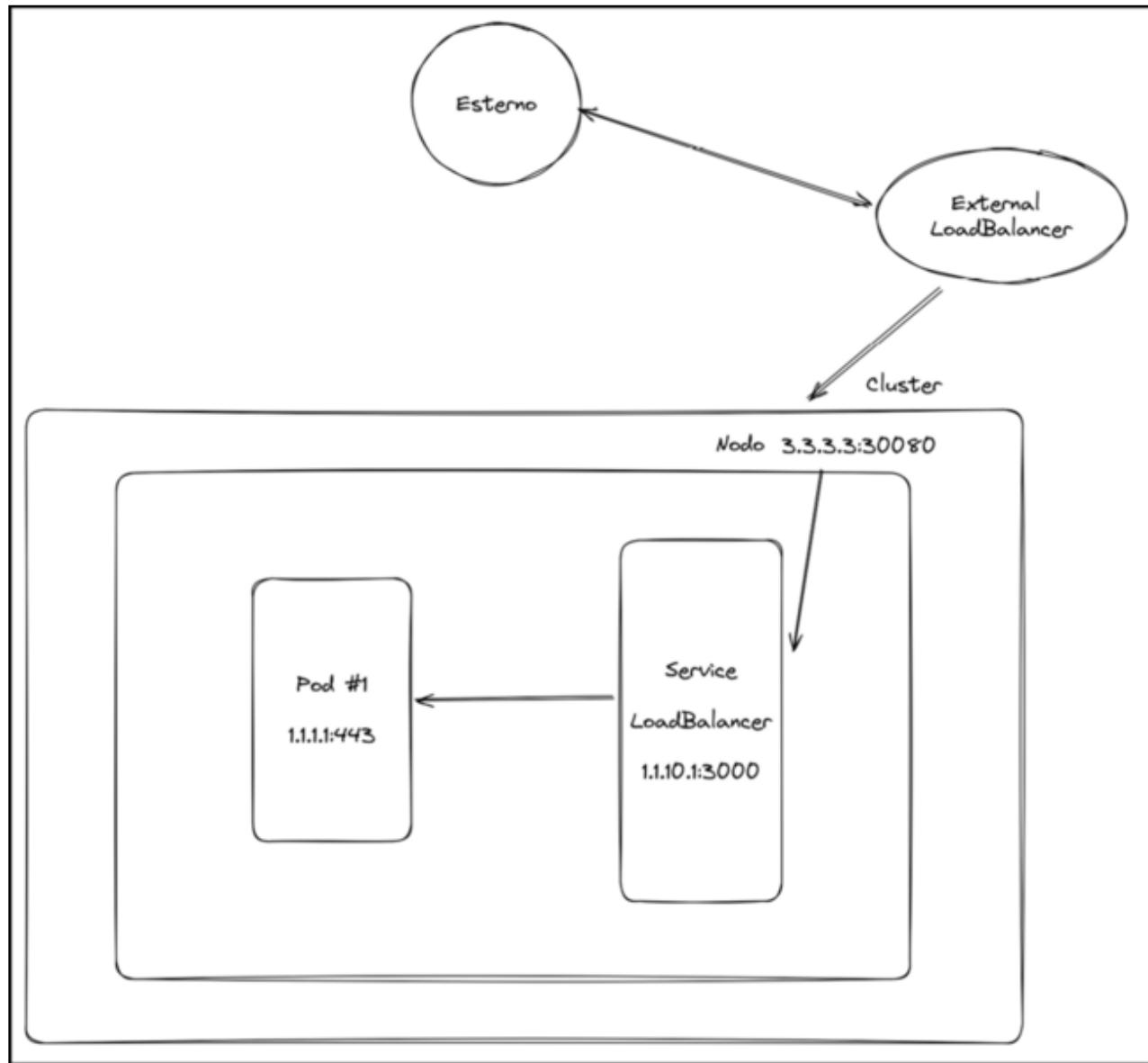


Figura 7.10 Rappresentazione di un Service di tipo LoadBalancer.

Un Service di questo tipo ha ancora la porta 30080 aperta per gli IP interni ed esterni dei nodi, come prima nel caso del NodePort, e funziona ancora come un Service ClusterIP, ma è raggiungibile dall'esterno e riesce a gestire il traffico tramite una policy definita. Se vuoi esporre direttamente verso l'esterno un Service, questo è il metodo predefinito: tutto il traffico sulla porta specificata verrà inoltrato

al Service. Non ci sono filtri, nessun routing, e ciò significa che puoi inviare quasi qualsiasi tipo di traffico a esso, come HTTP, TCP, UDP, Websockets o altro. Il grande svantaggio è che ogni Service che esponi con un LoadBalancer otterrà il proprio indirizzo IP e, a seconda dei cloud provider, dovrà pagare per un LoadBalancer per servizio esposto, il che può diventare costoso.

ExternalName

L'ultima tipologia di Service è l'*ExternalName*: questo potrebbe essere considerato un oggetto separato rispetto agli altri tre, perché serve a creare un Service interno con un endpoint che punta a un nome DNS. In altre parole, 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 essere raggiunto tramite un nome specifico, come `my-application.my-cluster.com`: il Service che creeremo, quando verrà richiamato, risolverà il nome del Pod con quanto specificato. Per questo motivo, non gli verrà assegnato alcun indirizzo IP come visto per il ClusterIP, ma sarà il DNS del cluster a ritornare quello che, in gergo tecnico, si chiama CNAME: questo serve a mappare un nome alternativo rispetto a un nome di dominio vero o canonico.

Listato 7.10 Definizione di un Service di tipo ExternalName

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: ExternalName
  externalName: my-application.my-cluster.com
```

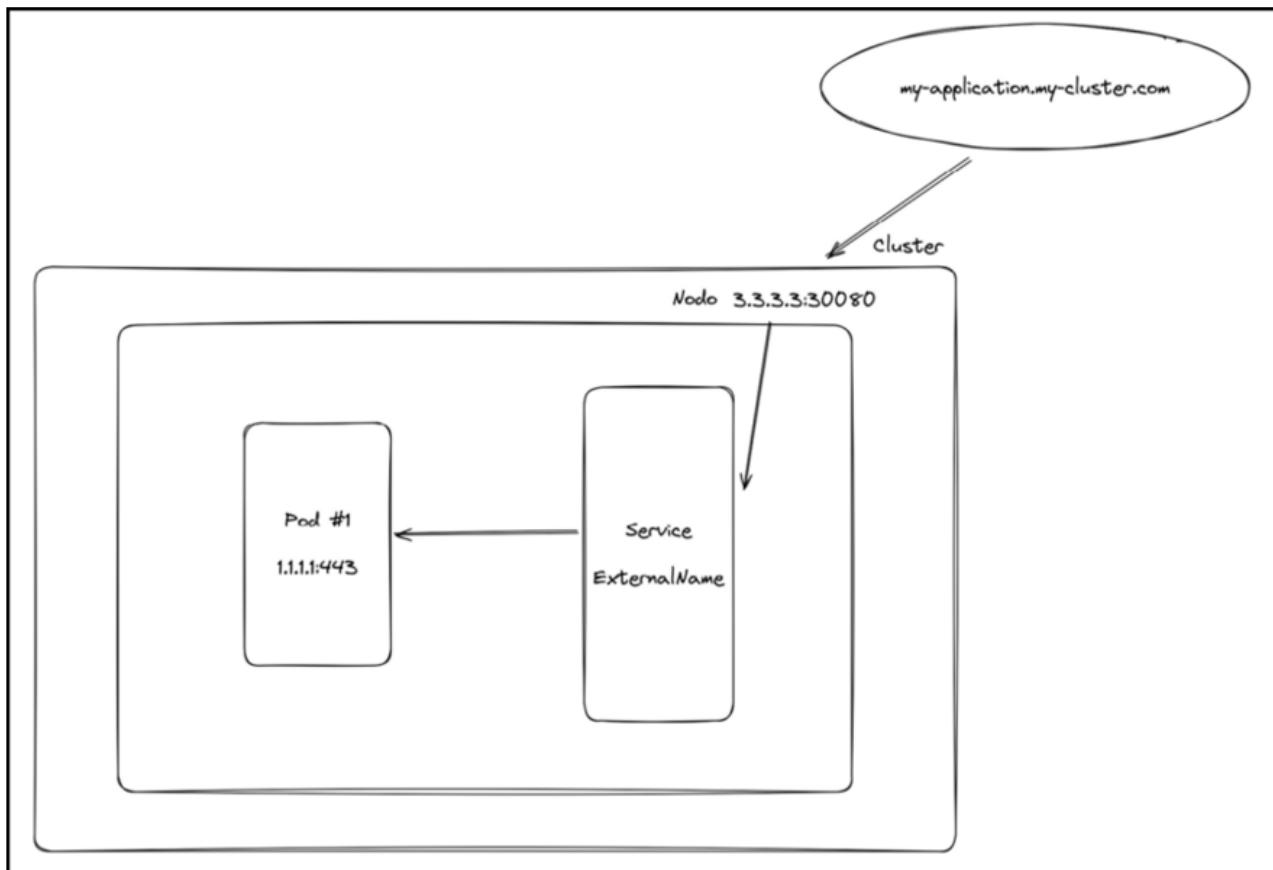


Figura 7.11 Rappresentazione di un Service di tipo ExternalName.

Questo tipo di Service funge da proxy, consentendo a un utente di reindirizzare le richieste a un'applicazione che risiede all'esterno (o all'interno) del cluster, e che deve avere un record CNAME per collegare il nome DNS a un nome locale del cluster: in questo modo i Pod potranno sfruttare il nome di quel Service come alias per raggiungere l'hostname specificato nel campo `externalName`!

Ingress

A differenza di tutti gli esempi precedenti, l'Ingress non è in realtà un tipo di Service. Invece, si trova di fronte a più servizi e funge da "router intelligente" o punto di ingresso nel tuo cluster. Secondo la documentazione ufficiale di Kubernetes un "Ingress espone le route HTTP e HTTPS dall'esterno del cluster ai servizi all'interno del cluster. Il routing del traffico è controllato dalle regole definite nella risorsa Ingress". In altre parole, un Ingress funge da qualche forma di collegamento per l'instradamento del traffico controllato tra i servizi distribuiti in un cluster Kubernetes e utenti o client esterni e lavora grazie a un Load Balancer che ne gestisce il traffico. L'Ingress è probabilmente il modo più potente per esporre i tuoi Service, ma può anche essere il più complicato: esistono molti tipi di controller Ingress, da Google Cloud Load Balancer, Nginx, Contour, Istio e altri. Esistono anche diversi plugin per i controller Ingress, per cui è fondamentale valutarne lo scopo: l'Ingress è estremamente utile se desideri esporre più Services con lo stesso indirizzo IP e questi servizi utilizzano tutti lo stesso protocollo, in genere HTTP. In questo modo, è possibile creare un solo bilanciatore e quindi, nel caso del cloud, non moltiplicare i costi.

Ma andiamo con ordine: il significato letterale di Ingress si riferisce all'atto di "entrare", ed è lo stesso anche nel mondo Kubernetes. "Ingress" indica il traffico che entra nel cluster e che ottiene una risposta in uscita dal cluster. Tramite questa risorsa, è possibile mantenere le configurazioni di routing DNS, grazie a un controller che ne esegue l'instradamento effettivo, leggendo le regole di routing grazie a quanto memorizzato in etcd. Senza l'Ingress Kubernetes, per esporre un'applicazione al mondo esterno, dovresti aggiungere un LoadBalancer ai Pod, quando questo lavora invece tramite un controller che costituisce un livello di proxy inverso tra il bilanciatore del carico e l'endpoint del servizio Kubernetes.

Abbiamo quindi nominato una risorsa "accessoria" accostata all'Ingress: un Ingress Controller. Per capire come funziona questo tipo di oggetto, dobbiamo infatti fermarci a capire qual è la differenza tra questi due: un Ingress è responsabile della memorizzazione delle regole di routing DNS nel cluster, mentre gli Ingress Controller (Nginx/HAProxy ecc.) sono responsabili dell'instradamento accedendo alle regole DNS applicate tramite l'Ingress. Un Ingress è una risorsa Kubernetes nativa in cui specifichi le regole di routing DNS: ciò significa che esegui il mapping del traffico DNS esterno agli endpoint del servizio Kubernetes interni e quindi richiede un controller che possa gestire il routing delle regole specificate nell'oggetto stesso.

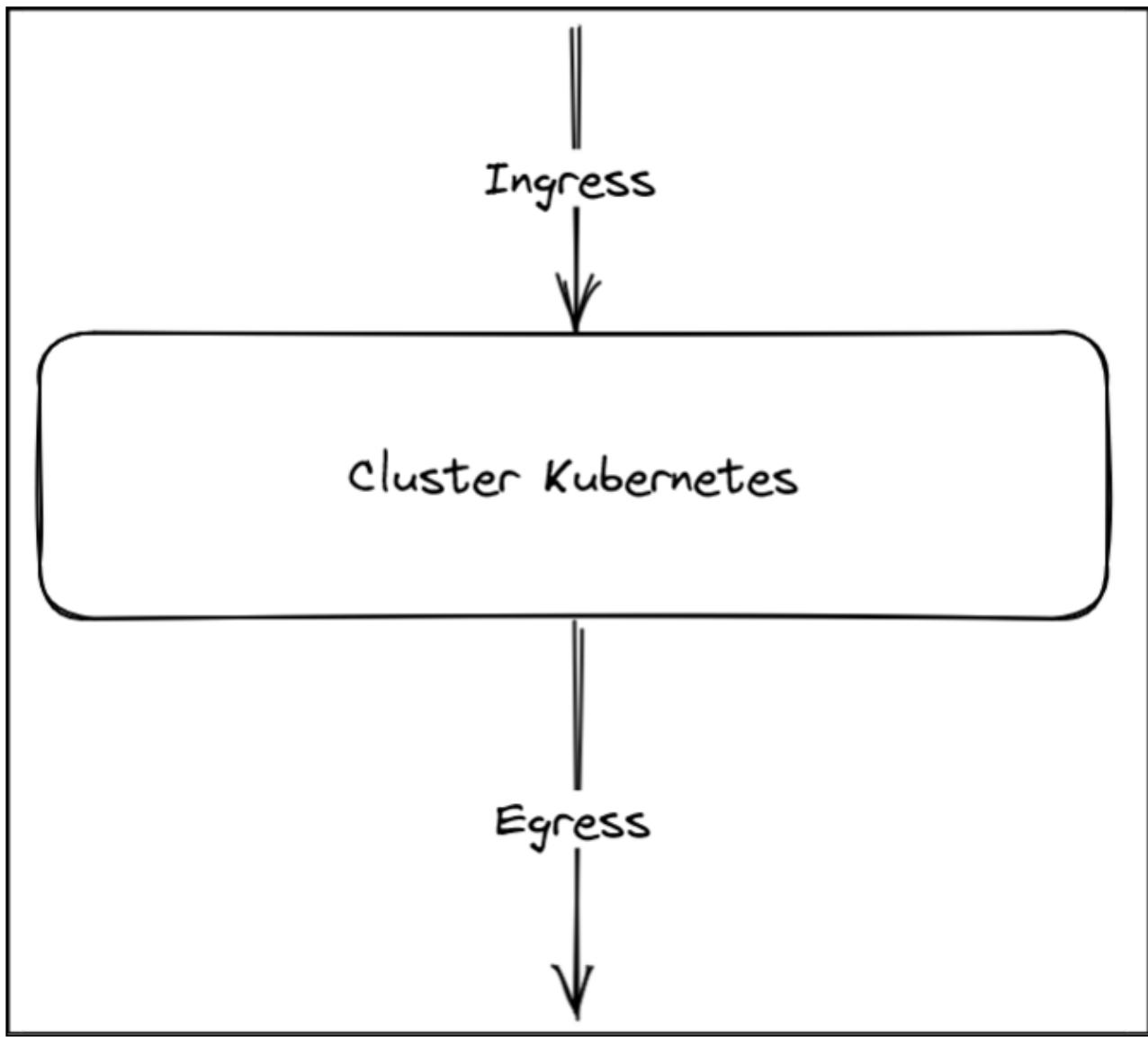


Figura 7.12 Funzionamento ideale di un Ingress.

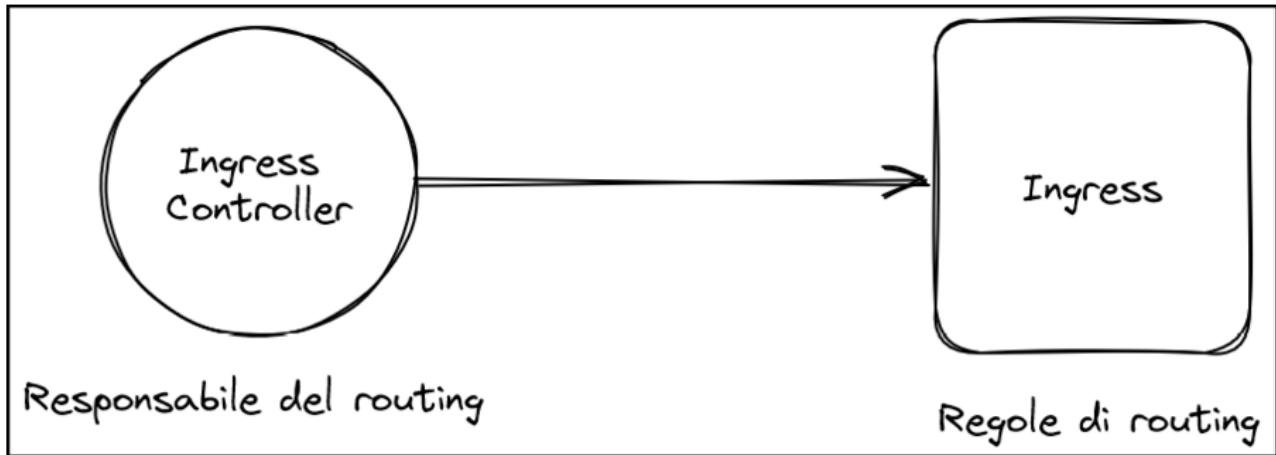


Figura 7.13 Componenti che entrano in azione per rendere funzionante la comunicazione dall'esterno del cluster tramite un Ingress.

Un esempio di Ingress è il seguente: tutte le chiamate verso `test.apps.com` dovrebbero raggiungere il Service denominato `my-service` sulla porta 80 che risiede nel namespace `dev`. Come puoi vedere, tutto ciò che viene specificato ha molto a che fare con le regole di instradamento: nel campo `http` è possibile aggiungere più endpoint di routing per instradare la request, come anche eventuali configurazioni TLS, e via dicendo.

Listato 7.11 Esempio di definizione di un Ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
  namespace: dev
spec:
  rules:
  - host: test.apps.com
    http:
      paths:
      - backend:
          serviceName: my-service
          servicePort: 80
```

Gli Ingress Controller non costituiscono un'implementazione nativa di Kubernetes: ciò significa che non rappresentano un oggetto predefinito nel cluster e dovremo configuralo affinché le regole degli Ingress funzionino. Sono disponibili diversi controller open source e in versione enterprise, e questo corrisponde in genere a un'implementazione del *reverse web server* che funge da proxy nel cluster. Nell'ambito di Kubernetes, si tratta di un oggetto avviato all'interno del cluster che funge da *reverse proxy* che lavora a stretto contatto con un Service di tipo LoadBalancer. Essendo un'applicazione interna al cluster, è possibile configurare più controller mappati a più risorse di bilanciamento del carico; in questo caso, ogni Ingress Controller dovrebbe avere un identificatore chiamato *ingress-class* aggiunto tra le annotazioni della risorsa. Nginx è un esempio abbastanza comune di Ingress Controller per Kubernetes: all'interno della configurazione di questo server è possibile specificare come comunicare con le API di Kubernetes per gestire il traffico e le relative regole da applicare.

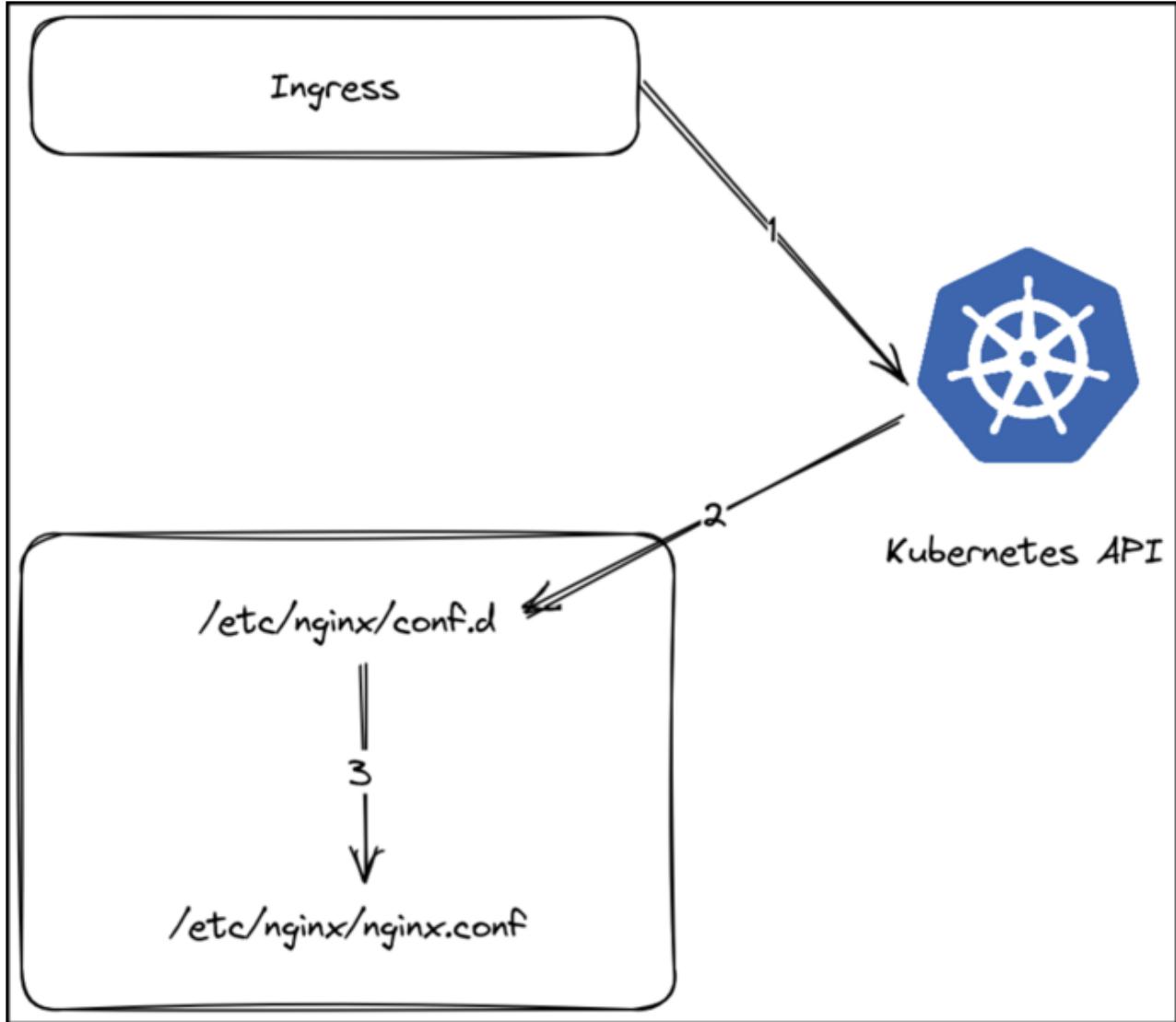


Figura 7.14 Come viene utilizzato l’Ingress per aggiornare le impostazioni del proxy.

Quel che succede è questo: il Pod che ospita Nginx ha un file di configurazione che serve a mantenere le informazioni circa il routing del traffico, e che le API di Kubernetes devono poter leggere. Per ogni risorsa di tipo Ingress, viene creata una configurazione all’interno della cartella `/etc/nginx/conf.d`, mentre il file `/etc/nginx/nginx.conf` manterrà tutte le configurazioni che provengono da questa cartella. Questo vuol dire che se aggiorni un Ingress con nuove configurazioni, la configurazione di Nginx viene nuovamente aggiornata. Se ci colleghiamo al Pod di Nginx usando `kubectl exec` e controlliamo il file `/etc/nginx/nginx.conf`, vedremo tutte le regole specificate dagli Ingress applicate nel file di configurazione.

Quanto descritto finora riguarda solo Nginx in veste di Ingress controller, ma la realtà è che esistono moltissime alternative: HAProxy è un’altra soluzione ampiamente adottata, così come, in ambito i corrispettivi oggetti presenti per ogni provider: GKE per Google e AWS ALB per AKS; ognuno di essi ha una sua installazione con relativa implementazione, che non può essere trattata in poche righe. Nella documentazione ufficiale di ognuno di essi ci sono le istruzioni dettagliate su come questi debbano essere installati e configurati, documentazione che si lascia come compito di lettura a casa.

Che cosa abbiamo imparato

- Come funziona la comunicazione tra i diversi oggetti presenti nel cluster Kubernetes, e che tipologie di connessione ci aspettiamo avvengano.
- Quali Service esistono tra le risorse Kubernetes e il significato di ognuna di esse con la relativa applicabilità.
- Come utilizzare un Ingress per esporre all'esterno, tramite HTTP/HTTPS, un'applicazione installata e avviata all'interno del cluster, e come funziona grazie all'esistenza di un Ingress Controller.

Storage

Penso che una delle grandi sfide sia in realtà coltivare le menti dei principianti per assicurarsi che tu sia ancora aperto al mondo per vedere cose nuove. Certo, potrai essere stanco. Potrai smettere di vedere cose nuove. Potrai iniziare a temere il fallimento. Ma queste sono le cose di cui un imprenditore ha bisogno: una mente aperta e la capacità di vedere il mondo con occhi nuovi.

– Caterina Fake, co-fondatrice di Flickr e Hunch

In questo capitolo analizzeremo più nel dettaglio che cosa significa poter “persistere” dei dati all’interno di un cluster Kubernetes: in effetti, abbiamo sempre parlato dei Pod come degli oggetti transienti, e quindi soggetti a riavvii con conseguenti perdite di dati o informazioni. In realtà, è ovviamente possibile salvare queste informazioni tramite degli oggetti che si occupino direttamente della loro gestione secondo diverse modalità e specifiche. Nelle seguenti sezioni parliamo quindi di che cosa rappresenti il concetto di “volume” in Kubernetes, e più in generale per un orchestratore o uno strumento di containerizzazione, e quali modalità abbiamo a disposizione per poter persistere i nostri dati attraverso queste risorse, facendo qualche esempio pratico di associazione di un volume a un Pod o a un controller.

Volumi

Nei capitoli precedenti, abbiamo introdotto i Pod e altre risorse Kubernetes che interagiscono con essi, vale a dire *ReplicationController*, *ReplicaSet* e *Deployment*. Ora facciamo un passo indietro e torniamo all’interno del Pod per scoprire come i suoi container possono accedere a un disco esterno e/o condividere l’archiviazione tra di loro. Abbiamo detto che i Pod sono simili agli host logici (come un qualsiasi laptop o server) in cui i processi in esecuzione al loro interno condividono risorse come CPU, RAM, interfacce di rete e altro. Ci si aspetterebbe che i processi condividano anche i dischi, ma non è così: ogni container presente in un Pod ha il proprio filesystem isolato, perché il filesystem deriva dall’immagine del container. Ogni nuovo container viene avviato con l’esatto insieme di file che è stato aggiunto all’immagine al momento della fase di build del container. Inoltre, sappiamo che i Pod sono oggetti transienti e che quindi, se i container in un Pod vengono riavviati (o perché il processo principale ha avuto un errore, o perché i test di *readiness* o *liveness* hanno segnalato a Kubernetes che i container non erano più attivi), ti renderai conto che tutto ciò che avviene all’interno dei container, rimane nei container. Scherzi a parte, è chiaro che in alcuni scenari si desidera che un container che è appena stato riavviato continui da dove è terminato l’ultimo; in questo caso, potresti non aver bisogno (o volere) che l’intero filesystem sia salvato, ma magari vorrai preservare le directory che contengono dei dati aggiornati. Kubernetes fornisce questa funzionalità attraverso il concetto di *storage* o i volumi di archiviazione. Non sono risorse applicative come i Pod, ma sono invece definite come *parte* di un Pod e condividono il suo stesso ciclo di vita: ciò significa che un volume viene creato all’avvio del Pod e viene distrutto quando questo viene definitivamente eliminato. Per questo motivo, il contenuto di un volume persistrà anche tra diversi riavvii del container. Infatti, dopo che un container è stato riavviato, il nuovo container può vedere tutti i file che erano scritti all’interno del volume dal container precedente

e questo può anche eventualmente essere utilizzato da tutti contemporaneamente, se un Pod contiene più container.

I volumi Kubernetes sono un componente del Pod e sono quindi definiti nelle specifiche del Pod, proprio come i container. Non sono un oggetto Kubernetes autonomo e non possono essere creati o eliminati da soli. Un volume è disponibile per tutti i container nel Pod, ma deve essere “montato” in ogni container che deve accedervi; per ogni container, puoi montare il volume in qualsiasi posizione del suo filesystem. Ognuno di questi concetti è estremamente importante, quindi cerchiamo di dettagliarlo in maniera più pratica: riprendiamo un esempio dei capitoli precedenti, ma concentriamo le nostre energie nel comprendere come funzionano i volumi. In questo esempio, avevamo un Pod con due container: il primo contiene un container primario che espone una pagina HTML tramite Nginx e che ha come cartella principale la directory `/usr/share/nginx/html`, mentre il secondo ha come compito quello di creare un file `index.html` all'interno della stessa cartella principale del server Nginx.

Listato 8.1 Esempio di Pod con due container

```
apiVersion: v1
kind: Pod
metadata:
  name: two-containers
spec:
  restartPolicy: Never
  volumes:
  - name: shared-data
    emptyDir: {}
  containers:
  - name: nginx-container
    image: nginx
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html
  - name: sidecar-container
    image: debian
    volumeMounts:
    - name: shared-data
      mountPath: /my-data
    command: ["/bin/sh"]
    args: ["-c", "echo Hello from the sidecar container > /my-data/index.html"]
```

La parte più interessante è quella relativa al campo `volumes`: qui vengono specificati i volumi che il container avrà a disposizione durante il ciclo di vita del Pod e dove potrà memorizzare o leggere dei dati. Ora, prima di proseguire con il resto delle spiegazioni, proviamo a introdurre questa semplificazione: la parola “volume” potrebbe far pensare a un disco vero e proprio che in qualche modo viene agganciato al Pod; come vedremo, questo è più vicino a una specie di cartella che andiamo a persistere da qualche parte (anche un disco, perché no), e il suo confine risiede proprio in quella cartella. Uno dei problemi principali quando si parla di volumi è proprio quello di immaginarli come un filesystem che ha una struttura aderente a una qualsiasi di un sistema operativo conosciuto; in questo caso, possiamo invece assimilare questo concetto più a una singola cartella che funge da archivio storico di ciò che vogliamo persistere dopo che il Pod sarà stato riavviato. Riprendendo l'esempio precedente, il primo punto di attenzione va sulla dichiarazione dei volumi: nel campo `volumes` riportiamo qual è il filesystem da utilizzare per i container che verranno. La configurazione prevede che si possa riportare uno o più volumi che si intende utilizzare all'interno dei diversi container, e infatti il campo prevede un array di elementi, dove il campo `name` rappresenta il nome da associargli. DoPodiché, abbiamo la definizione dei `volumeMounts`, che invece permettono di specificare la posizione del volume che deve essere montato: in altre parole, qual è la cartella che quel volume rappresenta,

all'interno della quale potremo andare a leggere o scrivere i nostri file e directory. In questo caso è obbligatorio utilizzare lo stesso nome di riferimento del volume specificato nel campo `volumes`, così come il `mountPath`, ovvero la cartella di cui sopra. I campi `volume` e `volumeMounts` vanno di pari passo: non è possibile creare un volume senza montarlo o montare un volume che non è stato creato: un piccolo esempio di quanto descritto finora è visibile nella Figura 8.1.

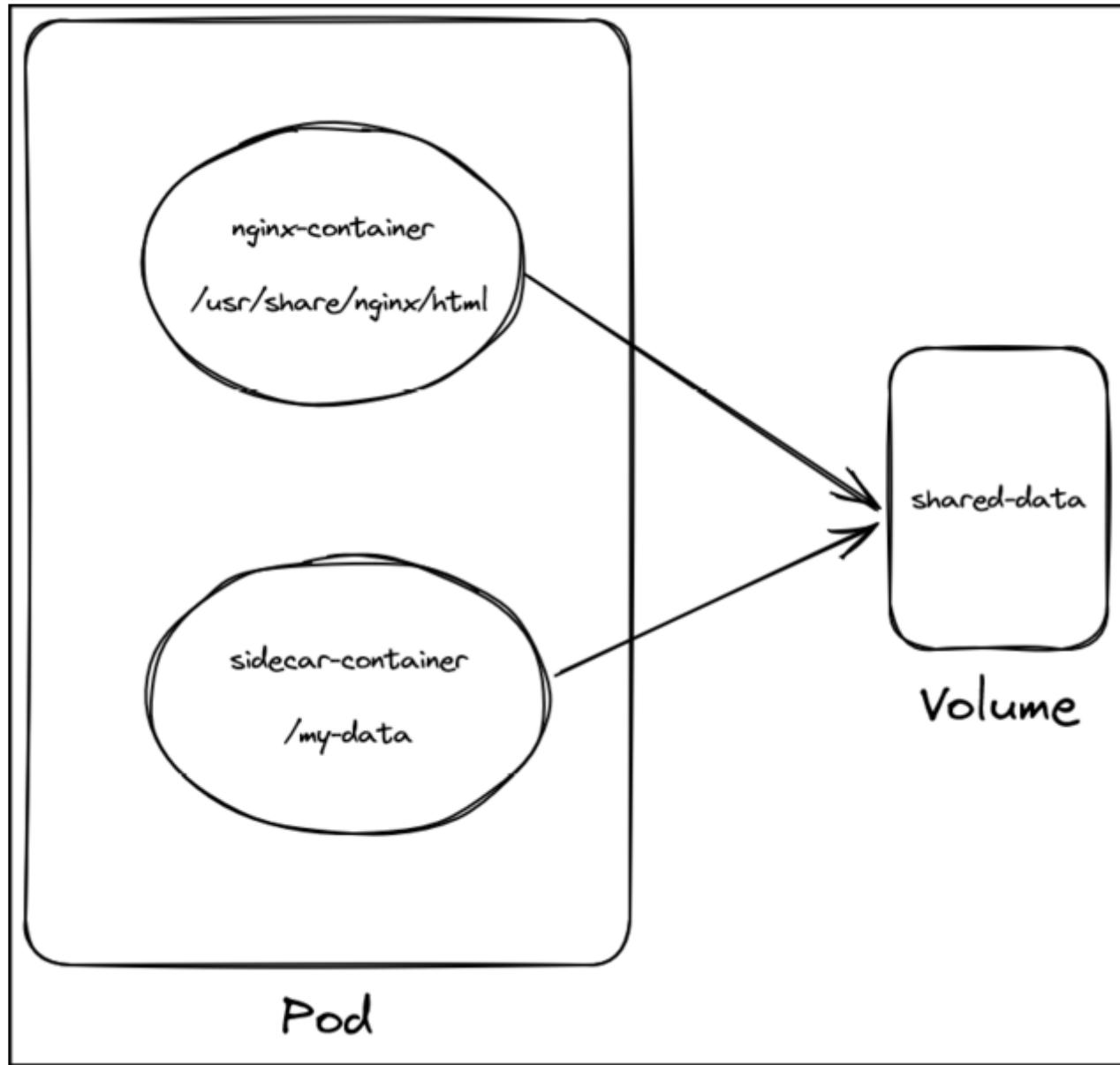


Figura 8.1 Rappresentazione del funzionamento di un volume che persiste le informazioni di un Pod.

Non abbiamo finito qui: nel campo `volumes` c'è un'altra proprietà, e si tratta della definizione di `emptyDir`. Questa rientra tra le diverse tipologie di volumi con cui potremo avere a che fare, quindi spostiamo l'occhio sulla prossima sezione.

Tipologie

Attualmente, ci sono diversi tipi di volume supportati da Kubernetes: alcuni richiedono una configurazione esterna come quelli legati a diversi cloud provider tra cui AWS, Azure e così via, il che

potrebbe comportare dei costi. Per il momento, li escluderemo dalla nostra pratica, ma ci concentreremo su altri tipi, come `emptyDir`, `hostPath`, `secret`, `configMap` e via dicendo. Partiamo con il dire che i tipi di volume sono classificati in due categorie principali.

- Temporaneo: questa è la categoria la cui durata dipende dal ciclo di vita del Pod e quindi persiste dopo il riavvio del container, ma non se il Pod viene cancellato. È una soluzione veloce, ma non persistente, quindi dovrebbe essere utilizzata per dati temporanei o applicazioni che non richiedono la persistenza dei dati. Una tipologia di volume a cui appartiene questa categoria è `emptyDir`.
- Durevole: si tratta di volumi che sopravvivono al ciclo di vita del Pod. La durata è indipendente dal ciclo di vita del Pod, e persiste tra i riavvii del container e del Pod oltre che alla loro cancellazione. I dati vengono conservati in questa categoria quando il Pod si arresta in modo anomalo o viene eliminato. I tipi di volume in questa categoria sono: `hostPath`, `persistentVolumeClaim` e quelli legati a diversi provider esterni.

emptyDir

Si tratta di un volume che viene creato per la prima volta quando un Pod viene assegnato a un nodo e la sua durata dipende dal ciclo di vita del Pod su quel nodo, per cui viene ricreato quando i container si arrestano in modo anomalo o si riavviano. Quando un Pod si interrompe, si arresta in modo anomalo o viene rimosso da un nodo, i dati nel volume `emptyDir` vengono eliminati e persi. Questo tipo di volume è adatto per l'archiviazione temporanea dei dati, come se fosse una cartella `temp`.

Un volume di tipo `emptyDir` viene generato creando prima un volume e in seguito specificando la sua tipologia come campo nel file del Pod nella sezione relativa, e poi può essere utilizzato; nell'esempio riportato in precedenza, abbiamo proprio utilizzato una cartella temporanea per condividere i dati tra i due container che, nel momento in cui il container verrà riavviato, non ci interessa perdere:

Listato 8.2 Esempio di Pod con un'emptyDir

```
apiVersion: v1
kind: Pod
metadata:
  name: two-containers
spec:
  restartPolicy: Never
  volumes:
  - name: shared-data
    emptyDir: {}
```

...

Per fare un test, provate a creare un Pod con questa tipologia di cartella come quello visto in precedenza: provate poi a creare un nuovo file all'interno della cartella dove `index.html` è memorizzato. Una volta che il Pod sarà cancellato e verrà riavviato, vedrete che questo file non esisterà più.

Listato 8.3 Test con emptyDir

```
# Creiamo il pod
kubectl create -f emptyDir.yaml

# Elenco dei pod
kubectl get pod
>>>
NAME          READY     STATUS    RESTARTS   AGE
two-containers 1/1      Running   0          15s
```

```
# Descrizione del pod
kubectl describe pod two-containers
>>>
Name:           two-containers
Namespace:     default
...
Environment:   <none>
Mounts:
  /my-data from my-volume (rw)
    /var/run/secrets/kubernetes.io/serviceaccount from default-token-z8p4x (ro)
Volumes:
  shared-data:
    Type:          EmptyDir (a temporary directory that shares a pod's lifetime)

# Entriamo nel pod tramite bash
kubectl exec -it two-containers -c nginx-container -- bin/bash

# Creiamo un file nel pod
echo I love Kubernetes > /usr/share/nginx/html/my-page.html

# Cancelliamo il pod
kubectl delete pod two-containers
>>>
pod "two-containers" deleted
```

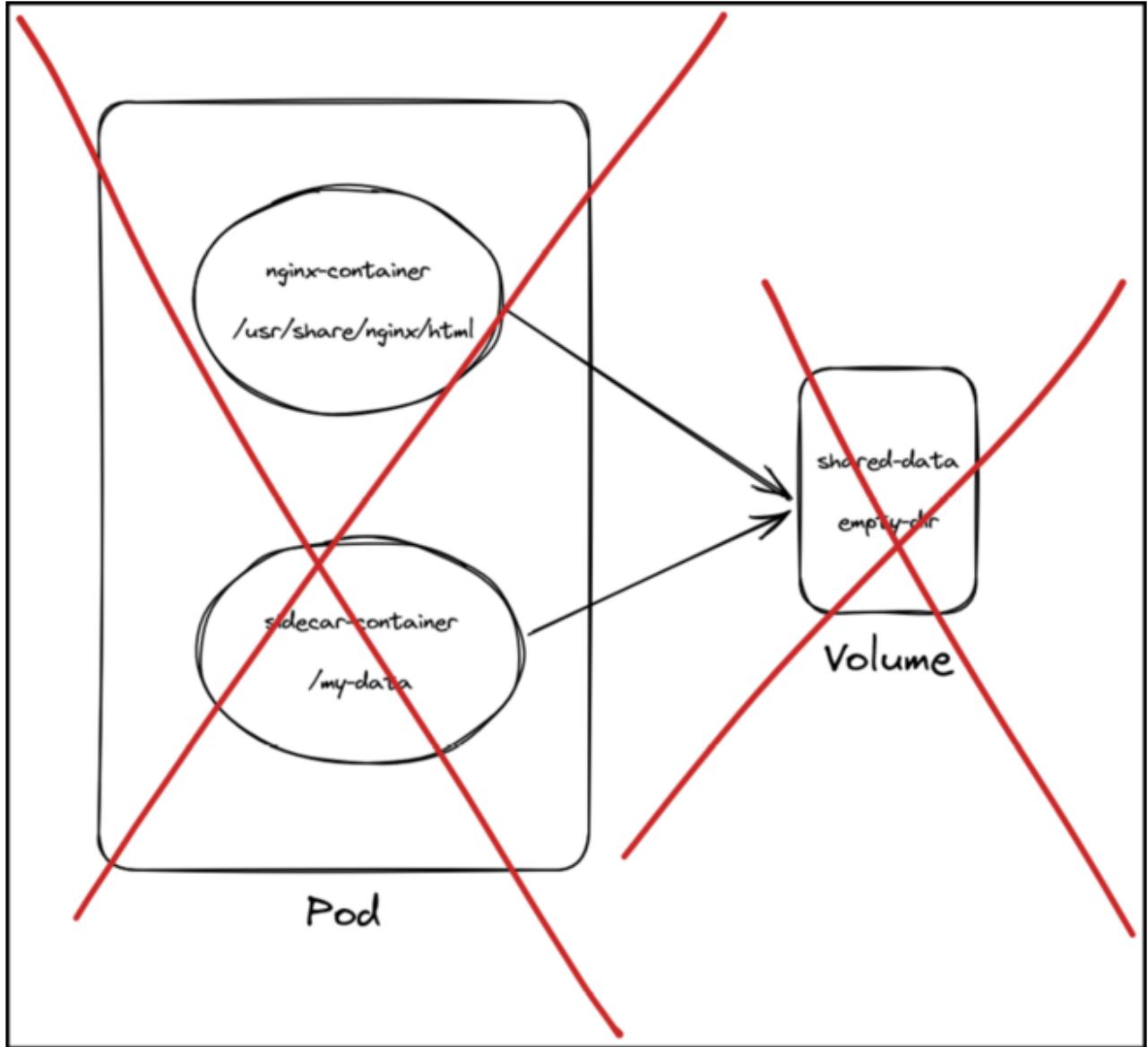


Figura 8.2 Cosa succede quando un Pod con un volume di tipo emptyDir viene cancellato.

hostPath

Il tipo di volume `hostPath` è invece un volume durevole che monta una directory a partire dal filesystem del nodo host in un Pod e il contenuto nel volume rimane intatto anche se il Pod si arresta in modo anomalo, viene terminato o eliminato. Per questo, è importante che la directory e il Pod vengano creati o eseguiti sullo stesso nodo. L'esempio seguente rappresenta una configurazione di questo tipo all'interno di un Pod:

Listato 8.4 Definizione di un volume con `hostPath`

```
...
volumes:
- name: my-volume
  hostPath:
    path: /data
...

```

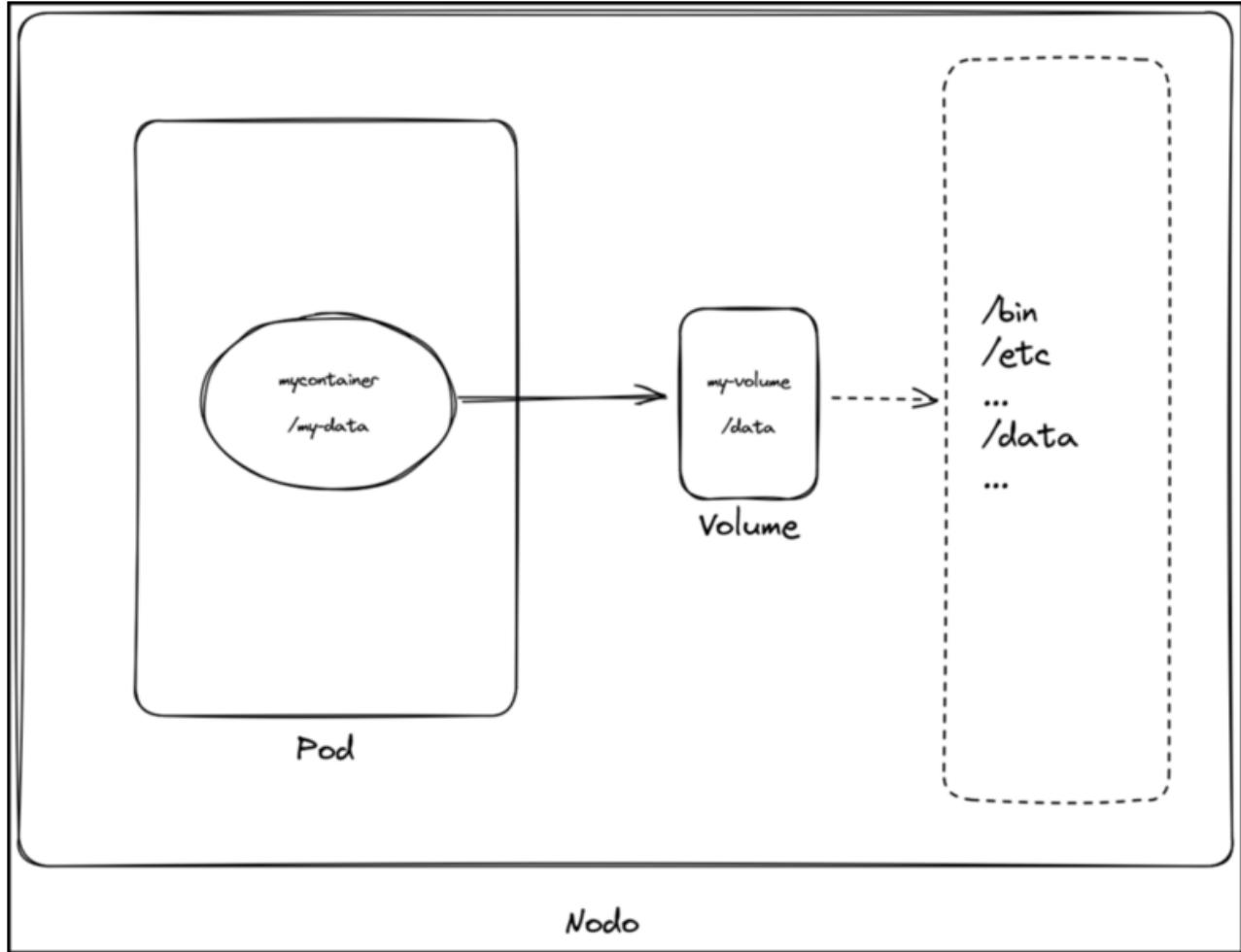


Figura 8.3 Esempio di rappresentazione di un volume di tipo hostPath.

La definizione di un volume `hostPath` è molto simile a quanto visto prima per il volume di tipo `emptyDir`: il nome del tipo di volume è necessario, così come il campo `hostPath`, che definisce la cartella del nodo che il volume utilizzerà:

Listato 8.5 Esempio completo con hostPath

```

apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-app
      image: nginx
      ports:
        - containerPort: 8080
      volumeMounts:
        - name: my-volume
          mountPath: /my-data
  volumes:
    - name: my-volume
      hostPath:
        path: /data

```

Se ripetiamo l'esperimento fatto in precedenza, e quindi tentiamo la creazione di un file all'interno di questa cartella e poi accediamo al nodo che ospita il Pod, vedremo qualche differenza:

Listato 8.6 Test con hostPath

```
# Creazione del pod
kubectl create -f hostPath.yaml

# Elenco dei pod
kubectl get pod
>>>
NAME          READY   STATUS    RESTARTS   AGE
my-pod        1/1     Running   0          33s

# Descrizione del pod
kubectl describe pod my-pod
>>>
Name:           two-containers
Namespace:      default
...
Environment:   <none>
Mounts:
  /my-data from my-volume (rw)
    /var/run/secrets/kubernetes.io/serviceaccount from default-token-a3k1t (ro)
Volumes:
my-volume:
Type:        HostPath

# Entriamo nel pod tramite bash
kubectl exec -it my-pod -- bin/bash

# Creiamo un file nel pod
echo I love Kubernetes > /my-data/some-data.txt

# Entriamo nel nodo tramite SSH
ssh utente@nodo-ip

# Andiamo nella cartella /data
cd /data

# Elenchiamo i file
ls
>>>
some-data.txt

# Mostriamo il contenuto del file
cat some-data.txt
>>>
I love Kubernetes
```

Il file persiste nella cartella `/data` e possiamo leggerne il contenuto. Chiaramente, è vero anche il contrario: se adesso creassimo un file all'interno di quella stessa cartella e provassimo ad accedere nuovamente al container del Pod, leggeremmo sia il file creato in precedenza, sia quello appena creato tramite il filesystem del nodo stesso. Se stai pensando di utilizzare un volume `hostPath` come posto in cui archiviare la directory dei dati di un database, pensa un secondo: dal momento che i contenuti del volume sono archiviati sul filesystem di un nodo specifico, quando il Pod del database viene riavviato, potrebbe essere *pianificato* per l'esecuzione su un altro nodo, e non sarà più in grado di leggere o scrivere dei dati. Questo spiega perché non è una buona idea utilizzare un volume `hostPath` per dei Pod applicativi, poiché rende il Pod *sensibile* al nodo sul quale andrà eseguito; questo tipo di volume è utilizzato, per esempio, da applicazioni di sistema come può essere Fluentd, un sistema che serve a

raccogliere i dati dei diversi Pod che vengono persistiti sempre all'interno della cartella `/var/log` del nodo, oppure per i cluster single-node, come quello creato tramite Minikube o Docker Desktop.

gitRepo

Citiamo qualche altro esempio di tipo di volume che potrebbe tornarci utile, soprattutto se avremo a che fare con persone che sviluppano codice insieme a noi e con cui dobbiamo quindi condividere dei file che vengono continuamente aggiornati: è il caso di `gitRepo`, ossia un volume che monta una directory vuota (non un'emptyDir, attenzione), e clona al suo interno un repository Git nel Pod che lo utilizza. Un esempio di applicazione è il seguente:

Listato 8.7 Esempio di volume con gitRepo

```
apiVersion: v1
kind: Pod
metadata:
  name: server
spec:
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /myrepo
      name: git-volume
  volumes:
  - name: git-volume
    gitRepo:
      repository: "git@somewhere:me/my-git-repository.git"
      revision: "0ac6305c685de2cf605cc128dd219dc3954bb65"
```

In questo caso, creiamo un volume chiamato `git-volume`, all'interno del quale sarà clonato il repository Git chiamato `my-git-repository` e che verrà montato nella cartella del container `nginx` chiamata `myrepo`; un piccolo dettaglio è il campo `revision`, che ci permette di specificare anche il commit di riferimento a partire dal quale clonare il repository, qualora volessimo saltare (o partire da) un punto specifico della storia del nostro repository. Questo può essere quindi un commit specifico, anche un branch come `master`, `develop` e così via.

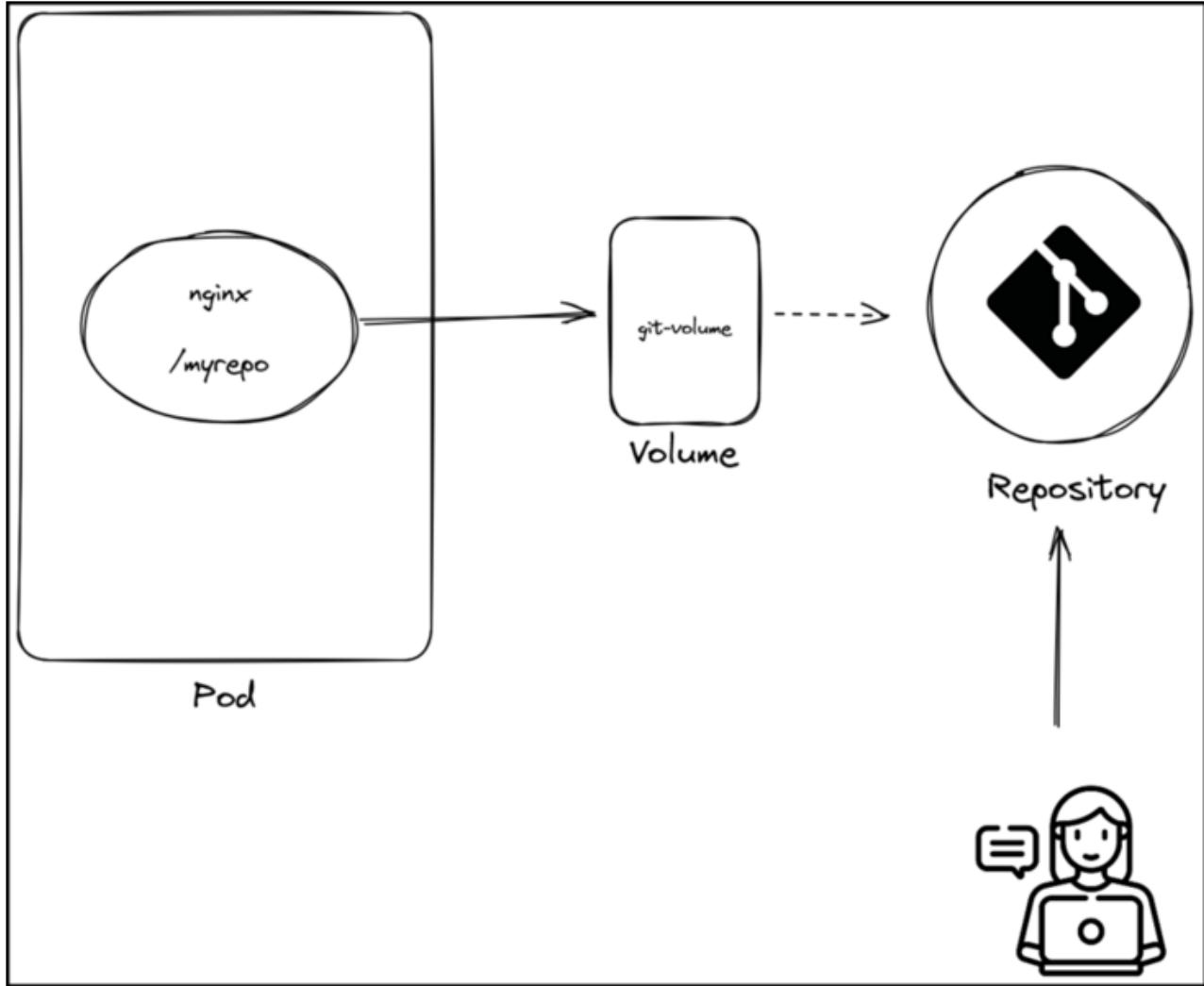


Figura 8.4 Come funziona un volume di tipo gitRepo.

Dopo che il volume `gitRepo` è stato creato, non viene però mantenuto sincronizzato con il repository a cui fa riferimento: i file nel volume non verranno aggiornati quando ci sarà un nuovo commit al repository Git, e questo per la natura del concetto di *clonazione* del repository. Tuttavia, se il Pod è gestito da un oggetto come un `ReplicationController`, l'eliminazione del Pod comporterà la creazione di un nuovo Pod e il volume di questo nuovo Pod conterrà quindi anche gli ultimi commit. Per esempio, puoi utilizzare un repository Git per salvare i file HTML statici del tuo sito Web e creare un Pod contenente un container Nginx (o qualsiasi altro server Web con cui hai dimestichezza) e un volume `gitRepo`: ogni volta che viene creato il Pod, questo estraerà l'ultima versione del tuo sito Web e inizierà a mostrarlo sul browser. Questo piccolo inconveniente può essere comunque risolto con un piccolo "barbatrucco": nei capitoli precedenti abbiamo parlato di un tipo particolare di container che si occupano di eseguire operazioni particolari in background e che affiancano un container principale, il quale è sempre attivo e vigile nello svolgere il suo compito. Questo è il caso perfetto di applicazione di un *sidecar container*, il quale può occuparsi di mantenere aggiornati i file che provengono dal repository Git al posto nostro: per trovare un'immagine esistente che faccia al caso nostro e che mantenga una directory locale sincronizzata con un repository Git, possiamo andare su Docker Hub e cercare "git sync". Troveremo molte immagini che fanno questa operazione e che possiamo utilizzare al posto dell'immagine del sidecar nel Pod dell'esempio precedente per montare il volume `gitRepo` e

configurare la sincronizzazione per mantenere i file sincronizzati con il tuo repository Git. Seguendo un esempio come il seguente (Listato 8.8), se impostiamo tutto correttamente, potremo vedere che i file che il web server sta mostrando sono mantenuti sincronizzati con il repository GitHub. In questo caso, il container `git-sync` si occuperà di mantenere aggiornato i file che vengono dal repository specificato nella variabile di ambiente `GIT_SYNC_REPO` tramite il branch `master` e controllerà se ci sono aggiornamenti ogni 20ms:

Listato 8.8 Esempio di Deployment con gitRepo

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: git-sync
          image: k8s.gcr.io/git-sync:v3.1.3
          volumeMounts:
            - name: www-data
              mountPath: /data
          env:
            - name: GIT_SYNC_REPO
              value: "https://github.com/myuser/myrepo.git"
            - name: GIT_SYNC_BRANCH
              value: "master"
            - name: GIT_SYNC_ROOT
              value: /data
            - name: GIT_SYNC_DEST
              value: "/dest"
            - name: GIT_SYNC_PERIOD
              value: "20"
              - name: nginx-helloworld
            image: nginx
            ports:
              - containerPort: 80
            volumeMounts:
              - mountPath: "/usr/share/nginx/html"
                name: www-data
      volumes:
        - name: www-data
          emptyDir: {}
```

Una nota dolente è che nelle ultime versioni di Kubernetes questo tipo di volume è deprecato: questo vuol dire che non è ancora stato rimosso, ma che lo sarà prossimamente. Una strategia alternativa per ottenere lo stesso risultato descritto prima è quello di eseguire un container con un repository Git che monta una `emptyDir` tramite un `initContainer`, il quale clonerà il repository utilizzando Git, e quindi monterà la stessa cartella nel container principale del Pod.

Se il tuo cluster è in esecuzione su un insieme di server, allora esiste una vasta gamma di altre opzioni supportate per montare l'archiviazione esterna per i volumi che andrai a gestire. Per esempio, hai a disposizione NFS, che sta per *Network File System*: si tratta di un tipo particolare di filesystem condiviso a cui è possibile accedere tramite la rete. Nota: NFS deve già esistere; Kubernetes non esegue direttamente questo filesystem, ma i Pod si limitano ad accedervi.

Un esempio è il seguente:

Listato 8.9 Esempio di Deployment con NFS

```
kind: Pod
apiVersion: v1
metadata:
  name: pod-using-nfs
spec:
  volumes:
  - name: nfs-volume
    nfs:
      server: 10.108.211.244
      path: /data
  containers:
  - name: app
    image: alpine
    volumeMounts:
    - name: nfs-volume
      mountPath: /var/data
      command: ["/bin/sh"]
    args: ["-c", "while true; do date >> /var/data/timestamp.txt; sleep 5; done"]
```

In questo caso, nel campo `volumes` abbiamo specificato come tipologia di volume `nfs` e abbiamo indicato il server da utilizzare e il path della cartella dove andranno scritti i dati; è possibile anche utilizzare il volume dell'esempio precedente per condividere i dati tra diversi Pod nel cluster. Basterà infatti aggiungere il volume a ciascun Pod e aggiungere la specifica per montare il volume da utilizzare per ciascun container.

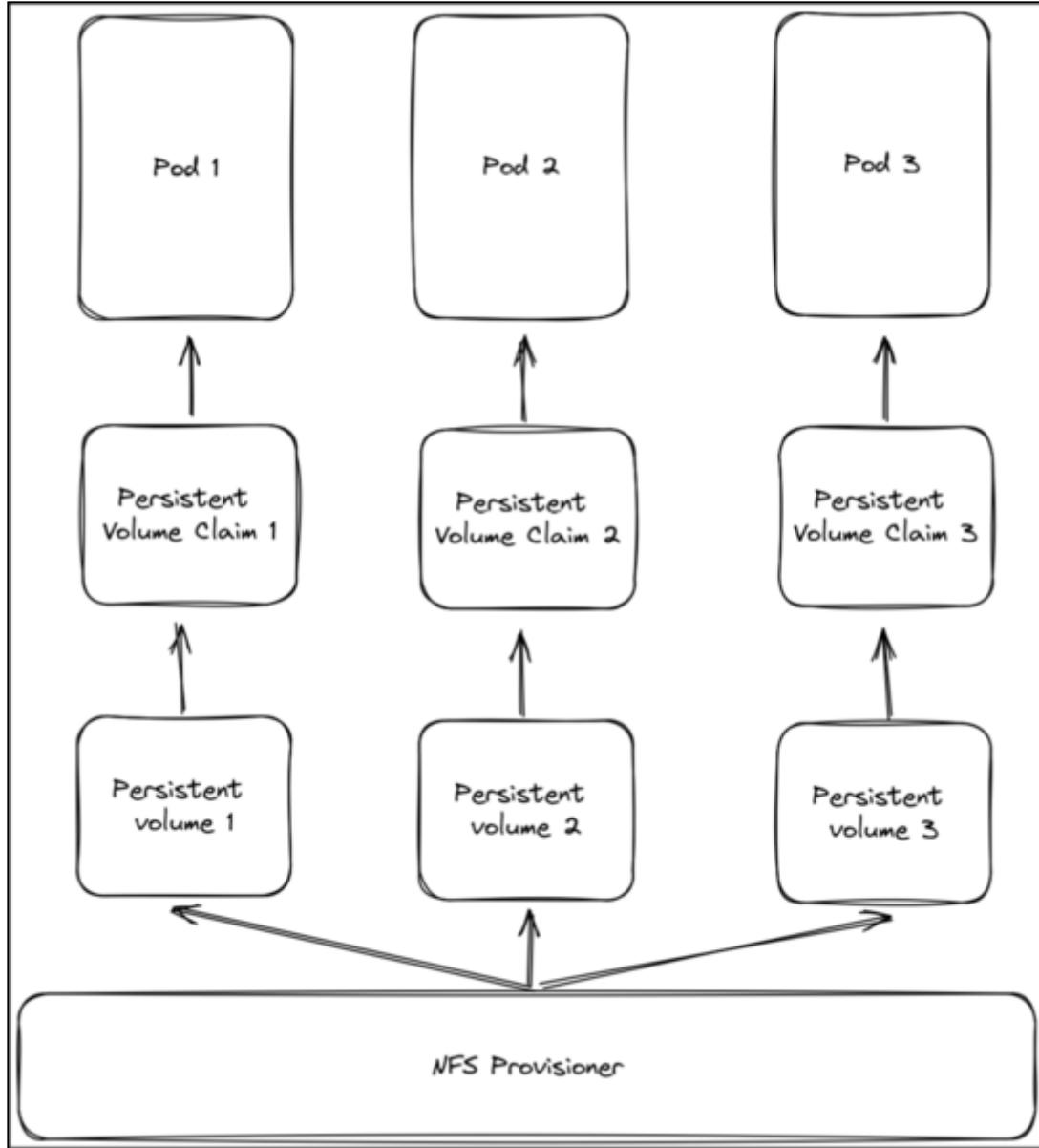


Figura 8.5 Astrazione semplificata di un sistema basato su un provisioner NFS.

CephFS

Ceph è un progetto che nasce per consentire il disaccoppiamento dei dati dall'hardware di storage fisico utilizzando livelli di astrazione software, che forniscono capacità di scalabilità e gestione dei guasti senza pari; in altre parole, è un software per gestire lo storage di oggetti, file e blocchi distribuito e replicato. Ciò rende Ceph ideale per infrastrutture su cloud e per Kubernetes, in quanto può soddisfare efficacemente le esigenze di archiviazione di grandi volumi di dati. Il vantaggio principale di Ceph è che fornisce interfacce per più tipi di storage all'interno di un singolo cluster, eliminando la necessità di più soluzioni di storage o qualsiasi hardware specializzato, riducendo così i costi generali di gestione. Questo significa che è possibile salvare file, ma anche oggetti o blocchi, sfruttando una sola tecnologia. Lo scopo di questo manuale è quello di fornire esempi concreti di ciò con cui potresti doverti trovare a lavorare, per cui non indagheremo ulteriormente su questa tipologia di storage: ci basterà sapere che, nel caso di *CephFS*, parliamo di un tipo di filesystem simile a quello della maggior

parte di quelli che utilizziamo quotidianamente, composto di file e directory. Questo software lavora infatti con tre tipologie di storage: *file* e *directory*, in maniera simile a quanto si ha con un sistema come NFS o EFS, poi *a blocco*, che funziona come un hard drive (o come EBS, in AWS), e infine *a oggetti*, per memorizzare qualsiasi tipo di file, come avviene S3 in AWS.

Un volume di questo tipo consente di montare dello storage *CephFS* (a file) esistente nel tuo Pod: come visto anche in altri casi, a differenza di un volume *emptyDir*, che viene cancellato quando un Pod viene rimosso, il contenuto di un volume di questo genere viene preservato e lo stesso viene semplicemente smontato. Ciò significa che un volume *cephfs* può essere popolato precedentemente con i dati e che questi possono essere condivisi tra diversi Pod. Per funzionare, un tipo di volume come questo avrà bisogno di specifiche che riguardano i monitor Ceph (come gli indirizzi IP o gli hostname dove risiedono), il path ("/" di default), un utente, il file contenente il *keyring* e indicazioni su come questo filesystem deve essere acceduto (in questo caso, in sola lettura).

Listato 8.10 Esempio di Deployment con CephFS

```
apiVersion: v1
kind: Pod
metadata:
  name: cephfs
spec:
  containers:
    - name: cephfs-rw
      image: kubernetes/pause
      volumeMounts:
        - mountPath: "/mnt/cephfs"
          name: cephfs
  volumes:
    - name: cephfs
      cephfs:
        monitors:
          - 10.16.154.78:6789
          - 10.16.154.82:6789
          - 10.16.154.83:6789
        user: admin
        secretFile: "/etc/ceph/admin.secret"
        readOnly: true
```

PersistentVolumes e PersistentVolumeClaims

Se hai letto fin qui, complimenti: non è un argomento facile. Parliamo di infrastrutture e storage, cosa che dovrebbe essere perlopiù materia di persone che se ne occupano tutti i giorni. Tutti i tipi di volume che abbiamo esplorato finora hanno richiesto a chi sviluppa o esegue un Pod di dare un'occhiata (o conoscere in maniera approfondita) l'effettiva infrastruttura di storage disponibile nel cluster. Per esempio, per creare un volume supportato da NFS, chi lo implementa deve conoscere il server su cui si trova il filesystem e quali sono le informazioni utili alla configurazione del Pod. Tutto ciò è contrario all'idea di base di Kubernetes, che mira a nascondere l'infrastruttura effettiva sia all'applicazione che a chi la utilizza, lasciandoci liberi dal doverci preoccupare di queste specifiche e rendendo le applicazioni portatili attraverso un'ampia gamma di provider cloud e on-premise.

Idealmente, infatti, non dovrebbe essere necessario sapere quale tipo di tecnologia di archiviazione viene utilizzata al di sotto del cluster, nello stesso modo in cui non deve sapere quale tipo di server fisico viene utilizzato per eseguire i propri Pod. I rapporti relativi all'infrastruttura dovrebbero essere di esclusiva competenza dell'amministratore del cluster. Questo significa che quando un utente necessita di una certa quantità di storage per la propria applicazione, può richiederlo a Kubernetes, nello stesso

modo in cui può richiedere CPU, memoria e altre risorse durante la creazione di un Pod. L'amministratore di sistema può configurare il cluster in modo che possa fornire alle applicazioni ciò che richiedono nella maniera più trasparente possibile. Per questo, introduciamo due oggetti: PersistentVolumes e PersistentVolumeClaims. Partiamo da quest'idea: invece di immaginare che ogni utente che necessita di un volume specifico per il proprio Pod, sarà l'amministratore del cluster a configurare l'archiviazione sottostante e far in modo che Kubernetes crei per l'utente una risorsa PersistentVolume tramite le proprie API ogni volta che questo ne fa richiesta. Quando un utente deve utilizzare dello spazio persistente in uno dei propri Pod, crea prima un file dove definisce le modalità con cui vuole ottenere lo storage, racchiuse all'interno di un oggetto che si chiama per l'appunto PersistentVolumeClaim, ossia *richiesta* di un PersistentVolume, specificando la dimensione minima e la modalità di accesso con cui vuole utilizzare questo volume. L'utente crea questa claim tramite la sua definizione e Kubernetes trova il PersistentVolume appropriato e associa il volume alla sua richiesta. Quanto descritto fino a poco fa può essere riassunto nella figura seguente, dove immaginiamo che l'infrastruttura sottostante non sia adesso di nostro interesse:

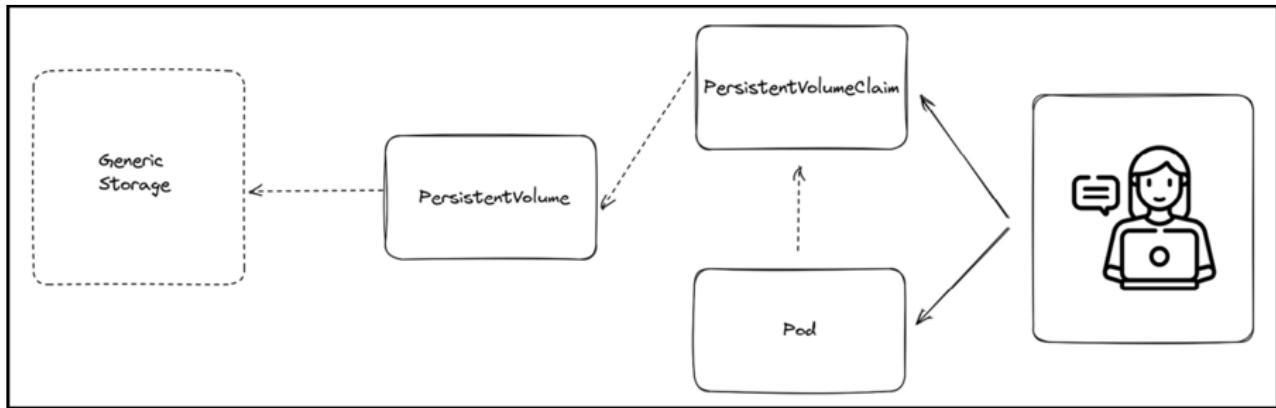


Figura 8.6 Come avviene l'associazione tra una richiesta di persistenza e il tool che si occupa di gestire lo storage.

Una volta che il PersistentVolume è stato creato e quindi la richiesta dell'utente soddisfatta, si dice che il PersistentVolume è stato “boundato” (dall'inglese *to bind*, “legare” o “vincolare”) alla *claim*. Nella rappresentazione non ci siamo preoccupati di cosa avvenga nella parte di sinistra, ma immaginiamo che lo spazio necessario all'utente, nelle modalità con cui l'ha richiesto, vengano soddisfatte e che quindi il Pod possa finalmente persistere i suoi dati.

Un esempio generico di definizione di PersistentVolumeClaim è il seguente:

Listato 8.11 Esempio di PersistentVolumeClaim

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
  requests:
    storage: 3Gi
  storageClassName: ""
  
```

In questo caso, abbiamo definito una richiesta per uno spazio di archiviazione che abbia 3 Gi di memoria (se non ricordi la differenza tra Gi e GB, nessun problema: è infinitesimale e non comporta alcuna differenza per questo esempio), e che abbia come modalità di accesso `ReadWriteOnce`: questo

significa che potremo leggere e scrivere su questo volume solo se il o i Pod risiedono sullo stesso nodo. Esistono infatti diverse modalità di accesso, tra cui anche `ReadOnlyMany` per permettere la sola lettura da più Pod su più nodi, `ReadWriteMany` per lettura e scrittura da più nodi e `ReadWriteOncePod` per la lettura e scrittura da un solo Pod (supportato solo per volume `CSI`, un'interfaccia per lo storage dei container che permette la connessione tra sistemi di archiviazione e orchestratori molto diffuso).

A questa richiesta, verrà quindi associato un `PersistentVolume` che abbia delle caratteristiche tali da soddisfare queste indicazioni: un esempio parziale è il seguente.

Listato 8.12 Esempio di PersistentVolume

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 3Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
...
```

Notare che le caratteristiche principali indicate nella claim corrispondono, e questo permette a Kubernetes di associare alla richiesta uno spazio fisico da allocare; inoltre, è interessante notare come le modalità di accesso espresse tramite il campo `accessModes` siano descritte tramite un array, che quindi consente la specifica di più modalità contemporaneamente. Un'altra informazione è quella relativa alla politica da adottare quando la richiesta eseguita non è più necessaria, come quando un Pod viene cancellato: il campo `persistentVolumeReclaimPolicy` serve infatti a specificare il comportamento di Kubernetes per quanto riguarda questo volume. I casi possibili sono tre: `Retain` indica che questo volume va conservato e che potrà essere utilizzato successivamente per essere associato a un Pod in maniera manuale; `Recycle` significa che il contenuto del volume verrà cancellato e questo potrà essere riutilizzato per un altro Pod che ne faccia richiesta. Infine, è possibile porre come valore `Delete`, che prevede la rimozione definitiva del volume.

La cosa interessante è che, mentre le richieste di un volume sono associate a un namespace, i PV non lo sono: questi infatti non appartengono a nessun contesto particolare, ma sono invece a disposizione del cluster nella sua interezza, come visibile in Figura 8.7.

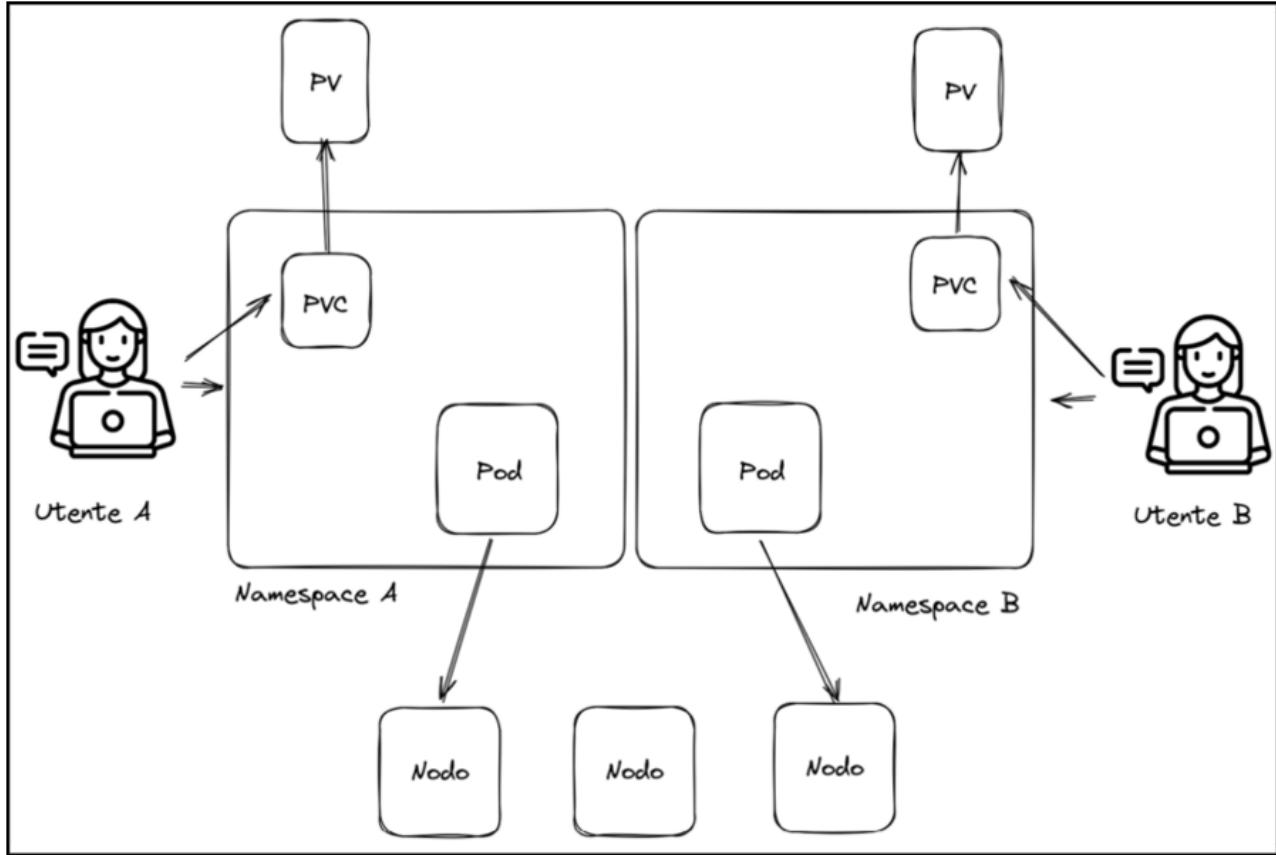


Figura 8.7 Due PV associati a due PVC differenti sono visibili all'interno del cluster, al contrario delle claim, che fanno riferimento al namespace specifico.

Se utilizziamo i due precedenti esempi e procediamo con la relativa creazione, potremo anche analizzarli come sempre tramite `kubectl`: con il comando `kubectl get pv` potremo elencare i PersistentVolume (abbreviati per comodità in `pv`), e con `pvc` le richieste presenti:

Listato 8.13 Test degli esempi fatti in precedenza

```
kubectl get pvc
>>>
NAME      STATUS    VOLUME   CAPACITY  ACCESSMODES  AGE
my-claim  Bound    my-pv    4Gi       RWO          3s

kubectl get pv
>>>
NAME      CAPACITY  RECLAIMPOLICY  ACCESSMODES  STATUS    CLAIM
my-pv    3Gi       Retain        RWO          Bound
default/my-claim
```

Come visibile dall'output, il PersistentVolume è stato associato alla claim definita in precedenza secondo le diverse informazioni descritte nei file e riportate nelle colonne, e questo legame è riportato nell'ultima colonna del secondo comando, dove si specifica qual è il namespace da cui la richiesta proviene. Ora che questo volume esiste, è possibile associarlo a un Pod o un controller semplicemente specificandone il nome nel campo `volumes`:

Listato 8.14 Esempio di un Pod con PersistentVolumeClaim

```
apiVersion: v1
kind: Pod
metadata:
```

```

name: mongodb
spec:
  containers:
    - image: mongo
      name: mongodb
      volumeMounts:
        - name: mongodb-data
          mountPath: /data/db
    ports:
      - containerPort: 27017
        protocol: TCP
    volumes:
      - name: mongodb-data
        persistentVolumeClaim:
          claimName: my-claim

```

Listato 8.15 Esempio di un Deployment con PersistentVolumeClaim

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: ubuntu-deployment
spec:
  selector:
    matchLabels:
      app: ubuntu
  replicas: 1
  template:
    metadata:
      labels:
        app: ubuntu
  spec:
    containers:
      - name: ubuntu
        image: ubuntu
        command:
          - sleep
          - "infinity"
        volumeMounts:
          - mountPath: /app/my-folder
            name: volume
    volumes:
      - name: volume
        persistentVolumeClaim:
          claimName: volume-claim

```

Tutto questo suppone che ci sia però qualcuno, o qualcosa, che si occupa di associare a questa richiesta lo spazio di archiviazione necessario: è questo il giusto momento di introdurre quindi il concetto di StorageClass e provisioner.

StorageClass

Cambiamo cappello e passiamo dal ruolo di sviluppatori a quello di amministratori del cluster: immagina di dover creare manualmente, ogni volta che un utente lo richiede, un volume dedicato a uno o più Pod. Noioso, no? Grazie a questi concetti che stiamo per introdurre, invece di creare PersistentVolume per ogni richiesta, possiamo sfruttare un provisioner che creerà i PersistentVolume attraverso diversi modelli che verranno messi a disposizione per consentire agli utenti di scegliere il tipo desiderato. Gli utenti possono fare riferimento a una cosiddetta StorageClass che descrive le proprietà dello storage a disposizione nelle proprie PersistentVolumeClaim e il provisioner ne terrà conto durante l'allocazione dell'archiviazione persistente, occupandosi in maniera automatica e

dinamica di associare ad ogni richiesta il PV necessario. Kubernetes include diversi provisioner per i fornitori di servizi cloud più diffusi, per cui chi amministra il cluster non dovrà implementare un provisioner ex-novo, ma potrà fruire di uno di quelli messi a disposizione.

Di base, la risorsa StorageClass specifica quale provisioner deve essere utilizzato per poter “staccare” e utilizzare un PersistentVolume quando, attraverso una PersistentVolumeClaim, ne viene fatta richiesta. I parametri definiti nella definizione StorageClass vengono passati al provisioner e sono specifici per ciascun plug-in del provisioner.

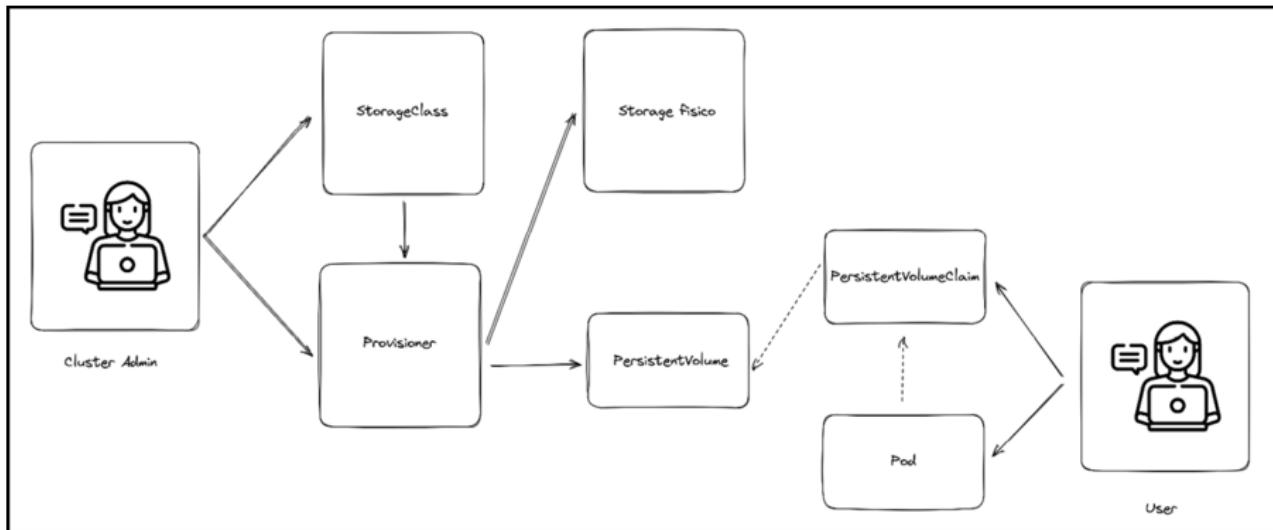


Figura 8.8 Cosa significa disporre di una StorageClass all'interno del cluster e in che modo semplifica il lavoro degli utenti e delle persone che amministrano il sistema.

Chi amministra il cluster può creare più StorageClass a seconda delle prestazioni o di altre caratteristiche. Chi andrà a fare richiesta di un volume, potrà quindi decidere quale sia quello più appropriato per ogni `claim` che crea. La cosa bella delle StorageClass è il fatto che le `claim` possono fare riferimento a esse tramite il loro nome, e questo le rende utilizzabili anche su cluster diversi, a condizione che i nomi utilizzati per le StorageClass siano gli stessi.

Facciamo un esempio pratico: immaginiamo di disporre di un cluster ospitato su AWS, uno dei tanti cloud provider a disposizione, e di voler sfruttare un servizio come EBS (alias di *Elastic Block Storage*) per la creazione dei nostri volumi. Potremo quindi definire una StorageClass come la seguente per permettere ai Pod di associare un volume in maniera dinamica, come mostrato nell'esempio:

Listato 8.16 Esempio di StorageClass per AWS EBS

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: aws-ebs
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  reclaimPolicy: Retain
allowVolumeExpansion: true
mountOptions:
  - debug
volumeBindingMode: Immediate
  
```

In questo caso, il provisioner è AWS EBS (o meglio, un servizio interno a AWS EBS si occuperà dell'allocazione di questi volumi), mentre il nome con cui potremo associare una claim alla StorageClass:

- è pari a `aws-ebs`: questo definisce anche la modalità utilizzata di default per preservare o “buttare” questi volumi, come descritto nel campo `reclaimPolicy`: in questo caso, i volumi verranno conservati. Il campo `volumeBindingMode` controlla quando devono verificarsi l’associazione (ossia il *binding*) del volume e il provisioning dinamico. Quando questa non è impostata, `Immediate` è il valore utilizzato per impostazione predefinita; questa indica che il binding del volume e il provisioning dinamico si verificano dopo la creazione di `PersistentVolumeClaim`.

Un esempio di StorageClass per NFS è invece il seguente:

Listato 8.17 Esempio di StorageClass per NFS

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: example-nfs
provisioner: example.com/external-nfs
parameters:
  server: nfs-server.example.com
  path: /share
  readOnly: "false"
```

Attraverso questo codice, specifichiamo qual è il nome e il server NFS da utilizzare attraverso il suo indirizzo IP o il suo hostname; inoltre, specifichiamo il path della directory esposta dal server e le modalità con cui lo storage può essere acceduto. Potremo inoltre definire quella che è la StorageClass di default da utilizzare quando un utente ne fa richiesta: questo ci permetterà di risparmiare del tempo se nella definizione della `claim` questa informazione dovesse mancare. Per farlo, è sufficiente aggiungere la seguente annotazione evidenziata in grassetto:

Listato 8.18 Configurazione di AWS EBS come default

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: aws-ebs
  annotations:
    storageclass.beta.kubernetes.io/is-default-class: "true"
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Retain
allowVolumeExpansion: true
mountOptions:
  - debug
volumeBindingMode: Immediate
```

A questo punto, una volta che avremo a disposizione almeno una StorageClass, potremo richiedere lo spazio necessario al Pod tramite la consueta `PersistentVolumeClaim`: riprendiamo l’esempio fatto all’inizio e rivediamo la parte relativa alla specifica dell’implementazione da utilizzare: questo ci permetterà di astrarci dal “come” questo storage ci verrà fornito, e demanderemo la sua gestione al provisioner (nell’esempio seguente, ad AWS EBS):

Listato 8.19 Modifica dell’esempio precedente di PersistentVolumeClaim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-claim
spec:
  accessModes:
```

```

        - ReadWriteOnce
resources:
requests:
  storage: 3Gi
storageClassName: "aws-ebs"

```

Cosa succede se il campo che fa riferimento alla StorageClass rimane come stringa vuota? Nel caso in cui una o più StorageClass siano state create, oppure no, questo vuol dire che una `claim` viene associata a un PersistentVolume che è stato creato manualmente, o rimarrà in attesa che questo venga creato e sia possibile associarlo; è possibile utilizzare questa “tecnica” per far in modo che il provisioner dinamico non interferisca.

Listato 8.20 Esempio di PersistentVolumeClaim

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
  requests:
    storage: 3Gi
storageClassName: ""

```

Per riportare l’elenco completo delle StorageClass presenti nel cluster, potremo utilizzare il comando seguente:

Listato 8.21 Elenco delle StorageClass

```

kubectl get sc
>>>
NAME          TYPE
aws-ebs (default)      kubernetes.io/aws-ebs

```

StorageClass e dintorni

Potremmo fare decine di esempi di questo tipo. Ogni provisioner ha le sue specifiche, e quindi le sue definizioni: per non perdere alcun dettaglio sulle modalità che ognuno di essi fornisce, è bene fare riferimento alla documentazione ufficiale, presente sul sito di Kubernetes: <https://kubernetes.io/docs/concepts/storage/storage-classes/>.

Se vogliamo fare un riassunto di quanto visto finora, quando un utente ha bisogno di associare dello spazio di archiviazione al proprio Pod, quello che avviene è rappresentato in Figura 8.9.

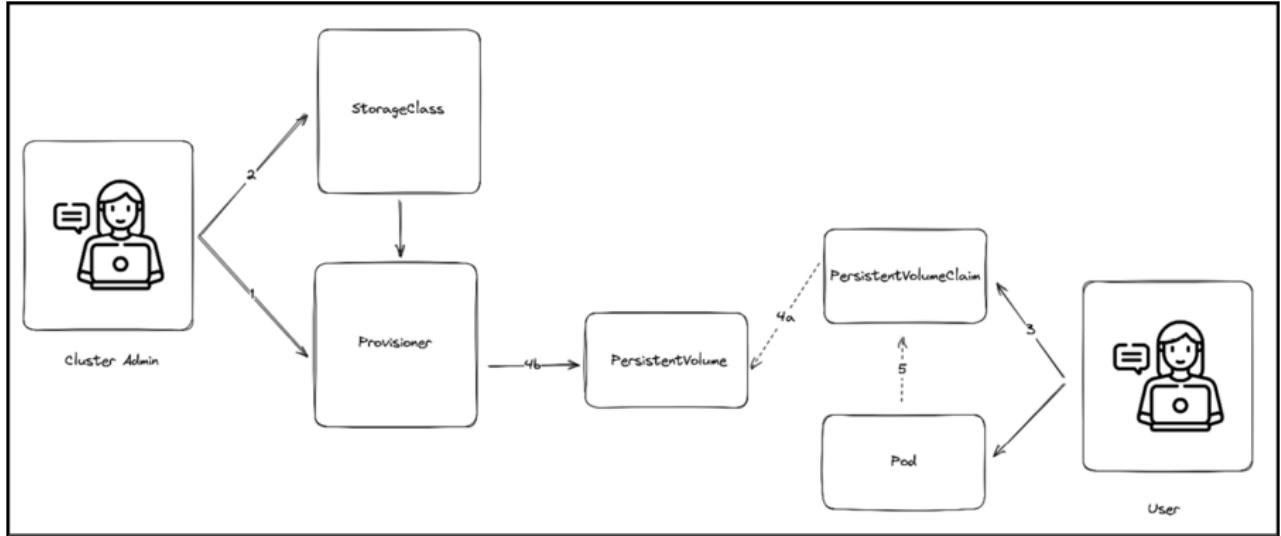


Figura 8.9 Esempio di flusso con gli step per l'associazione di un volume a un Pod.

La persona che amministra il cluster si occupa di decidere qual è il/i provisioner che potranno essere utilizzati all'interno del cluster (passaggio 1 in Figura 8.9); a quel punto, sarà sufficiente definire una o più StorageClass da mettere a disposizione degli utenti che ne avranno bisogno, seguendo le indicazioni del provisioner (passaggio 2). L'utente potrà quindi farne richiesta tramite una claim (passaggio 3), facendo riferimento alla StorageClass da utilizzare (o non specificando niente, e utilizzando quella di default, se prevista), così che Kubernetes possa sfruttare queste informazioni per richiedere la creazione di un PV (passaggio 4a) secondo le indicazioni fornite e associare alla richiesta il PV (passaggio 4b). L'utente potrà quindi referenziare il volume creato tramite il nome della claim (passaggio 5).

Che cosa abbiamo imparato

- Abbiamo visto come creare un Pod multi-container e fare in modo che i container del Pod operino sugli stessi file condividendo il filesystem e aggiungendo un volume al Pod e montandolo in ciascun container, ma anche come:
 - utilizzare il volume `emptyDir` per archiviare dati temporanei e non persistenti;
 - utilizzare il volume `gitRepo` per popolare facilmente una directory con i contenuti di un repository Git all'avvio del Pod;
 - utilizzare il volume `hostPath` per accedere ai file dal nodo host;
 - sfruttare dei provisioner esterni per rendere persistenti i dati del Pod tra i riavvii del Pod;
 - separare il pod dall'infrastruttura di storage utilizzando PersistentVolumes e PersistentVolumeClaims.
- Infine, abbiamo visto in che modo disporre di PersistentVolumes della StorageClass desiderata attraverso il provisioning dinamico.

Risorse aggiuntive

Le persone non colgono le opportunità perché o il tempismo è pessimo, o l'investimento finanziario non è sicuro.
Troppe persone passano il tempo ad analizzare ogni dettaglio. A volte, invece, devi solo provarci.

– Michelle Zatlyn, co-fondatrice di CloudFlare

Quando abbiamo parlato di Pod e di Deployment, abbiamo usato diverse volte il termine *stateless*: questo perché il presupposto è che queste applicazioni non vadano a salvare alcuna modifica all'interno dei container, se non attraverso dei volumi. Abbiamo anche visto come gestire più repliche di un Pod attraverso i ReplicaSet (e i ReplicationController), senza però mai approfondire che cosa succede quando c'è bisogno di gestire la replicazione nel caso di un'applicazione come un database, oppure come gestire un'attività che, in background, compie delle azioni ripetitive al posto nostro. Diamo quindi un'occhiata ad alcune risorse aggiuntive, come gli StatefulSet, DaemonSet e CronJob.

StatefulSet

Ora che sappiamo come eseguire dei Pod, proviamo a spingerci un po' più in là: possiamo sì eseguire diverse istanze del Pod di un web server ma, nel caso di un database, le cose si complicano un po'. I database, per loro natura, servono a persistere delle informazioni e per farlo utilizzano un volume, come un PersistentVolume vincolato da una PersistentVolumeClaim. Questi sistemi servono a memorizzare le informazioni più importanti per l'applicazione e, nel caso di perdita, l'applicazione potrebbe non funzionare correttamente: difatti, per prevenire di incorrere in conseguenze di questo tipo, solitamente un database in un ambiente (non di sviluppo) ha un'architettura come quella rappresentata in Figura 9.1.

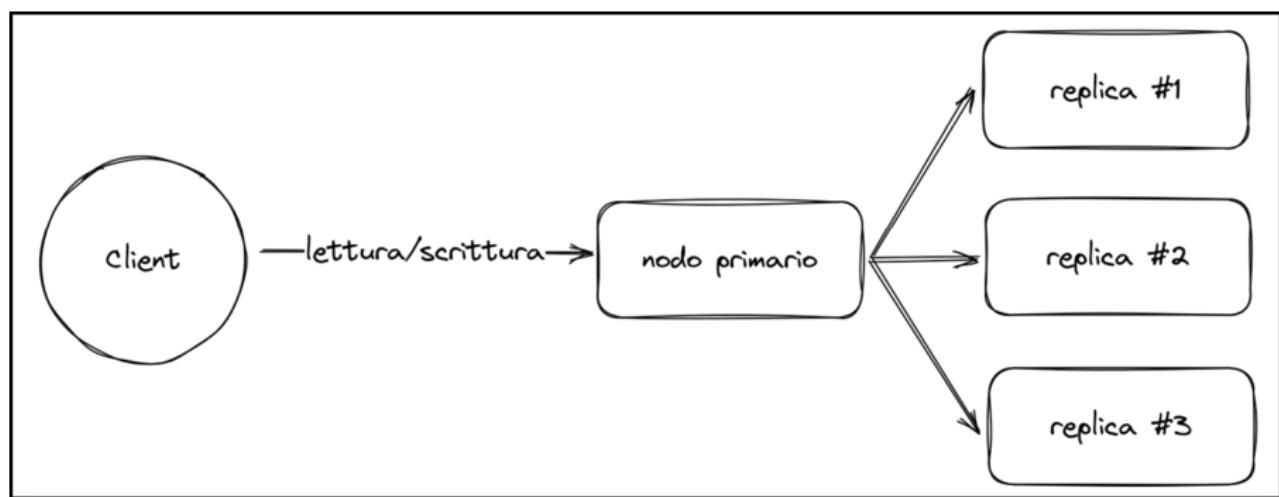


Figura 9.1 Rappresentazione di uno StatefulSet.

Questa architettura ha molte cose in comune con il cluster Kubernetes: c'è solitamente un nodo primario con il suo volume, che si occupa della persistenza e delle operazioni di lettura e scrittura dei

dati che arrivano tramite il client utilizzato dall'utente, e poi ci sono una serie di repliche che servono a tenere una copia dei dati, nel caso ci sia un errore con quello principale. MongoDB è uno dei sistemi che, per esempio, permette la creazione di un cluster come questo: un set di *repliche* MongoDB è infatti un gruppo di uno o più server che contengono la copia esatta dei dati presenti nel sito primario. Sebbene sia tecnicamente possibile avere uno o due nodi replica, il minimo consigliato è tre. Un nodo primario è responsabile di fornire le operazioni di lettura e scrittura dell'applicazione, mentre due nodi secondari contengono una replica dei dati. Tornando a noi, spostiamo il focus su *dove* questi dati vengono salvati: ci si aspetta che, nel caso di un "guasto" al nodo primario, gli altri debbano avere una copia ciascuno delle informazioni salvate per poter rimediare. Possiamo quindi modificare la configurazione precedente nel modo rappresentato in Figura 9.2.

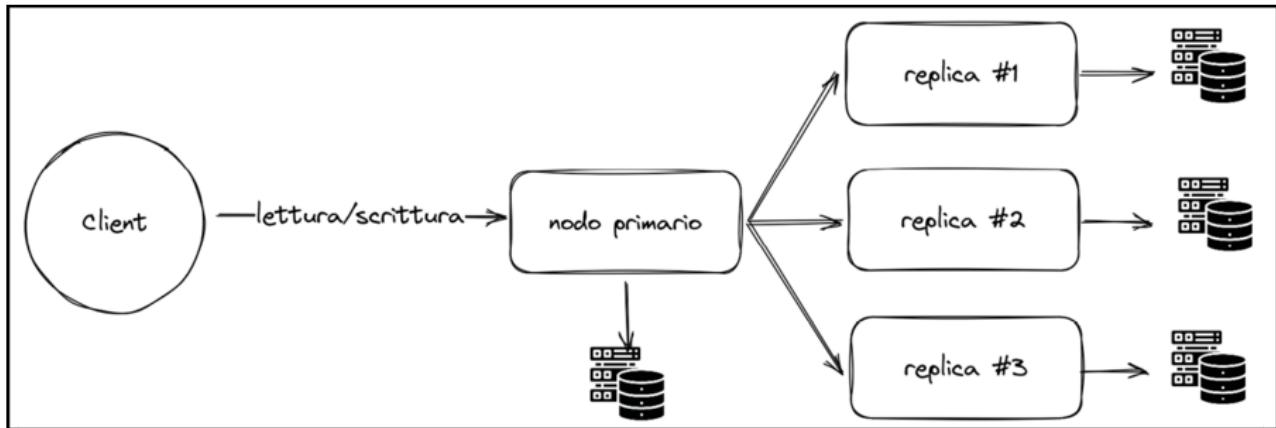


Figura 9.2 Esempio di StatefulSet utilizzato per un database con replicazione dei dati.

Ha senso che ogni replica abbia quindi il suo volume, così che sia resistente ai "guasti". Torniamo adesso al mondo Kubernetes: di base, i ReplicaSet creano più repliche di Pod a partire da un singolo template che descrive un Pod e queste repliche non differiscono tra loro in alcun modo, a parte il nome assegnato al Pod e l'indirizzo IP. Se nella definizione del Pod si include anche un volume che fa riferimento a una specifica PersistentVolumeClaim, tutte le repliche del ReplicaSet utilizzeranno esattamente lo stesso volume associato a questa richiesta, il che è ben differente dalla situazione descritta nell'immagine precedente. Altra nota riguarda l'indirizzo IP: ricordando la definizione di Pod, per la sua natura di risorsa *transiente*, quando questo viene riavviato cambia il nome e cambia l'indirizzo IP.

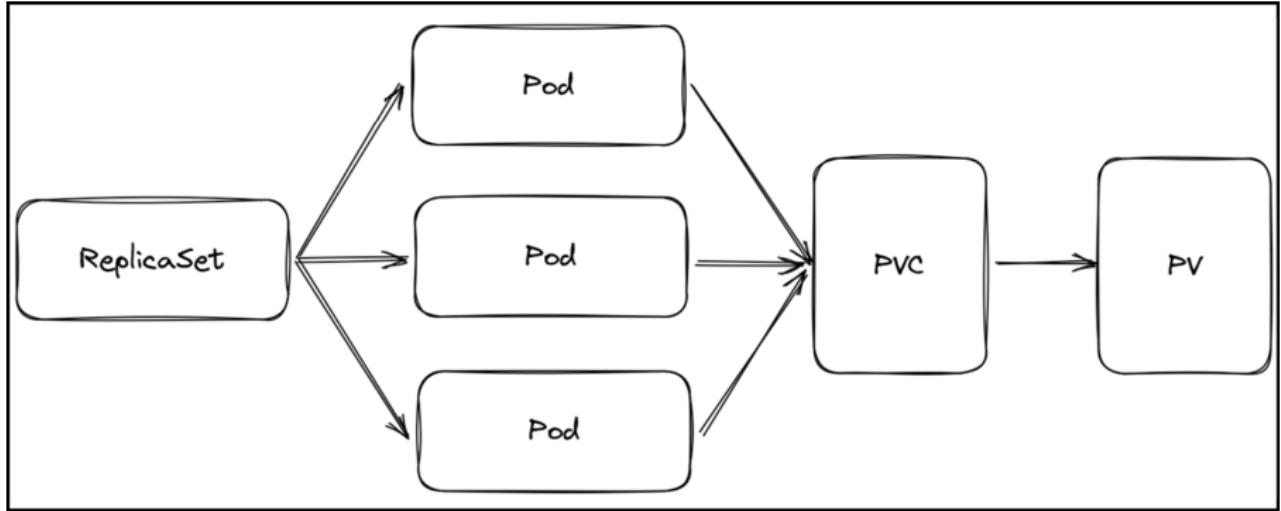


Figura 9.3 In che modo un ReplicaSet differisce da uno StatefulSet.

A questo punto, possiamo affermare che un semplice ReplicaSet (o anche un ReplicationController) non è adatto a questo compito: ci serve qualcosa di più *stabile*. È qui che entrano in gioco gli StatefulSet: si tratta di risorse che sono create appositamente per essere *cucite* sulle istanze dei Pod le cui repliche non possono essere trattate come semplici “copie” dell’applicazione, ma che hanno un nome e soprattutto uno stato proprio. Proviamo con una metafora piuttosto potente: se pensiamo a un’applicazione, le persone che sviluppano potrebbero trattare le istanze della propria applicazione con i relativi microservizi come fossero degli animali domestici, assegnando a ciascuna istanza un nome e cercando di monitorare lo stato della singola istanza; quando abbiamo avuto a che fare con i Pod, e più in generale con i ReplicaSet, abbiamo lavorato con questi oggetti come fossero un branco di oggetti identici (non a caso, si chiamano *Pod*), per cui non abbiamo prestato particolare attenzione alla singola istanza, ma guardato piuttosto lo stato complessivo). D’altra parte, con le applicazioni *stateful*, un’istanza dell’app è più simile a un cucciolo, che non può essere facilmente sostituito, la cui identità è importante così come il suo stato di salute. Questo vale per i Pod che fanno parte di uno StatefulSet: ogni Pod conta e non è uguale all’altro.

PetSet vs CattleSet

Gli StatefulSet non hanno sempre avuto questo nome: una volta si chiamavano *PetSet*, per l’analogia con gli animali domestici. In inglese, *pets* sta proprio per “cuccioli”, o “animali domestici”, mentre *cattle* rappresenta il bestiame. Curioso, no?

Uno StatefulSet si assicura che i Pod vengano ripianificati in caso di errore in modo tale da mantenere la loro identità e il loro stato, e consentendo anche di aumentare e diminuire facilmente il numero di istanze che sono in esecuzione. Uno StatefulSet, come un ReplicaSet, ha un campo per definire il numero delle repliche desiderato che determina quanti “cuccioli” di Pod vuoi avere in quel momento. Analogamente ai ReplicaSet, i Pod vengono creati a partire dal template di un Pod che è parte della definizione stessa dello StatefulSet, ma la differenza sta nel fatto che questi oggetti non sono repliche esatte l’uno dell’altro: ognuno può avere il proprio insieme di volumi, avrà il suo nome univoco e la sua gestione per la comunicazione con altri applicativi.

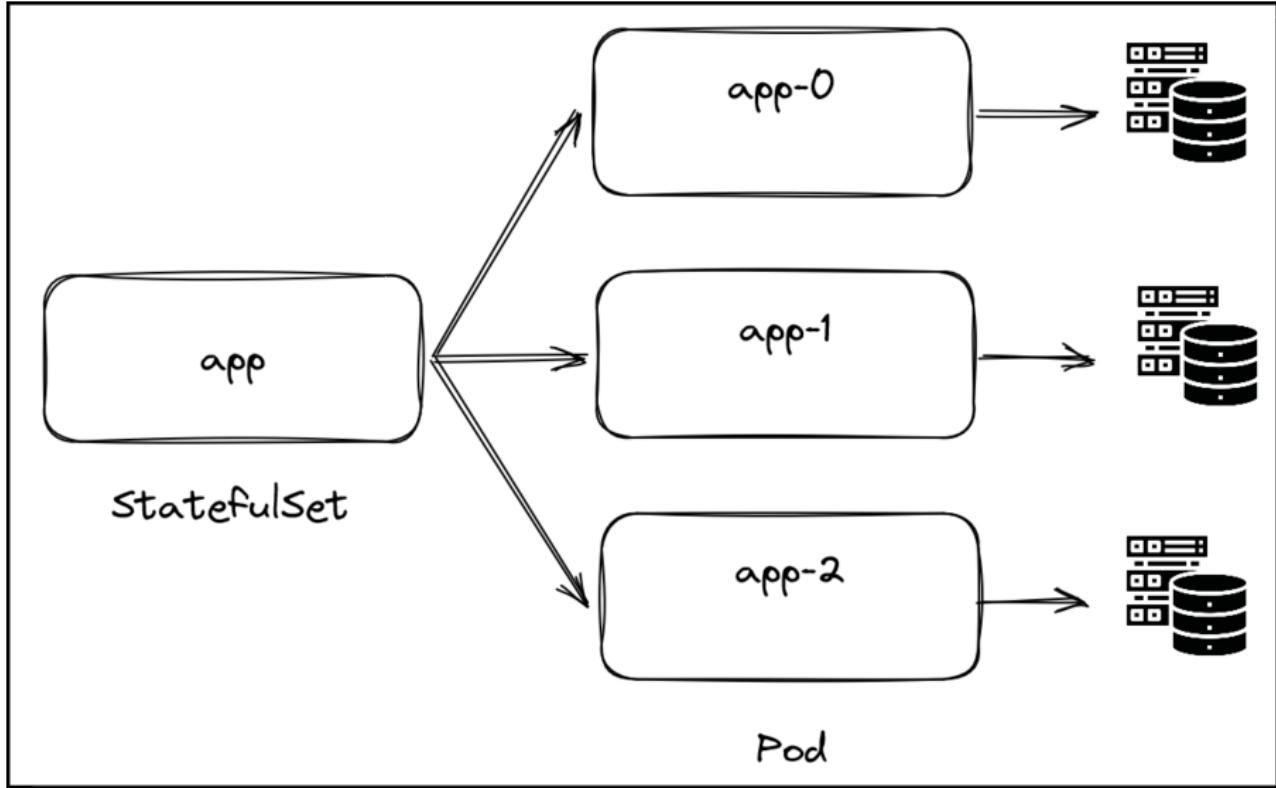


Figura 9.4 Come uno StatefulSet associa un'identità e un volume persistente a ogni Pod.

Come mostrato anche in Figura 9.4, ogni StatefulSet ha un proprio nome, e i Pod che dipendono da questa risorsa assumono un nome il cui suffisso è un numero che, seguendo la notazione *zero-based*, parte da zero e cresce. In questo modo, i nomi dei Pod diventano prevedibili e sono fissi: se il Pod *app-0* verrà riavviato, ne verrà eseguito uno in sostituzione con lo stesso identico nome. Ma non si tratta solo di Pod che hanno un nome univoco: a differenza dei normali Pod, questi a volte devono essere raggiungibili tramite il loro *hostname*. Riprendendo l'esempio del database, potremmo pensare di voler raggiungere una delle istanze replicate, piuttosto che quella primaria: al contrario dei Pod "tradizionali", avere a disposizione un *hostname* che ci permetta di raggiungere quello specifico Pod è fondamentale. Per questo motivo, uno StatefulSet richiede la creazione di un Service definito come *headless* che viene utilizzato per fornire l'effettiva identità di rete a ciascun Pod. Attraverso questo servizio, ogni Pod ottiene il proprio record sul DNS, quindi tutte le applicazioni che hanno bisogno di comunicare con le singole istanze, potranno farlo tramite il nome specificato.

Nella figura seguente si vede come ogni Pod abbia il suo *hostname* specifico: per fare riferimento a uno dei Pod e contattarlo per poter comunicare, è sufficiente utilizzare il suo nome (*app-0*, per esempio), il nome del Service (nel nostro caso, *myservice*), il namespace dove sono in esecuzione (in figura, *default*) e il suffisso *svc.cluster.local* che indica che si tratta di un Service (*svc*, abbreviazione di `kubectl` per queste risorse) unitamente al dominio di default assegnato al cluster.

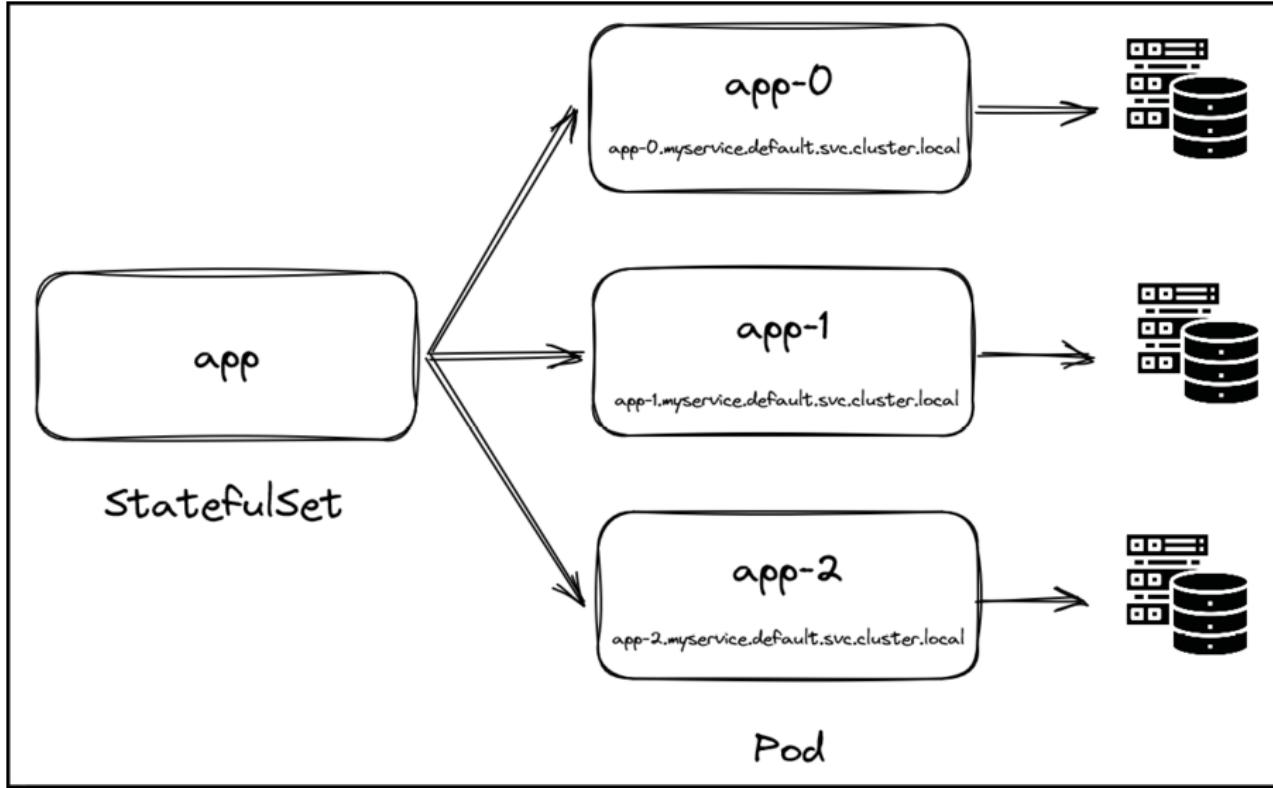


Figura 9.5 Rappresentazione dei Service associati ai Pod dello StatefulSet.

Così come avviene con i ReplicaSet, se uno dei Pod dovesse essere terminato, lo StatefulSet si occuperebbe di crearne un altro a sostituzione di quello perso, mantenendo le stesse informazioni che c'erano in precedenza. Com'è facile immaginare, se venisse eliminato lo StatefulSet o venisse modificato il numero di repliche che gestisce, queste andrebbero sempre a rispettare la *naming convention* che usa il suffisso numerico: se scaliamo a 2 Pod quello in esempio, il Pod eliminato sarà quello il cui nome è `app-2`, così come se questo oggetto venisse eliminato; i suoi Pod verrebbero rimossi a partire da quello con il numero più alto, e a scendere. La ragione per cui i Pod verranno eliminati uno per volta è che alcune applicazioni non gestiscono correttamente un cambiamento in termini di numerosità troppo rapido: in questo senso, gli StatefulSet ridimensionano solo un'istanza di Pod alla volta, per evitare che si perdano dei dati se più istanze si arrestano contemporaneamente.

Ora che abbiamo iniziato a delineare un'idea di che cosa sono gli StatefulSet, vediamo un esempio concreto: definiamo in un file YAML un cluster MongoDB con tre repliche in ascolto sulla porta 27017.

Listato 9.1 Esempio di StatefulSet per MongoDB

```

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongo
spec:
  serviceName: "mongo"
  replicas: 3
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
        - name: mongodb

```

```

image: mongo:6.0.4
command:
  - mongod
  - --replSet
  - rs0
ports:
  - containerPort: 27017
  name: peer

```

Come vedi nell'esempio, la definizione di uno StatefulSet è molto simile a quella di un ReplicaSet: gli viene assegnato un nome nel campo relativo ai metadati, eventualmente delle label, e poi si passa alla parte di specifica dei container: in questo caso, viene utilizzata l'immagine di MongoDB nella versione 6.0.4 e all'interno viene avviato il demone `mongod`.

Listato 9.2 Avvio dei Pod dello StatefulSet

```

kubectl apply -f statefulset.yaml
>>>
statefulset.apps/mongo created

kubectl get pods
>>>
NAME      READY   STATUS    RESTARTS   AGE
mongo-0   1/1     Running   0          26s
mongo-1   1/1     Running   0          17s
mongo-2   1/1     Running   0          7s

```

Fatto questo, dobbiamo creare un Service *headless* che ci permetta di raggiungere le diverse istanze: per chiarire, *headless* significa che questo Service non avrà un indirizzo IP virtuale assegnato e un load balancer che smisti il traffico tra le diverse repliche, ma piuttosto un *hostname* che sarà possibile utilizzare per raggiungere i diversi Pod.

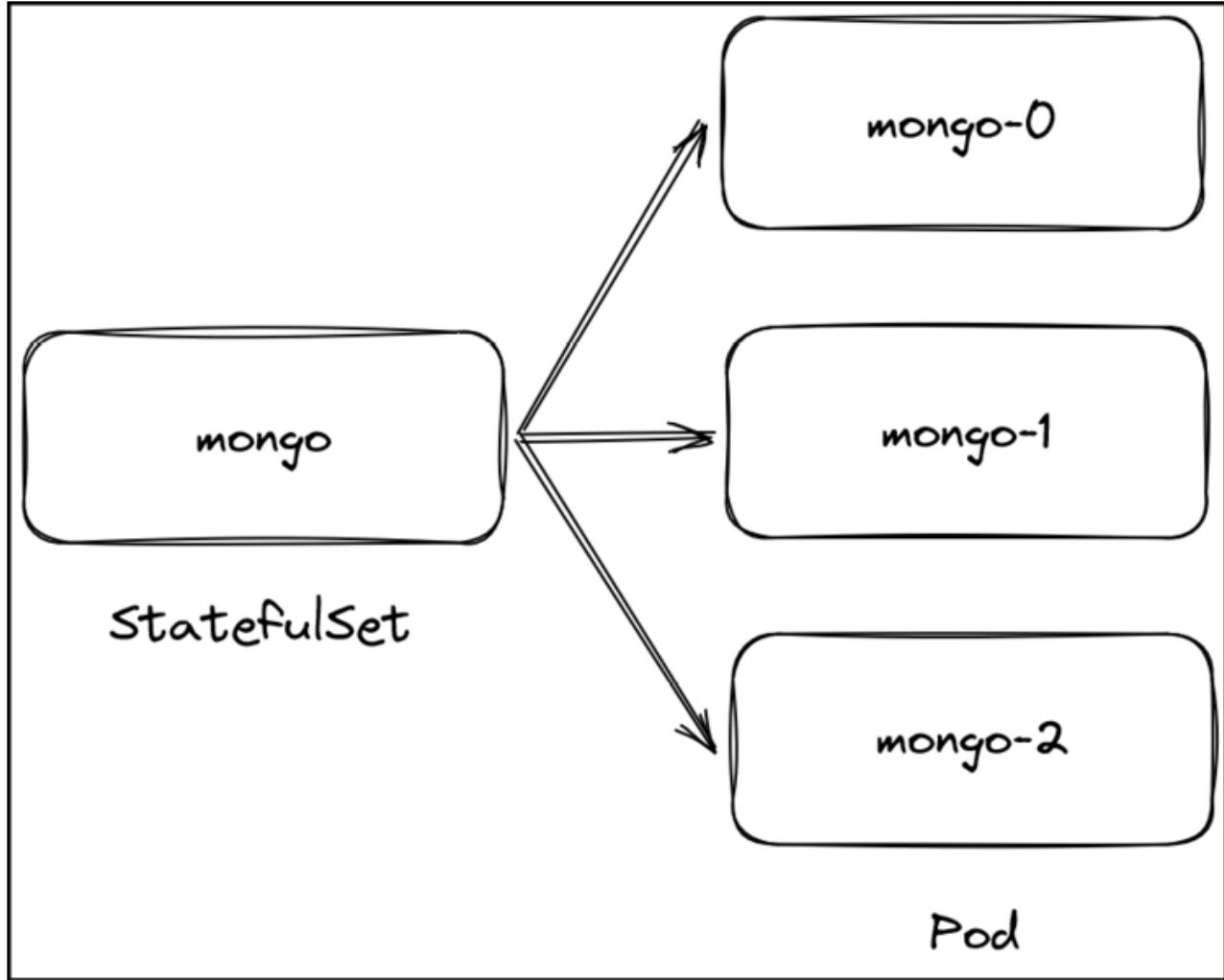


Figura 9.6 Esempio di StatefulSet per MongoDB.

Listato 9.3 Esempio del Service headless per MongoDB

```
apiVersion: v1
kind: Service
metadata:
  name: mongo
spec:
  ports:
    - port: 27017
      name: peer
  clusterIP: None
  selector:
    app: mongo
```

Listato 9.4 Creazione Service per MongoDB

```
kubectl apply -f service.yaml
>>>
service/mongo created

kubectl get svc
>>>
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)
AGE
```

mongo	ClusterIP	None	<none>	27017/TCP
6s				

A differenza di quanto visto con altri Service, qui il ClusterIP non ha un valore e per contattare uno dei Pod appena creati avremo bisogno solamente del nome del Pod e del nome del servizio:

Listato 9.5 Esempio dell'hostname per comunicare con il Pod mongo-1

```
mongo-1.mongo.svc.cluster.local

# oppure

mongo-1.mongo # è possibile omettere il namespace se il pod si trova in quello corrente
```

E per la persistenza? L'approccio che prevede l'utilizzo di volumi è molto simile a quanto visto per istanze come quelle *singole*, ma con una piccola modifica: quando specifichiamo all'interno della definizione dei container i volumi, non è sufficiente indicare la `claim` da utilizzare, ma serve indicare anche le informazioni che descrivono lo storage da aggiungere. Questo perché lo StatefulSet dovrà creare delle copie identiche di storage per ogni Pod replicato al suo interno, dove l'unica differenza starà nel nome: il controller andrà a richiedere la creazione di 3 volumi che avranno il nome specificato come suffisso (in questo caso, `database`) seguito dal nome del Pod a cui fanno riferimento (per esempio, `mongo-0`, `mongo-1` e così via).

Listato 9.6 Aggiunta dei volumi allo StatefulSet per MongoDB

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongo
spec:
  selector:
    matchLabels:
      app: mongo
  serviceName: "mongo"
  replicas: 3
  template:
    metadata:
      labels:
        app: mongo
  spec:
    containers:
      - name: mongodb
        image: mongo:6.0.4
        command:
          - mongod
          - --replSet
            - rs0
        ports:
          - containerPort: 27017
            name: peer
        volumeMounts:
          - name: database
            mountPath: /data/db
  volumeClaimTemplates:
  - metadata:
      name: database
    spec:
      accessModes: [ "ReadWriteOnce" ]
  resources:
    requests:
      storage: 100Mi
```

Affinché questi volumi persistenti replicati funzionino correttamente, è necessario che sia configurato il provisioning automatico per i volumi persistenti oppure è necessario creare dei volumi persistenti che lo StatefulSet potrà utilizzare; se questo non troverà dei volumi che corrispondono alle richieste, non sarà in grado di creare i Pod corrispondenti.

DaemonSet

Negli esempi visti finora, i Pod sono replicabili all'interno di un Deployment, anche grazie al supporto di risorse come i ReplicaSet. Questi Pod potranno essere messi in esecuzione su un nodo piuttosto che un altro a seconda delle disponibilità del cluster, ma generalmente non spetta all'utente decidere quale nodo utilizzare. Se invece ci trovassimo nel caso in cui l'applicazione debba essere eseguita su ogni singolo nodo presente nel cluster? In questo caso, i DaemonSet sono la risorsa perfetta: questo tipo di oggetto serve ad assicurarsi che una copia del Pod sia in esecuzione all'interno di tutti i nodi del cluster. Per fare un caso d'uso reale, i DaemonSet vengono impiegati per rilasciare applicazioni come collettori di log o agenti che monitorano il cluster, e che quindi hanno bisogno di accedere ai singoli nodi per poter raccogliere informazioni necessarie a questo tipo di analisi. Come i ReplicaSet, queste entità creano dei Pod che ci si aspetta siano in esecuzione a lungo come servizi all'interno del cluster e il cui stato e attività devono essere costantemente salvaguardate. Al contrario dei ReplicaSet, i DaemonSet devono essere scelti come risorsa che *ingloba* i Pod solo nel caso in cui ci sia la necessità effettiva che una copia di ognuno di essi giri su ogni nodo presente nel cluster, situazione che non avviene attraverso l'uso dei ReplicaSet: sarà lo scheduler Kubernetes a scegliere il nodo più adatto a ospitare il Pod che deve essere avviato.

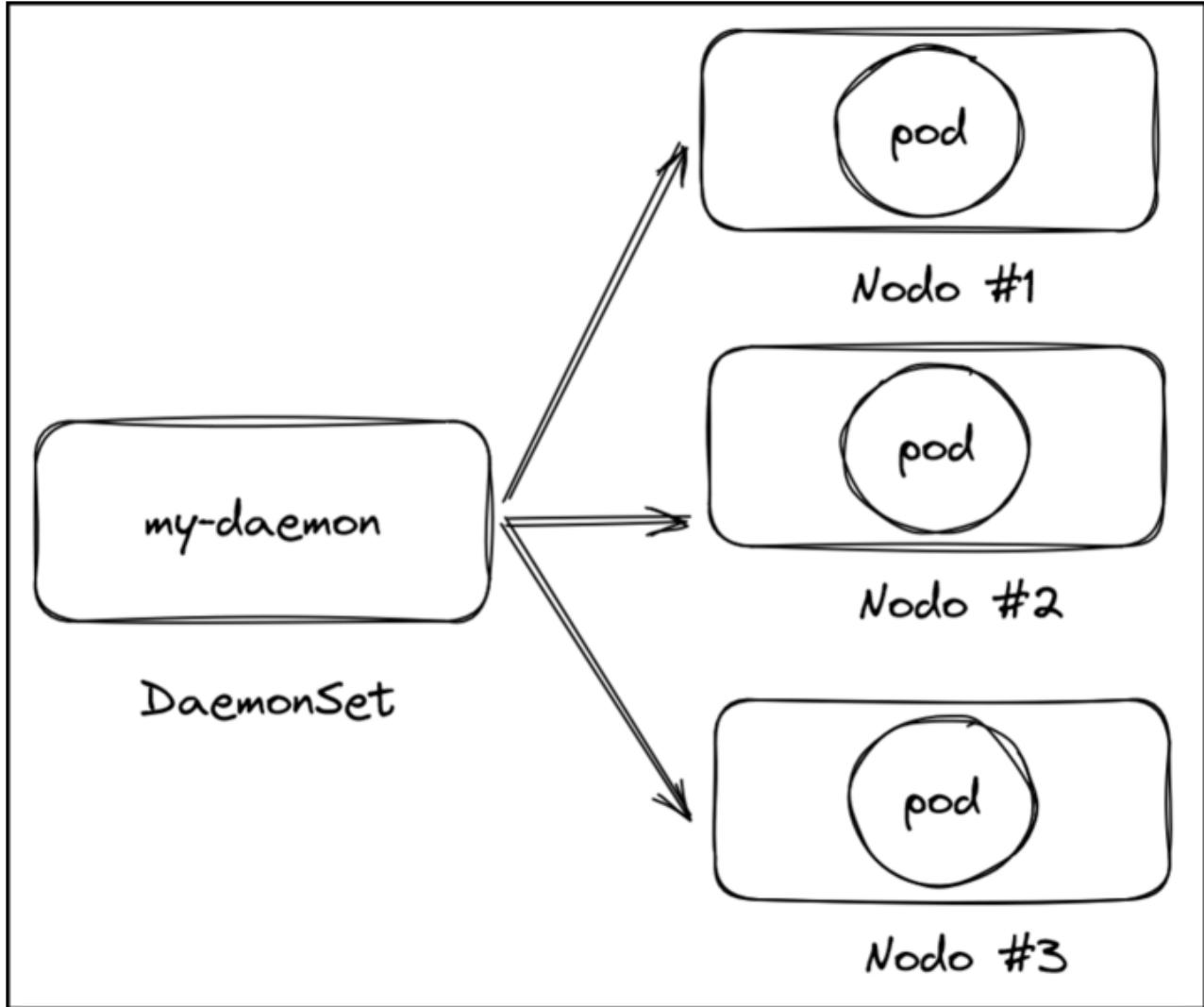


Figura 9.7 Rappresentazione di un DaemonSet.

Per impostazione predefinita, un DaemonSet creerà una copia di un Pod su ogni nodo (*applicativo*, i *control-plane* hanno altre mansioni), a meno che non venga utilizzato un selettore, e quindi una label che specifica i nodi da includere, che limiterà i nodi *idonei* a quelli con un set di etichette corrispondenti. I DaemonSet determinano su quale nodo verrà eseguito un Pod al momento della creazione dello stesso, specificando questa informazione attraverso il campo `nodeName` all'interno della propria definizione. Di conseguenza, i Pod creati da DaemonSet vengono completamente ignorati dallo scheduler Kubernetes, che ne demanda la gestione ai DaemonSet.

Sia i DaemonSet che i ReplicaSet sono la dimostrazione di come Kubernetes adotti un pattern architetturale fortemente disaccoppiato anche nei suoi componenti: il cuore di tutto ciò che abbiamo visto finora è e rimane il Pod, che ha un suo modello con le relative proprietà; i controller che scegiamo di adottare cambiano in base alle esigenze, e il Pod non è che l'esecutore di quanto descritto all'interno degli attributi macroscopici del suo gestore. In questo modo, non c'è bisogno di scrivere più e più volte il modello che descrive un Pod nel caso in cui si scelga di cambiare strategia, ma è sufficiente adottare la stessa definizione e poi lasciare che sia il controller a portare avanti le proprie strategie. Questo vuol dire anche che quando aggiungi un nuovo nodo al cluster, vi viene aggiunto un Pod dal DaemonSet. Allo stesso modo, quando rimuovi un nodo dal tuo cluster, il Pod viene rimosso insieme a lui; solo

l'eliminazione di un DaemonSet elimina i Pod che sono stati creati in precedenza. In questo senso, un Daemonset non è altro che un controller che gestisce dei Pod esattamente come avviene per Deployment, ReplicaSet e StatefulSet, ma che è stato creato per uno scopo specifico: garantire che i Pod siano in esecuzione su tutti i nodi del cluster.

Vediamo un esempio di DaemonSet e della sua definizione e partiamo da un esempio piuttosto classico, ma molto utile: utilizziamo Fluentd, un software che raccoglie dati (in questo caso log) e che quindi deve essere presente su ogni nodo del cluster. Si tratta di uno strumento molto adottato all'interno di architetture basate su Kubernetes, come anche su OpenShift: permette raccogliere i log prodotti dai vari Pod, per poi elaborarli e aggiungere delle informazioni di contesto, modificando la struttura dei log e quindi inoltrandoli al componente che si occupa della persistenza dei log, come può essere Elasticsearch. Qui non è tanto importante il come avviene questo flusso di lavoro, ma la finalità di Fluentd: prendere tutti i log (che vengono inviati sullo `stdout`, ossia il terminale dei singoli Pod) e raccoglierli per poterli archiviare altrove. Per fare tutto questo, è necessario che sia presente un agente di Fluentd su ogni nodo per poter leggere questi log e raggruppargli nel database che li andrà a persistere. Qui il DaemonSet calza a pennello: la sua definizione è estremamente semplice ed è simile a quanto visto per altri controller: dopo aver specificato un nome ed eventualmente il namespace che dovrà ospitarlo (che, per convenzione, è `kube-system`), si definiscono i dettagli dei Pod che verranno avviati: tra queste informazioni, troviamo l'immagine di Fluentd, le risorse necessarie ad avviarli, i volumi da utilizzare, e così via:

Listato 9.7 Esempio parziale di definizione di DaemonSet per Fluentd

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
    version: v1
spec:
  selector:
    matchLabels:
      k8s-app: fluentd-logging
  version: v1
  template:
    metadata:
      labels:
        k8s-app: fluentd-logging
        version: v1
    spec:
      tolerations:
        - key: node-role.kubernetes.io/control-plane
          effect: NoSchedule
        - key: node-role.kubernetes.io/master
          effect: NoSchedule
      containers:
        - name: fluentd
          image: fluent/fluentd-kubernetes-daemonset:v1-debian-elasticsearch
          env:
            - name: K8S_NODE_NAME
              valueFrom:
                fieldRef:
                  fieldPath: spec.nodeName
            - name: FLUENT_ES_HOST
              value: "elasticsearch-logging"
            - name: FLUENT_ES_PORT
              value: "9200"
```

```

...
      resources:
        limits:
          memory: 200Mi
          requests:
            cpu: 100m
        memory: 200Mi
      volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: dockercontainerlogdirectory
          mountPath: /var/log/pods
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
        - name: varlog
          hostPath:
            path: /var/log
        - name: dockercontainerlogdirectory
          hostPath:
            path: /var/log/pods

```

Nella definizione di questa risorsa ci sono alcuni aspetti su cui vale la pena soffermarsi, partendo dall'immagine: in questo caso, si dà per scontato che la persistenza dei log che Fluend andrà a raccogliere verrà eseguita su Elasticsearch, un motore di ricerca e analisi distribuito, gratuito e aperto per tutti i tipi di dati, inclusi testuali, numerici, geospaziali, strutturati e non strutturati che è molto comune come "collega" di Fluentd. Inoltre, all'interno delle specifiche del DaemonSet vengono indicate delle *toleration*: nel capitolo dedicato alle *best practice* in ambito enterprise parleremo di questi strumenti, ma per ora ci basti sapere che, attraverso quelle poche righe, il DaemonSet specifica che non verranno eseguiti dei Pod sui nodi *control-plane* (precedentemente indicati come *master*). Poi, all'interno della definizione dei container, sono presenti le informazioni di connessione all'istanza di Elasticsearch, che in questo caso si suppone sia sempre interna al cluster e che abbia un Service con il nome `elasticsearch-logging` in ascolto sulla porta 9200. In ultimo, ma non in ultimo, nella sezione dedicata alle *mount* dei volumi, viene specificata l'ubicazione dei log dei Pod, che sono storicamente memorizzati nella cartella `/var/log`, ma nel caso in cui questi si trovino sotto `/var/lib/docker/containers`, è possibile cambiare il path. Una cosa che vale la pena sottolineare è il nome: quello scelto per questo DaemonSet, ossia `fluentd`, dovrà essere unico tra tutti gli altri DaemonSet presenti nel cluster, e questo proprio per la sua natura: i Pod che esso gestisce vengono distribuiti tra tutti i nodi e non potranno avere lo stesso nome.

Se creiamo il DaemonSet specificato tramite il comando `kubectl apply`, potremo utilizzare `kubectl describe` per monitorare lo stato dei Pod presenti sui diversi nodi: in questo caso, avremo 3 Pod attivi, in quanto i nodi applicativi sono 3 e non c'è stato alcun nodo che ha "rifiutato" l'esecuzione di queste risorse:

Listato 9.8 Descrizione del DaemonSet di Fluentd

```

kubectl describe daemonset fluentd
>>>
Name: fluentd
Image(s): fluent/fluentd-kubernetes-daemonset:v1-debian-elasticsearch
Selector: k8s-app=fluentd-logging
Node-Selector: <none>
Labels: k8s-app=fluentd-logging
Desired Number of Nodes Scheduled: 3
Current Number of Nodes Scheduled: 3
Number of Nodes Misscheduled: 0
Pods Status: 3 Running / 0 Waiting / 0 Succeeded / 0 Failed

```

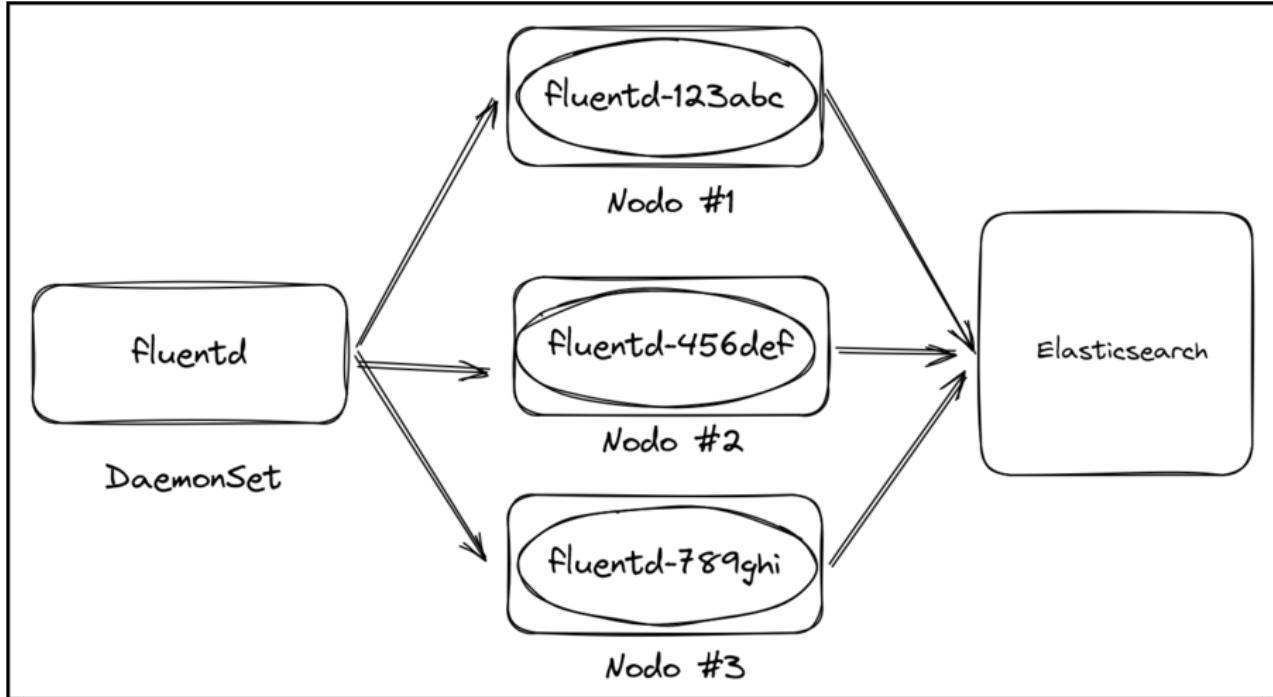


Figura 9.8 Come funziona il DaemonSet di Fluentd.

Così come avviene per le altre risorse viste finora che “gestiscono” dei Pod, a partire da Kubernetes 1.6, i DaemonSet possono essere aggiornati o modificati con le stesse strategie utilizzate dai Deployment o dalle altre risorse: di default, quando viene effettuata una modifica nella definizione del DaemonSet, tutti i Pod che esso gestisce vengono cancellati e ricreati da zero. Dal momento che abbiamo detto che il compito principale delle risorse di questo tipo è per le attività di monitoraggio, un disservizio potrebbe comportare ad un “buco” di informazioni: per questo, è possibile specificare all’interno delle proprietà del DaemonSet parametri come il numero massimo di Pod che possono essere simultaneamente aggiornati (proprietà `minReadySeconds`), così da avere margine di manovra con gli altri, o anche il tempo di attività di un Pod prima che si aggiornino anche gli altri (proprietà `maxUnavailable`). Quest’ultimo è indicato per impostare un valore ragionevolmente lungo, per esempio 30-60 secondi, di modo da assicurarsi che il Pod appena aggiornato sia in esecuzione correttamente prima che l’aggiornamento proceda.

Listato 9.9 Aggiornamento del DaemonSet con l’uso dei campi `maxUnavailable` e `minReadySeconds`

```

apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      minReadySeconds: 30
...

```

Per approfondire

Se vuoi studiare nel dettaglio il funzionamento di Fluentd per Kubernetes, puoi fare riferimento alla documentazione ufficiale presente sul sito: <https://docs.fluentd.org/container-deployment/kubernetes>.

Job

Finora negli esempi abbiamo parlato di applicazioni web o simili, ma non abbiamo mai dato spazio a flussi di lavoro che devono, magari, essere eseguiti in maniera estemporanea, compiendo una o più azioni per poi tornare nello stato iniziale. Se, per esempio, avessimo bisogno di aggiornare il valore di un campo presente in una delle risorse in esecuzione sul cluster, questa operazione potrebbe essere eseguita una tantum e sarebbe di breve durata. Utilizzare una delle risorse viste finora sarebbe probabilmente fuori dalla sua portata, mentre questo è il caso perfetto per un oggetto come un Job, creato appositamente per gestire questi tipi di attività. Un Job, infatti, crea Pod che vengono eseguiti fino alla loro conclusione, dove si intende un esito positivo se il codice di uscita del container è pari a 0. Questo tipo di risorse sono utili per quelle operazioni che vogliamo compiere solo una volta, come può avvenire nel caso di migrazioni di un database.

Il Job è dunque un'entità che è responsabile della creazione e della gestione dei Pod definiti all'interno del suo template, i quali sono eseguiti non appena il Job viene creato e rimangono in esecuzione fintanto che non vengono completati con successo (o escono con un errore). Questo tipo di oggetti permette anche di coordinarsi con diversi Pod, che possono eseguire più attività in parallelo: in questo modo, possiamo eseguire più operazioni contemporaneamente, per poi chiudere, e si basa fondamentalmente su due concetti: quelli di completamento e di parallelismo. Abbiamo già detto che un Job conclude positivamente quando viene portata a compimento l'attività per cui è stato creato, e questo vuol dire che possiamo decidere di eseguire la stessa operazione una volta, o anche N volte, a seconda del tipo di operazione: se vogliamo, come visto in precedenza, eseguire la migrazione di un database, ci aspettiamo che sia sufficiente un'esecuzione completata con successo per poter dichiarare il Job concluso, e quindi un *completamento* sarebbe abbastanza; nel caso di una coda di dati che devono essere gestiti, potrebbe essere utile avere più "cicli" di esecuzione dello stesso Job. Allo stesso modo, è possibile che, in un'attività di gestione di una coda di messaggi, sia utile eseguire in parallelo più Pod per portare a termine il processamento dei dati quanto prima, e quindi il *parallelismo* potrebbe essere maggiore di uno. Per fissare meglio questi due concetti, proviamo con un esempio pratico.

Listato 9.10 Esempio di definizione di Job

```
apiVersion: batch/v1
kind: Job
metadata:
  name: my-job
spec:
  completions: 4
  parallelism: 2
  template:
    metadata:
      name: my-job-001
  spec:
    containers:
      - name: c
        image: gcr.io/<project>/myjob
        env:
          - name: BROKER_URL
            value: amqp://guest:guest@rabbitmq-service:5672
          - name: QUEUE
            value: job1
    restartPolicy: OnFailure
```

In questo caso, stiamo utilizzando un Job per andare a gestire i messaggi presenti in una coda: il funzionamento di un tipo di struttura come questa è abbastanza facile, ed è il seguente. Ci sono diverse applicazioni che producono dei dati e c'è una coda dove questi messaggi vengono inseriti per poi essere gestiti: questa coda li invierà, utilizzando diversi tipi di ordine, a dei servizi preposti per la loro elaborazione. Mentre le applicazioni iniziali vengono solitamente definite *producer*, quelle finali vengono chiamate *consumer*: uno schema molto semplice di questo flusso di lavoro è riportato di seguito:

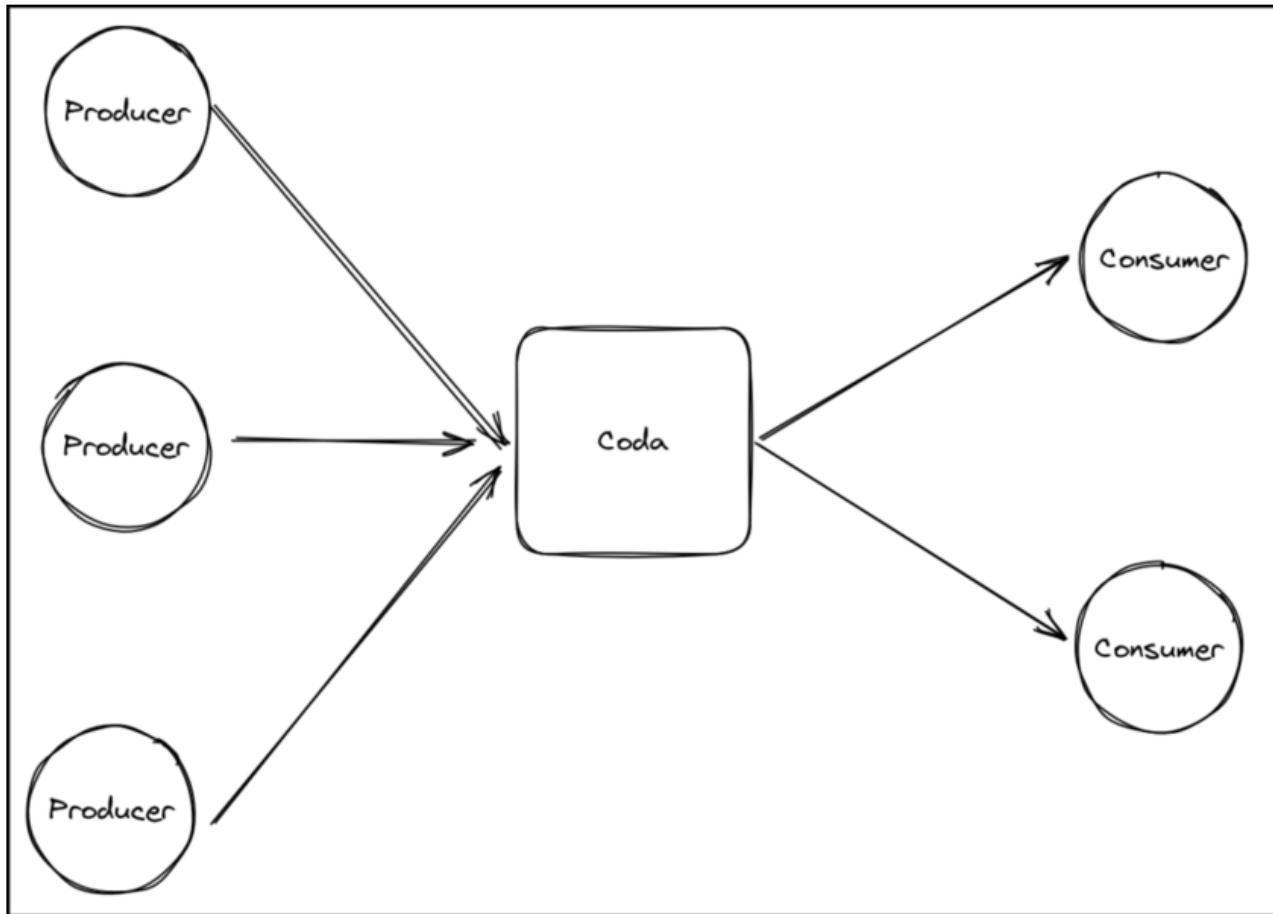


Figura 9.9 Rappresentazione di un servizio con producer e consumer che sfruttano una coda per scrivere e leggere dei messaggi.

Che cosa succede, quindi, nell'esempio mostrato in precedenza? Utilizzando RabbitMQ, un popolare broker di messaggi (per semplicità, un gestore di messaggi), quelli in arrivo vengono gestiti finché la capacità massima non è stata raggiunta (nel nostro caso 4), e parallelizzando le attività con 2 messaggi elaborati simultaneamente (quindi 2 Pod contemporanei in esecuzione).

Listato 9.11 Esempio di Pod di un Job

```
kubectl get pods
>>>
NAME          READY   STATUS    RESTARTS   AGE
job1-41h87    1/1    Running   0          5m
my-job-001-twjtc 1/1    Running   0          2m
my-job-001-915ai 1/1    Running   0          2m
```

Un Job come quello nell'esempio può fallire per qualsiasi motivo, tra cui un errore dell'applicazione, un'eccezione non rilevata durante la sua esecuzione o un errore del nodo su cui è stato avviato prima

che questo abbia la possibilità di essere completato. In tutti i casi, sarà compito del Job creare nuovamente il Pod fino a quando non si verifica il completamento dell'attività in maniera corretta.

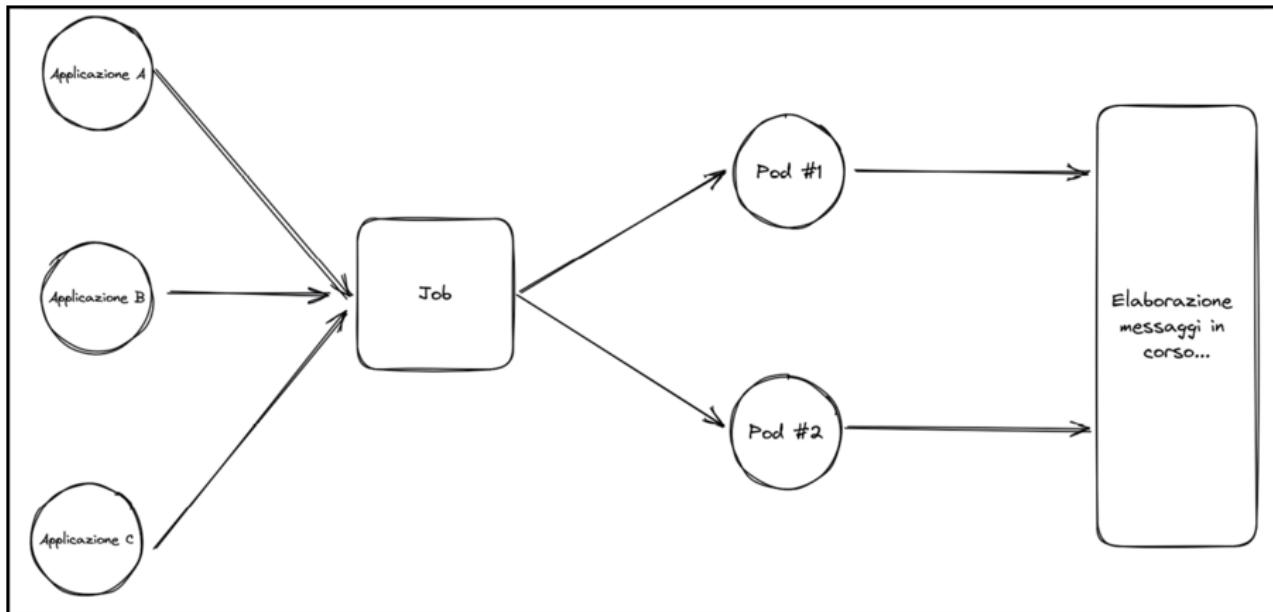


Figura 9.10 Come applicare un Job per ripetere l'esperimento.

Come visto anche per i DaemonSet, `kubelet` non si preoccuperà in alcun modo di ciò che avviene all'interno di questa risorsa, in quanto la gestione è demandata totalmente al Job: questo vuol dire che il Pod potrebbe andare in errore ed essere riavviato continuamente. Per questo, come abbiamo trovato in alcuni esempi precedenti, il comando `kubectl logs` potrebbe tornarci utile: se il Pod venisse riavviato a causa di un errore dell'applicazione, potrebbe utilizzarlo per visualizzare quanto prodotto dall'applicazione per cercare di risolvere.

Grazie alla possibilità di eseguire più attività in parallelo, i Job riescono a scalare in maniera dinamica: possiamo infatti utilizzare il comando `kubectl scale` per aumentare il numero di Pod che il Job può eseguire e velocizzare il completamento dell'attività sfruttando una proprietà comune a molti altri controller. Rispetto al caso di esempio precedente, questo vuol dire che ci saranno 3 Pod in esecuzione contemporaneamente e non più 2:

Listato 9.12 Scaling di un Job

```
kubectl scale job myjob --replicas 3
>>>
job "myjob" scaled
```

Vediamo un'ultima cosa sui Job: quanto tempo deve attendere il Job affinché la sua attività sia considerata “completata”? Si tratta di una domanda importante, soprattutto se all'interno del container c'è qualcosa che non va e rischiamo che il Pod si blocchi senza riuscire a portare a termine il suo compito in un tempo ragionevole. Il tempo di esecuzione può essere limitato impostando la proprietà `activeDeadlineSeconds` nelle specifiche del Pod: questo serve a specificare il tempo massimo di esecuzione del Pod, che verrà poi terminato con esito negativo (e quindi, con stato `Failed`) se il codice di uscita del container non sarà pari a 0. È possibile inoltre configurare quante volte un Job può essere avviato prima che venga contrassegnato come *fallito*, specificando il campo `backoffLimit`: questo indica il numero di tentativi da eseguire prima che il suo stato sia indicato in maniera definitiva come

`Failed` o `Completed`, in caso di esito positivo; se non viene specificato esplicitamente, il valore predefinito è 6.

CronJob

Per introdurre al meglio questa nuova risorsa, immaginiamo di farci questa domanda: è possibile indicare a Kubernetes delle operazioni che devono essere eseguite periodicamente? I Job, per loro natura, eseguono i loro Pod immediatamente quando avviene la loro creazione, ma molti processi devono essere eseguiti in un momento specifico nel futuro o anche ripetutamente nel corso del tempo. Nei sistemi operativi UNIX-like, questo tipo di attività sono meglio conosciute come `cron` e fanno proprio riferimento a un comando che definisce un lavoro che deve essere eseguito a un'ora specificata. Poteva Kubernetes non supportarli?

Un CronJob è infatti un oggetto che viene configurato grazie alla specifica del tipo di attività e del momento in cui l'attività dovrà essere eseguita, utilizzando il formato `cron`: se hai familiarità con i "normali" `cron`, non avrai nessuna difficoltà nel comprendere le potenzialità di questo oggetto. Difatti, all'ora configurata, Kubernetes creerà una risorsa Job in base a quanto presente nella sua definizione e, a seconda delle specifiche, uno o più Pod verranno creati e avviati. Non c'è niente di più.

Per questa ragione, vediamo subito un esempio: immagina di voler stampare un messaggio "I'm alive" a mezzanotte, tutti i giorni. Per farlo, puoi utilizzare la seguente definizione dove, oltre a specificare il tipo di oggetto (in questo caso CronJob), indichiamo la `schedule`, ossia la data in cui vogliamo che l'attività venga eseguita e poi, così come fatto per i Job, andiamo a scrivere qual è il container da utilizzare per produrre il messaggio di stampa.

Listato 9.13 Esempio di CronJob

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "0 0 * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox:1.28
              imagePullPolicy: IfNotPresent
              command:
                - /bin/sh
                - -c
                  - date; echo I'm alive
  restartPolicy: OnFailure
```

Vediamo alcuni punti importanti su questo esempio di CronJob: intanto, il campo `schedule` specifica che il processo deve essere eseguito una volta al giorno a mezzanotte, usando il formato `cron`. Per i CronJob senza fuso orario specificato, sarà `kube-controller-manager` a interpretare la data specificata relativamente al fuso orario locale di dove è installato. Il campo `containers`, come di consueto, specifica quale container deve essere eseguito dal Job, gestito dal CronJob: per mantenerci sul semplice, abbiamo utilizzato un'immagine `busybox`, al cui interno definiamo una serie di comandi `shell` da eseguire sul container, in questo caso stampando "I'm alive" sulla console. Infine, abbiamo la `restartPolicy`: questa può essere uguale a `Never` o `OnFailure`: nel caso di esempio, se l'operazione

non va a buon fine, vogliamo che il Job sia eseguito nuovamente; con l'opzione `Never`, non ci aspettiamo alcun riavvio in caso di errore.

Se non hai familiarità con il formato `cron`, esiste moltissima documentazione online che ti permette di approfondire a fondo il tema; vediamo comunque qualche esempio pratico che ci può tornare utile. Una *schedulazione* tramite `cron` è definita da una stringa composta da cinque parti che, lette da sinistra a destra, fanno riferimento a minuti, ore, giorni del mese, mese e giorno della settimana. Nell'esempio precedente, il primo 0 descrive i minuti, il secondo le ore, mentre gli asterischi rappresentano ogni giorno del mese, di ogni mese, per ogni giorno della settimana.

Listato 9.14 Esempio di CronJob con schedulazione ogni 15 minuti

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: batch-job
spec:
  schedule: "0,15,30,45 * * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          labels:
            app: periodic-batch-job
            spec:
              restartPolicy: OnFailure
        containers:
          - name: main
            image: myrepo/check-webserver
```

In quest'altro esempio, si desidera eseguire il processo descritto all'interno dell'immagine `check-webserver` ogni 15 minuti: per questo, viene indicato un elenco di minuti (ossia “0,15,30,45) seguito da una serie di asterischi, che significa che ogni ora, quando avremo 0 minuti, 15 minuti, e così via, il CronJob verrà avviato. Se, invece, volessi farlo funzionare ogni 30 minuti, ma solo il primo giorno del mese, dovresti impostare la programmazione a “0,30 * 1 * *”; se vuoi che funzioni alle 3 del mattino ogni domenica (sì, il primo giorno della settimana nello standard europeo è la domenica), dovresti impostarlo su “0 3 * * 0” (l'ultimo zero sta per domenica).

Listato 9.15 Esempio di CronJob con schedulazione ogni domenica alle 3 del mattino

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: batch-job
spec:
  schedule: "0 3 * * 0"
  jobTemplate:
    spec:
      template:
        metadata:
          labels:
            app: periodic-batch-job
            spec:
              restartPolicy: OnFailure
        containers:
          - name: main
            image: myrepo/check-webserver
```

Prima di chiudere questo capitolo, prendiamoci un attimo per analizzare il funzionamento del CronJob: quando questo viene avviato, viene creata una risorsa di tipo Job all'incirca all'ora pianificata; “all'incirca”, perché i CronJob potrebbero non essere “puntuali”. Questo può accadere per mancata

disponibilità delle risorse del cluster, e quindi potrebbe essere necessario specificare che il Job non deve essere avviato troppo oltre l'orario pianificato. Per questo, all'interno del CronJob è possibile specificare un campo, definito `startingDeadlineSeconds`, che serve a descrivere la scadenza oltre la quale non va più pianificata l'esecuzione del Job. Nell'esempio seguente, il Job dev'essere eseguito ogni 15 minuti, quindi dovrebbe esserlo alle 8.30; se per qualche ragione il CronJob non inizia entro le 10:30:15, il Job non verrà eseguito e verrà visualizzato come `Failed`.

Listato 9.16 Esempio di CronJob con deadline

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: batch-job
spec:
  schedule: "0 3 * * 0"
  startingDeadlineSeconds: 15
  jobTemplate:
    ...

```

In circostanze normali, un CronJob crea sempre un solo Job per ogni esecuzione configurata all'interno della sua pianificazione, ma può accadere che vengano creati due Job nello stesso tempo a causa di un ritardo nell'esecuzione del prima: per questa ragione, le attività eseguite all'interno del CronJob dovrebbero essere *idempotenti*; questo vuol dire che, se la loro esecuzione avvenisse più volte anziché una volta sola, questo non dovrebbe portare a risultati indesiderati. Se invece ci trovassimo nel caso in cui non venga creato nessun Job, nonostante la pianificazione, bisognerebbe assicurarsi che l'esecuzione successiva del Job esegua qualsiasi attività che avrebbe dovuto essere eseguita da quella mancata. Questo può avvenire per un motivo molto semplice: il CronJob controlla ogni 10 secondi la pianificazione: se il campo `startingDeadlineSeconds` è impostato su un valore inferiore a 10 secondi, il CronJob potrebbe non essere pianificato per tempo, e questo porterebbe a un'esecuzione mancata.

Che cosa abbiamo imparato

- Quali risorse abbiamo a disposizione per gestire delle necessità particolari, come possono essere la replicazione dei dati su più volumi oppure delle attività che devono essere eseguite *una tantum*.
- Che cosa sono gli StatefulSet, come lavorano e quando adottarli.
- Che cosa sono i DaemonSet, qual è il loro scopo e quando vengono scelti come controller.
- Che cos'è un Job, come funziona e quando utilizzarlo.
- Che cos'è un CronJob, qual è la differenza con il Job e quando scegliere di applicarlo.

Autenticazione e autorizzazione

La sfida più grande che la leadership femminile affronterà nell'industria tech sarà quella di avere la determinazione di rimanere nel settore per far carriera, nonostante le differenze di trattamento che sicuramente incontrerà.

– Mara Marzocchi, Founder di Codemotion

Tra le funzionalità presenti di default di Kubernetes, come abbiamo visto, non c'è un'interfaccia Web: è possibile configurare un cluster Kubernetes e utilizzare Docker Desktop, o Rancher, per gestire il proprio cluster, ma si tratta, tuttavia, di uno strumento che non viene installato insieme allo strumento stesso. Kubernetes fornisce una serie di *add-ons* che è possibile configurare in seguito all'avvio del cluster per avere delle funzionalità aggiuntive: tra queste, c'è la gestione dei log a livello di cluster, la configurazione del DNS e del monitoraggio. Inoltre, avere una dashboard di default che non richieda l'installazione di altri strumenti può essere molto comodo, anche per ciò che concerne la gestione degli utenti e dei ruoli. A proposito di questo, avrai capito che quando esegui il comando `kubectl apply` la request viene inviata al cluster e il server API è il primo componente dei nodi control plane a ricevere tale richiesta. Il server API analizza la richiesta ed eventualmente memorizza l'oggetto in etcd. O almeno, questo è il quadro generale. Le cose sono in realtà un po' più complicate di così, perché le API sono organizzate in una collezione di componenti richiamati uno dopo l'altro.

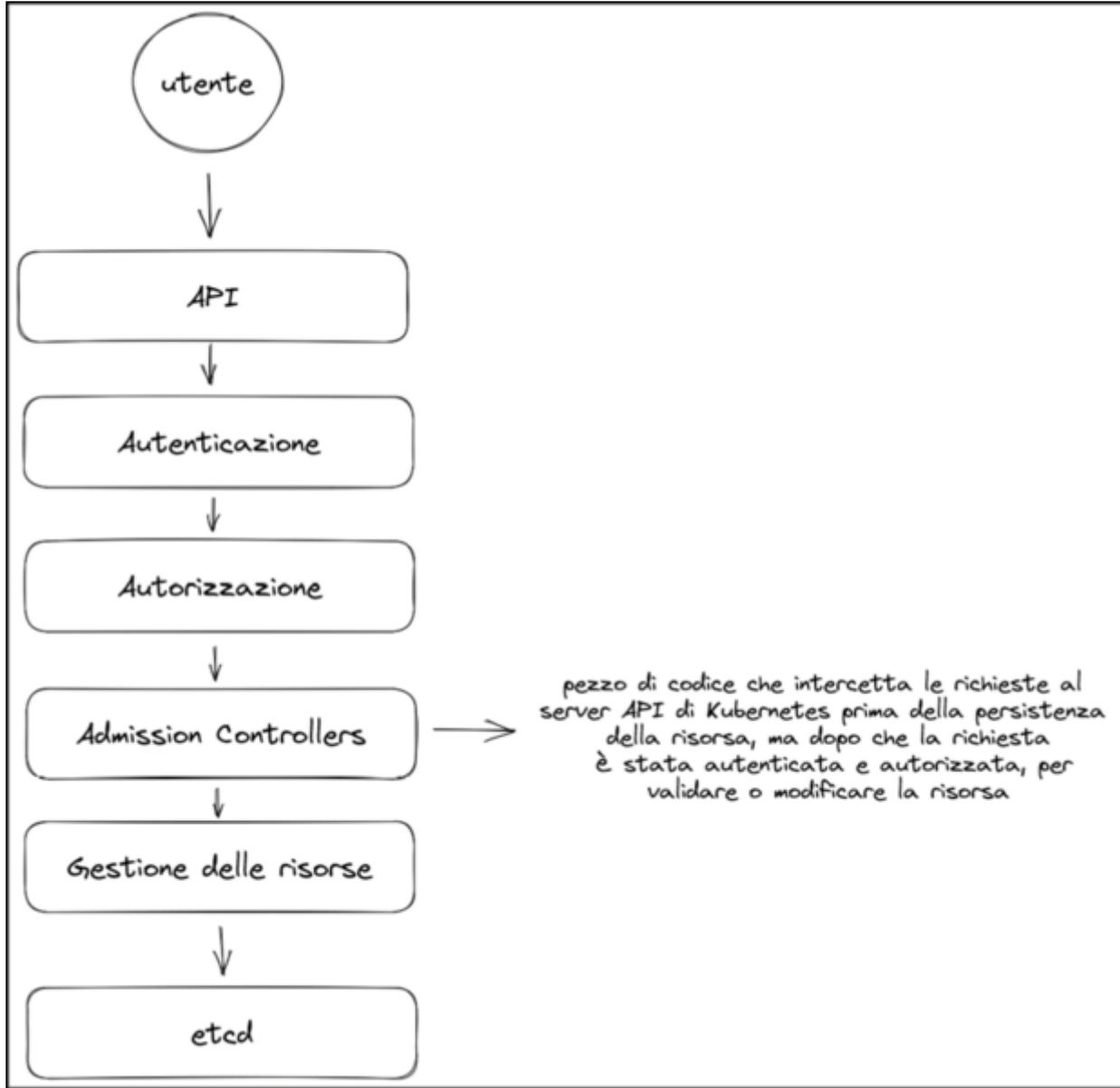


Figura 10.1 Componenti necessari per poter gestire degli utenti e i relativi ruoli all'interno di un'infrastruttura Kubernetes.

Uno dei componenti è responsabile in particolare dell'autenticazione delle richieste, e Kubernetes fa distinzione tra gli utenti interni ed esterni. Quelli interni gestiti da Kubernetes sono account creati dal cluster stesso e generalmente utilizzati dalle applicazioni intra-cluster, mentre quelli *non* gestiti da Kubernetes sono, per esempio, utenti autenticati tramite provider esterni come Keystone, Google e LDAP.

Come mostrato in Figura 10.1, autenticazione e autorizzazione sono due cose ben distinte: la prima serve a identificare e confermare ciò che l'utente dichiara, mentre l'autorizzazione concede l'accesso (o lo limita a seconda dei casi) a una o più risorse in base all'identità dell'utente. Ogni plugin che installiamo, focalizzandoci su quelli per la gestione dell'autenticazione, ha punti di forza e di debolezza e offre un meccanismo diverso per autenticare le identità. Questo si inserisce all'interno di quanto visto prima tra l'API server e lo strato di autenticazione: una volta che il meccanismo con cui si gestisce l'accesso degli utenti ha successo, la richiesta viene passata al modulo di autorizzazione. Se hai avuto successo durante questa prima fase, questo componente per l'autenticazione recupera i tuoi dettagli e

li impacchetta in un oggetto che servirà a contenere queste informazioni per il servizio di autorizzazione. Per quanto riguarda l'autorizzazione, Kubernetes implementa il modello di controllo degli accessi in base al ruolo (abbreviato in inglese in *RBAC*) per proteggere le risorse nel cluster. Agli utenti viene concesso l'accesso agli endpoint API e la lettura, l'aggiornamento e la creazione di risorse. Per gestire l'accesso all'API, Kubernetes raggruppa le autorizzazioni in risorse chiamate *Roles* e *ClusterRoles*, mentre la connessione tra tali identità (per esempio gli utenti, ma non solo) e i relativi permessi (*Roles* e *ClusterRoles*) viene definito utilizzando delle risorse chiamate *RoleBinding* o *ClusterRoleBinding*.

Mentre la configurazione di uno strumento di autenticazione dipende dalle logiche di business, quali sono gli oggetti che ci permettono di manipolare l'accesso al cluster è ciò che vedremo in questo capitolo, dando per scontato che la scelta del meccanismo di autenticazione sia una scelta del tutto individuale. Facciamo però prima un piccolo esperimento: finora, non ci siamo mai soffermati/sull'identità che assumiamo quando lavoriamo all'interno del cluster. Quindi, "chi siamo" noi per il cluster? Per procedere con questo esperimento, apriamo il terminale e proviamo a scoprirla. Per prima cosa, è necessario individuare l'indirizzo delle API del server Kubernetes, e per farlo possiamo utilizzare `kubectl`, in questo modo:

Listato 10.1 Configurazione di kubectl

```
kubectl config view
>>>
apiVersion: v1
clusters:
...
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://kubernetes.docker.internal:6443
    name: docker-desktop
...
...
```

Il comando appena eseguito ci permette di modificare o gestire ciò che è scritto nel file `kubeconfig`, file che detiene tutta la definizione del contenuto del nostro cluster a livello di configurazione: nell'output ci viene infatti riportato l'indirizzo dell'API (in questo caso) per il cluster ospitato su Docker Desktop. A seconda dell'installazione adottata, questo indirizzo potrebbe cambiare, ma di certo non cambia il risultato: vengono elencati tutti i cluster a cui ci siamo connessi, e il riferimento alle API. Proviamo allora a utilizzare questo indirizzo per ottenere l'elenco dei namespaces, e quindi proviamo a sottrarci a ciò che `kubectl` fa per noi, interrogando direttamente le API, in questo modo:

Listato 10.2 Elenco del namespace tramite API

```
export API_SERVER_URL=https://kubernetes.docker.internal:6443

curl $API_SERVER_URL/api/v1/namespaces
>>>
curl: (60) SSL certificate problem: unable to get local issuer certificate
More details here: https://curl.haxx.se/docs/sslcerts.html
curl failed to verify the legitimacy of the server and therefore could not
establish a secure connection to it. To learn more about this situation and
how to fix it, please visit the web page mentioned above.
```

L'output suggerisce che l'API sta gestendo il traffico attraverso il protocollo HTTPS con un certificato non riconosciuto, quindi `curl` ha interrotto la richiesta. Ignoriamo temporaneamente la verifica del certificato usando il parametro `-k` ed eseguiamo nuovamente il comando:

Listato 10.3 Elenco del namespace tramite API senza SSL

```

curl -k $API_SERVER_URL/api/v1/namespaces
>>>
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {},
  "status": "Failure",
  "message": "namespaces is forbidden: User \\"system:anonymous\\" cannot list resource
\\\"namespaces\\\" in API group \\\"\\\" at the cluster scope",
  "reason": "Forbidden",
  "details": {
    "kind": "namespaces"
  },
  "code": 403

```

Otteniamo finalmente una risposta dal server, ma leggiamo che ci è proibito accedere all'endpoint richiesto (con il codice di stato 403) e che la nostra identità corrisponde a `system:anonymous`, anche se questa identità non è autorizzata a elencare namespace. Il test precedente rivela alcuni importanti meccanismi di funzionamento nel `kube-apiserver`: in primo luogo, identifica l'utente di una richiesta (il corrispondente del comando `whoami` nei sistemi Unix-like), e poi determina quali operazioni sono consentite per questo utente (quali autorizzazioni hai). Se proviamo a rivedere cosa è successo nella precedente esecuzione di `curl`, possiamo vedere quanto ottenuto da un'altra prospettiva: non avendo fornito le credenziali di alcun utente, lo strato di Kubernetes che si occupa dell'autenticazione non è stato in grado di assegnarci un'identità, e quindi ha etichettato la richiesta come anonima. Lo strato di autorizzazione di Kubernetes ha verificato se `system:anonymous` disponeesse dell'autorizzazione per elencare i namespace nel cluster e, poiché non ha i permessi per farlo, restituisce un messaggio di errore `403 Forbidden`. Di default, infatti, le richieste fatte al cluster non vengono rifiutate, ma vengono gestite come utenti anonimi, che fanno parte del gruppo `system:unauthenticated`. Se avessimo voluto, invece, “costringere” Kubernetes a richiedere un'identità precisa per lavorare con il cluster, avremmo potuto avviare `kubelet` con l'opzione `--anonymous-auth=false`: in questo modo un test come quello eseguito in precedenza riporterà una risposta `401, ossia non autorizzata`.

Utenti

Il modulo di autenticazione è il primo filtro per il cluster che serve ad autenticare tutte le richieste utilizzando un token statico, un certificato o un'identità gestita esternamente. Kubernetes dispone di un modulo di autenticazione con diverse caratteristiche: è in grado di supportare sia gli utenti esterni (per esempio le app distribuite al di fuori del cluster), sia quelli che fruiscono del programma in esecuzione, ma anche gli utenti interni (per esempio gli account creati e gestiti da Kubernetes). Abbiamo detto che supporta strategie di autenticazione standard, come un token statico, un certificato formato X509 e così via, con la possibilità di disporre anche di più strategie di autenticazione contemporaneamente.

Le possibilità sono diverse, ma possiamo tuttavia classificare gli utenti nei seguenti tipi:

- Utenti gestiti Kubernetes: account utente creati dal cluster Kubernetes stesso e utilizzati dalle applicazioni presenti nel cluster.
- Utenti non gestiti da Kubernetes: utenti esterni al cluster Kubernetes, per esempio:
 - utenti che possiedono un token o dei certificati statici forniti dagli amministratori del cluster;
 - utenti autenticati tramite provider di identità esterni come Keystone, account Google e LDAP.

Per quelli esterni, come abbiamo detto, è necessario che le API possano in qualche modo ottenere le informazioni circa l'utente e la sua identità, e che possano quindi autenticarlo: un modo molto semplice per farlo potrebbe essere quello di fornire dei token associati a un certificato a ogni utente, e

modificare il file `kubeconfig` di modo che, al posto di quanto visto in precedenza con il comando `kubectl config view`, questo certificato venga utilizzato insieme al server che fornisce l'identità degli utenti e sia a disposizione delle API del cluster per poter confermare l'identità delle utenze che tentano di usare le risorse del cluster.

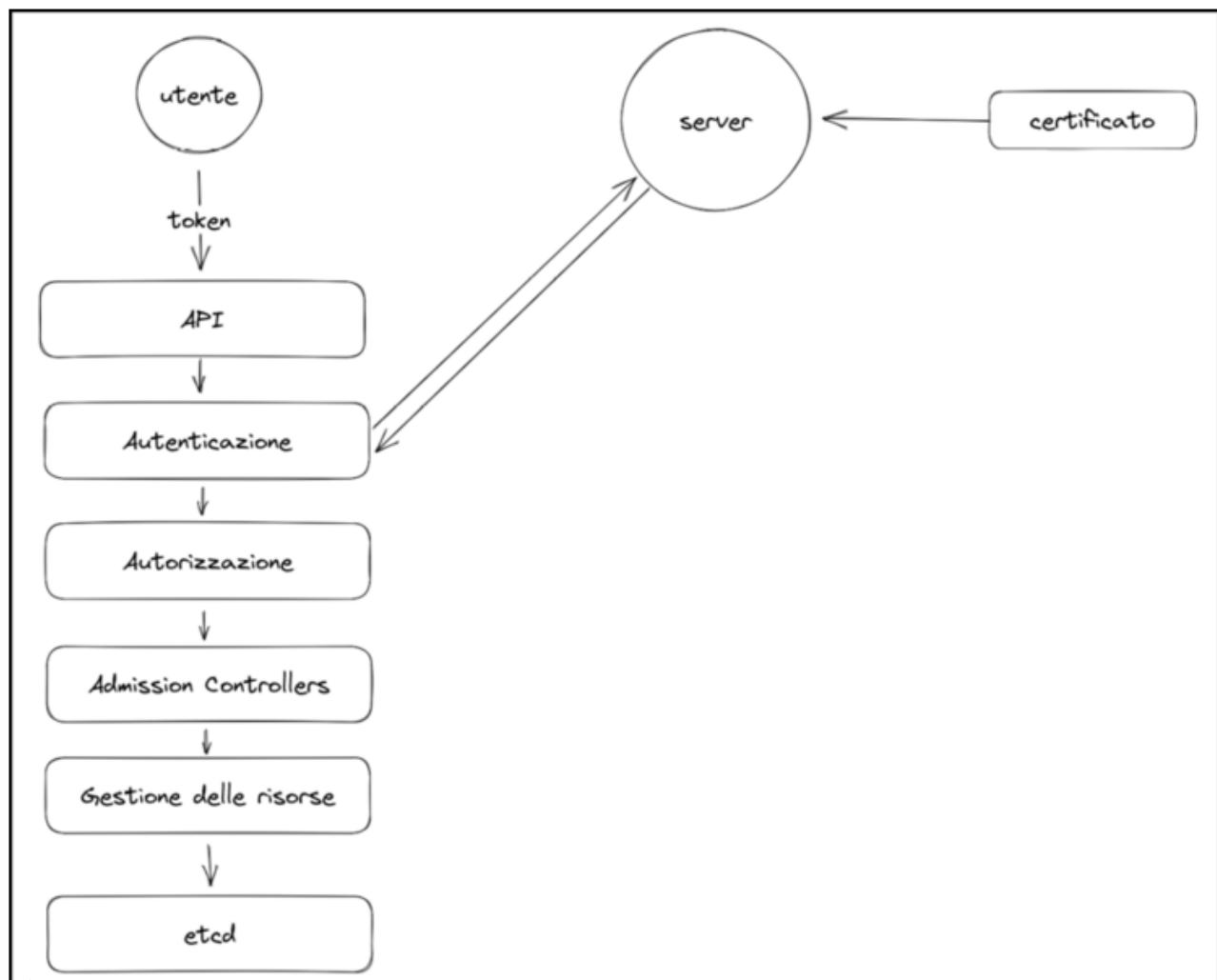


Figura 10.2 Come gestire i certificati con un server all'interno del flusso.

In Kubernetes, gli utenti interni vengono invece assegnati a delle identità denominate *ServiceAccount*, o *account di servizio*. Tali identità vengono create da `kube-apiserver` e assegnate alle applicazioni. Infatti, quando un'applicazione interna al cluster effettua una richiesta, il server API può verificarne l'identità condividendo un token firmato collegato al suo *ServiceAccount*. Questo significa che finora, per tutte le applicazioni che abbiamo avviato all'interno del cluster, qualcuno ha creato un account al posto nostro e l'ha assegnato al Pod (o più genericamente, a ogni risorsa) con i relativi permessi. Facciamo, anche in questo caso, un piccolo esperimento: utilizziamo il comando `kubectl describe` su uno qualsiasi dei Pod che abbiamo a disposizione, e proviamo a leggerne l'output:

Listato 10.4 Output della descrizione di un Pod

```

kubectl describe pod nginx-deployment-cd55c47f5-2mssb
>>>
Name:           nginx-deployment-cd55c47f5-2mssb
Namespace:      default
Priority:      0
  
```

```

Service Account: default
...
Controlled By: ReplicaSet/nginx-deployment-cd55c47f5
...

```

Notiamo che il Pod ha un ServiceAccount assegnato, che si chiama proprio `default`: questo serve a fornire un'identità ai processi in esecuzione all'interno di un Pod. Possiamo elencare e descrivere tutti quelli presenti tramite il comando `kubectl get sa`, dove `sa` è l'abbreviazione del nome della risorsa:

Listato 10.5 Elenco dei ServiceAccount del namespace corrente

```

kubectl get sa
>>>
NAME      SECRETS   AGE
default   0          19d

kubectl describe sa default
>>>
Name:           default
Namespace:     default
Labels:         <none>
Annotations:   <none>
Image pull secrets: <none>
Mountable secrets: <none>
Tokens:        <none>
Events:        <none>

```

Come vediamo nell'output, questa risorsa non ha alcun tipo di oggetto al suo interno: come fa quindi ad autenticarsi all'interno del cluster, dal momento che non c'è nessun token o chiave associata? Esaminiamo nuovamente la definizione del Pod e noteremo diverse cose che potrebbero attirare la nostra attenzione.

Listato 10.6 Descrizione del Pod

```

kubectl get pod nginx-deployment-cd55c47f5-2mssb -o yaml
>>>
apiVersion: v1
kind: Pod
...
spec:
  containers:
    - image: nginx:latest
      imagePullPolicy: Always
      name: nginx
      ports:
        - containerPort: 80
          protocol: TCP
      resources: {}
      terminationMessagePath: /dev/termination-log
      terminationMessagePolicy: File
  volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: kube-api-access-96vbf
      readOnly: true
...
  serviceAccount: default
  serviceAccountName: default
  terminationGracePeriodSeconds: 30
...
  volumes:
    - name: kube-api-access-96vbf
      projected:
        defaultMode: 420
  sources:

```

```

- serviceAccountToken:
    expirationSeconds: 3607
    path: token
- configMap:
    items:
    - key: ca.crt
      path: ca.crt
      name: kube-root-ca.crt
- downwardAPI:
    items:
    - fieldRef:
      apiVersion: v1
      fieldPath: metadata.namespace
      path: namespace
...

```

Esaminiamo il tutto con calma: partendo dalla fine, notiamo che c'è un volume chiamato `kube-api-access-...` montato all'interno del Pod, il quale fa riferimento a un percorso `/var/run/secrets/kubernetes.io/serviceaccount` accessibile in sola lettura.

Projected volume: che cosa significa

Un volume proiettato è un volume che combina diversi volumi esistenti in uno solo. Questo vuol dire, in altre parole, che è possibile mappare diverse sorgenti esistenti nella stessa directory, come se diverse applicazioni condividessero (tramite una rete, per esempio) la stessa cartella.

In questo caso particolare, il volume è una combinazione di un volume chiamato `serviceAccountToken` montato su di un path chiamato `token`, uno definito come `configMap` e infine un altro il cui nome è `downwardAPI`. Il volume `serviceAccountToken` è un volume speciale che monta un Secret associato al ServiceAccount utilizzato dal Pod (nel nostro caso `default`). Questo viene utilizzato al solo scopo di popolare il file `/var/run/secrets/kubernetes.io/serviceaccount/token` con il token corretto, per permettere al Pod di comunicare con altre risorse all'interno del cluster o, più genericamente, di vivere l'ecosistema del cluster attraverso le API del server Kubernetes. Il volume chiamato `configMap` è un volume che monta i certificati interni al cluster come file nella directory corrente, come il file `ca.crt`, necessario per chiamare l'API Kubernetes. Infine, il volume `downwardAPI` è un volume speciale che utilizza l'API `Downward` per esporre le informazioni sul Pod ai suoi container. A volte è infatti utile che un container disponga delle informazioni su se stesso, senza essere legato alla definizione fornita da Kubernetes. Questa particolare API (che non ha nulla a che fare con il server API di Kubernetes, sia chiaro) consente ai container di ottenere informazioni su se stessi o sul cluster senza utilizzare il client Kubernetes o il server API. Alcune di queste informazioni sono, per esempio, il proprio nome, il namespace dove risiedono, le risorse in termini di CPU o memoria definite, oppure proprio il ServiceAccount.

Downward API

Per approfondire e vedere quali sono i campi messi a disposizione da questa API, puoi consultare la documentazione ufficiale: <https://kubernetes.io/docs/concepts/workloads/pods/downward-api/>.

L'aspetto importante è il seguente: ogni ServiceAccount creato o disponibile appartiene a uno specifico namespace: questo vuol dire che ci aspettiamo esista un account di questo tipo per ogni namespace presente nel cluster (tranne quelli di sistema, come `kube-system`).

Che cosa succede, quindi, se voglio creare un account interno al cluster che abbia, per esempio, la possibilità di vedere quali Pod sono in esecuzione? Prima di seguire questo passaggio, parliamo di ruoli e *binding*.

Ruoli

Man mano che il numero di applicazioni e di componenti aumenta nel tuo cluster, potresti voler rivedere e limitare le azioni che le persone che usano il cluster possono intraprendere. Per esempio, potresti voler limitare l'accesso ai sistemi di produzione a un numero ristretto di persone, oppure potresti voler concedere un insieme ristretto di autorizzazioni a un utente nel cluster.

Il framework RBAC (abbreviazione di *Role-Based Access Control*) in Kubernetes ti consente di fare proprio questo: tramite il controllo degli accessi in base al ruolo, è possibile assegnare autorizzazioni granulari e limitare ciò che un utente o un'applicazione può fare. In termini più pratici, quanto visto in precedenza nello schema di inizio capitolo, quando il server API riceve una richiesta, deve prima autenticare l'utente e verificare l'accesso a una determinata risorsa: se questa ha esito positivo, l'utente può continuare a operare, altrimenti otteniamo (come visto) un errore 403 Forbidden. RBAC è infatti un modello molto diffuso e progettato per concedere l'accesso alle risorse in base ai ruoli dei singoli utenti all'interno di un'organizzazione. Per capire nel dettaglio come funziona, facciamo un passo indietro e immaginiamo di dover progettare un sistema di autorizzazione da zero. Come possiamo assicurarci che un utente abbia accesso a una particolare risorsa per poterla, per esempio, modificare? Un semplice schema che rappresenti questa situazione potrebbe portare alla creazione di una tabella come la seguente:

Listato 10.7 Elenco di utenti e risorse

Utente	Risorsa	Permessi
Andrea	applicazione A	lettura
Aurora	applicazione B	lettura
Virginia	applicazione A	lettura/scrittura

In questo esempio: Andrea ha accesso in lettura (quindi niente modifiche) all'applicazione A, ma non alla B; Aurora può leggere l'applicazione B, ma non la A, e infine Virginia può accedere all'applicazione A sia in lettura che in scrittura. Che cosa succede se immaginiamo Andrea e Virginia partono dello stesso team e vogliamo che abbiano i permessi per lavorare insieme all'applicazione C? Con lo schema precedente, dovremmo aggiungere due righe per ogni utente e per il relativo permesso:

Listato 10.8 Elenco di utenti e risorse – aggiornato per il team

Utente	Risorsa	Permessi
Andrea	applicazione A	lettura
Aurora	applicazione B	lettura
Virginia	applicazione A	lettura/scrittura
Andrea	applicazione C	lettura/scrittura
Virginia	applicazione C	lettura/scrittura

Al momento stiamo lavorando con appena 3 utenti e 3 applicazioni, ma le righe già si stanno moltiplicando: immagina di dover lavorare anche con i team. Una strategia per risolvere il problema potrebbe essere quella di aggiungere una colonna `Team` alla tabella, ma un'alternativa migliore è lavorare con il concetto di ruolo: questo ci permette di definire un contenitore generico per i permessi, e quindi di assegnare delle autorizzazioni agli utenti attraverso queste risorse *comuni*. Potremo quindi modificare lo schema precedente come nel Listato 10.9.

Listato 10.9 Elenco di utenti e ruoli

Ruolo	Permesso	Risorsa
adminC	lettura/scrittura	applicazione C
lettura	lettura	applicazione B

...

Utente Ruolo
----- -----
Aurora lettura
Andrea adminC
Virginia adminC

Che cosa succede, invece, se voglio far lavorare Virginia con l'applicazione A? Possiamo attribuirle un altro ruolo.

Listato 10.10 Elenco di utenti e ruoli

Ruolo Permesso Risorsa
----- ----- -----
adminC lettura/scrittura applicazione C
adminA lettura/scrittura applicazione A
lettura lettura applicazione B
...
Utente Ruolo
----- -----
Aurora lettura
Andrea adminC
Virginia adminC, adminA

Kubernetes utilizza gli stessi tre concetti spiegati in precedenza: identità e ruoli, con le relative **associazioni** alle risorse, solo che li chiama con nomi leggermente diversi. Per esempio, esaminiamo la seguente definizione YAML necessaria per concedere l'accesso a un Pod. Il file è diviso in tre blocchi, dove il primo definisce un ServiceAccount chiamato `serviceaccount:api` e corrisponde all'identità di chi accede alle risorse; il secondo descrive un ruolo, chiamato `role:viewer`, e le relative regole che lo autorizzano ad accedere a determinate risorse. Il terzo corrisponde al RoleBinding, ossia l'oggetto che collega l'identità (`Service Account`) alle autorizzazioni (`Role`).

Listato 10.11 Esempio di definizione di un ServiceAccount, Role e RoleBinding

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: serviceaccount:appl
  namespace: demo
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: role:viewer
  namespace: demo
rules:
  - apiGroups:
    - ''
      resources:
        - services
        - pods
      verbs:
        - get
        - list
  - apiGroups:
    - apiextensions.k8s.io
      resources:
        - customresourcedefinitions
      verbs:
        - list
---
apiVersion: rbac.authorization.k8s.io/v1
```

```

kind: RoleBinding
metadata:
  name: rolebinding:app1-viewer
  namespace: demo
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: role:viewer
subjects:
  - kind: ServiceAccount
    name: serviceaccount:app1
    namespace: demo

```

Mentre il ServiceAccount al momento risulta abbastanza semplice da comprendere, concentriamoci sul ruolo: in Kubernetes abbiamo bisogno di poter specificare quali sono le risorse che un utente (termine che useremo al momento per riferirci a un ServiceAccount) può utilizzare, e in che modo. All'interno di un `Role` andremo quindi a definire queste informazioni, seguendo questo schema: partiamo dalle risorse, e noteremo che all'interno del ruolo abbiamo riportato i nomi di riferimento di alcune risorse. Questo è infatti il modo con cui Kubernetes è in grado di riconoscere quali sono le entità con cui un utente potrà (o non) lavorare.

Listato 10.12 Definizione delle risorse di un ruolo

```

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: role:viewer
  namespace: demo
rules:
  - apiGroups:
    - ''
      resources:
        - services
        - pods

```

Per i permessi, la storia è diversa: abbiamo a disposizione alcuni verbi che descrivono le azioni che quel ruolo può compiere sulle risorse; questi sono `get`, `list`, `create`, `patch`, `delete` e così via.

Listato 10.13 Definizione dei verbi di un ruolo

```

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: role:viewer
  namespace: demo
rules:
  - apiGroups:
    - ''
      resources:
        - services
        - pods
      verbs:
        - get
        - list

```

Infine, abbiamo le API: dal momento che in Kubernetes esiste la possibilità di utilizzare risorse custom, magari installate sfruttando dei progetti esterni, come fa Kubernetes a distinguere quali definizioni utilizzare? All'interno del campo `apiGroups` è possibile indicare infatti a quali fare riferimento: nel caso di esempio, viene specificata una stringa vuota, che sta a indicare le API core di Kubernetes.

Listato 10.14 Definizione delle API di un ruolo

```

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: role:viewer
  namespace: demo
rules:
  - apiGroups:
    - ''
      resources:
        - services
        - pods
      verbs:
        - get
        - list

# in alternativa

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: role:viewer
  namespace: demo
rules:
  - apiGroups:
    - ''
      resources: ['services', 'pods']
      verbs: ['get', 'list']

```

Come mostrato nell'esempio, possiamo anche ridurre le proprietà di `rules` sfruttando gli array: questa notazione riduce significativamente il numero di righe ed è più concisa. Se avessimo utilizzato delle risorse personalizzate, che magari hanno delle entità proprie i cui nomi coincidono con alcune già presenti, non avremmo avuto alcun tipo di problema.

Listato 10.15 Definizione delle API custom di un ruolo

```

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: role:viewer
  namespace: demo
rules:
  - apiGroups:
    - 'myresource.io'
      resources:
        - apps
        - instances
      verbs:
        - get
        - list

```

Ora che il ruolo è pronto, possiamo associarlo a un'utenza: per questo, è necessario servirsi del `RoleBinding`. Un `RoleBinding` concede le autorizzazioni definite in un ruolo a un utente, un account di servizio o un gruppo, e questo vuol dire che potrà essere associato anche a utenti esterni o gruppi definiti in un secondo momento.

Listato 10.16 Esempio di definizione di un ServiceAccount, Role e RoleBinding

```

apiVersion: v1
kind: ServiceAccount
metadata:
  name: serviceaccount:app1
...
---
apiVersion: rbac.authorization.k8s.io/v1

```

```

kind: Role
metadata:
  name: role:viewer
...
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: rolebinding:app1-viewer
  namespace: demo
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: role:viewer
subjects:
  - kind: ServiceAccount
    name: serviceaccount:app1
    namespace: demo

```

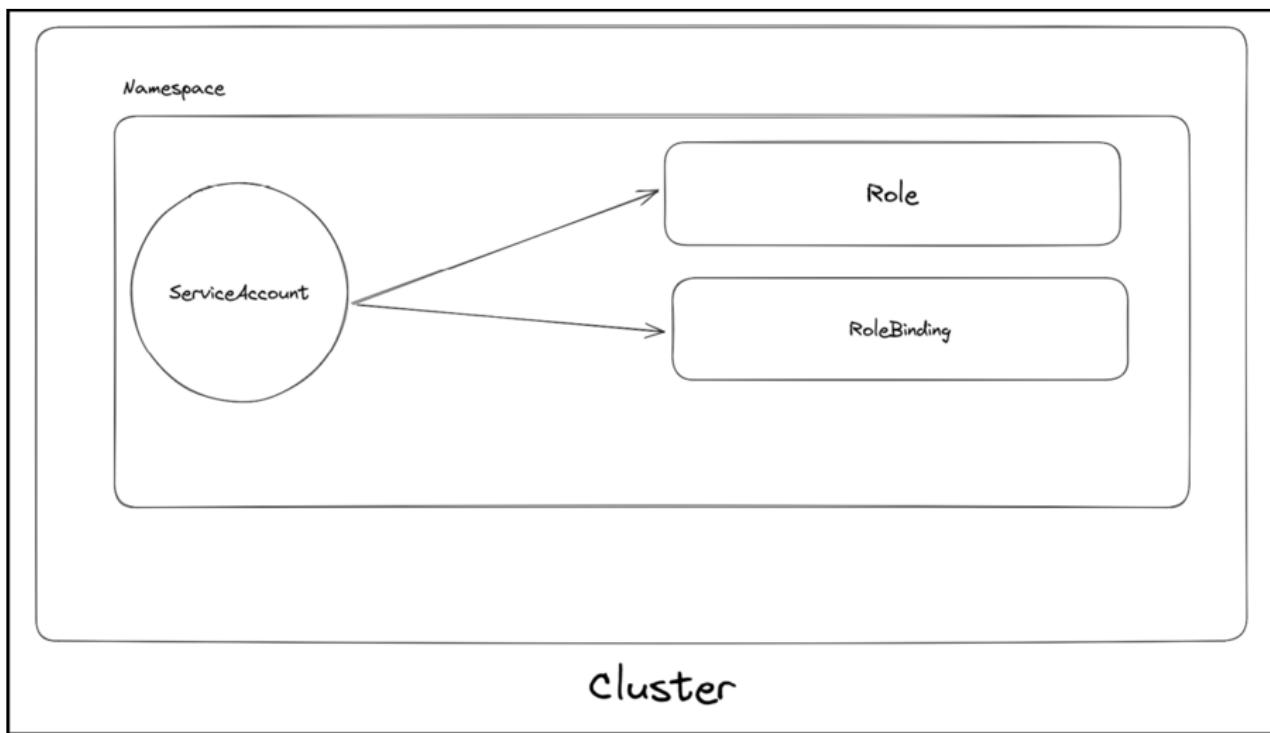


Figura 10.3 Come si relaziona un ServiceAccount a un Role e un RoleBinding, in maniera semplificata.

La definizione del RoleBinding ha due campi importanti: il `roleRef` fa riferimento al ruolo, mentre `subjects` riporta quali sono le entità a cui applicare questo ruolo. Non appena creiamo questa risorsa nel cluster, l'applicazione o l'utente che utilizza il ServiceAccount `serviceaccount:app1` avrà accesso alle risorse elencate nel ruolo. Se cancellassimo questo RoleBinding, l'applicazione o l'utente perderebbero l'accesso a tali risorse, pur rimanendo il ruolo, pronto per essere utilizzato da altre risorse. Notiamo anche che il campo `subject` ha un elenco di oggetti che contengono `kind`, `name` e `namespace`. Il tipo (campo `kind`) è necessario per poter distinguere ServiceAccount da utenti e gruppi, così come il nome: ma il campo `namespace`? Abbiamo detto quanto sia utile suddividere il cluster in namespace e limitare l'accesso alle risorse sfruttando queste entità per account specifici. Nella maggior parte dei casi, Role e RoleBinding vengono allocati all'interno di uno specifico namespace per concedere l'accesso a un insieme di risorse ben definito. Dal momento che i ServiceAccount sono

legati a un namespace, è necessario specificarlo all'interno di questo campo. Ci sono però delle risorse che fanno riferimento a dei namespace e che possono essere raggiunte da qualsiasi luogo, e che rappresentano degli oggetti fondamentali per il cluster, per esempio i nodi o i PV: dobbiamo quindi introdurre il concetto di *ClusterRole* e *ClusterRoleBinding*.

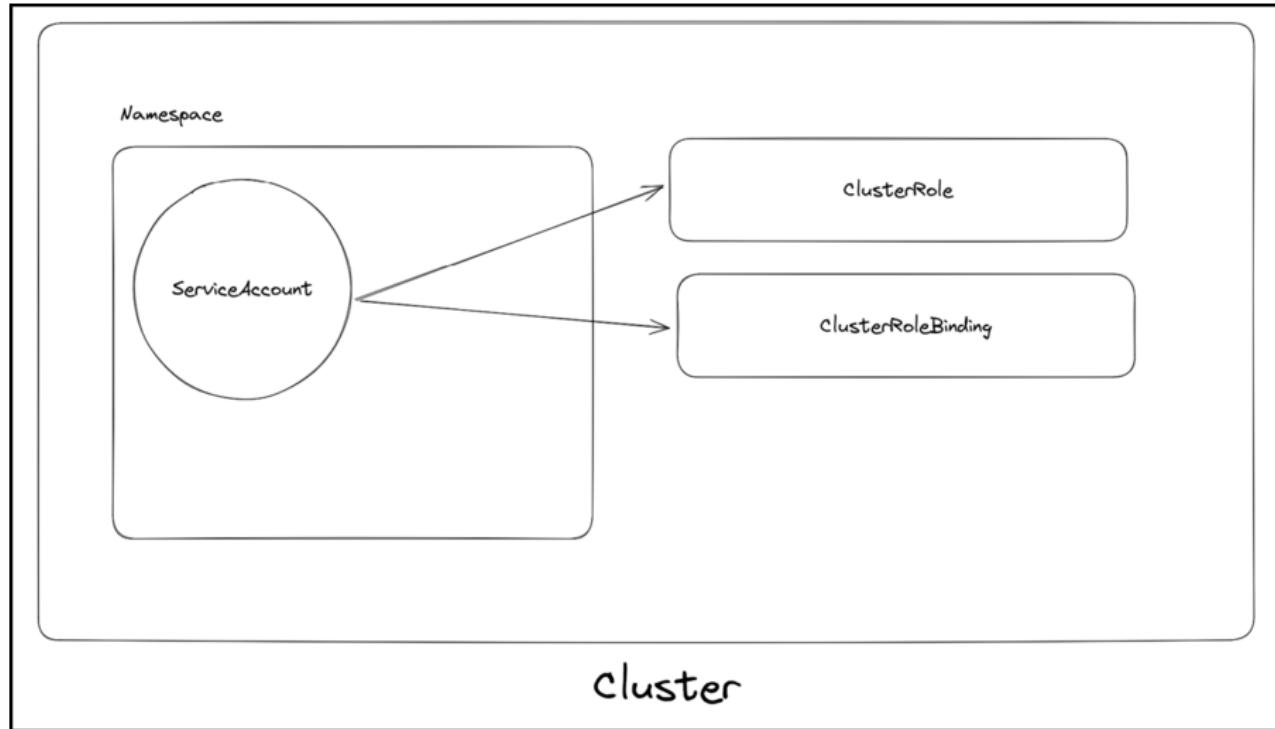


Figura 10.4 Come si relaziona un ServiceAccount a un ClusterRole e un ClusterRoleBinding, in maniera semplificata.

Abbiamo detto che i volumi e i nodi sono risorse che lavorano nell'ambito del cluster; tuttavia, i ruoli possono concedere l'accesso alle risorse che fanno capo a uno specifico namespace. Questo vuol dire che, se volessimo rendere il nostro ruolo adatto a lavorare con risorse di questo tipo, dovremo utilizzare la risorsa ClusterRole al posto di Role: per fortuna, non ci sono grossi cambiamenti lato definizione da apportare, se non nella tipologia dell'oggetto e delle risorse che intendiamo utilizzare. Per esempio, un ruolo che permetta la visualizzazione dei nodi presenti nel cluster potrebbe essere la seguente.

Listato 10.17 Esempio di definizione di un ClusterRole

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: node-viewer
rules:
  - apiGroups:
    - ''
      resources:
        - nodes
      verbs:
        - get
        - list
        - watch
```

Questo tipo di risorsa non è limitata alle sole che hanno visibilità su tutto il cluster, ma ne permette la definizione: questo vuol dire che all'interno della stessa definizione potremmo aggiungere anche risorse come Pod o Service, che invece fanno riferimento a uno specifico namespace.

A questo punto, diventa semplice comprendere che parte interpreta il ClusterRoleBinding: questo ci permette di associare a un'utenza un ClusterRole.

Listato 10.18 Esempio di definizione di un ClusterRoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: node-viewer-binding
subjects:
- kind: ServiceAccount
  name: test
  namespace: demo
roleRef:
  kind: ClusterRole
  name: node-viewer
  apiGroup: rbac.authorization.k8s.io
```

Come verificare se un'utenza (genericamente parlando) ha i permessi per eseguire delle operazioni? Sfruttando il comando `kubectl auth can-i`, abbiamo un'istruzione definita `can-i` che ci permette di ottenere una risposta booleana rispetto a un verbo, una risorsa e un utente che può (o non) eseguire delle operazioni: nel primo esempio, chiediamo tramite `kubectl` di verificare se è possibile ottenere l'elenco dei Pod (verbo `get` e risorsa `pods`) come ServiceAccount all'interno del namespace `demo` chiamato `serviceaccount:app1`. Nel secondo facciamo lo stesso test, ma su un namespace diverso: la risposta è negativa, perché quel ServiceAccount esiste solo all'interno del namespace `demo`, come definito negli esempi precedenti.

Listato 10.19 Esempio di utilizzo di can-i

```
kubectl auth can-i get pods --as=system:serviceaccount:demo:serviceaccount:app1
>>>
yes
kubectl auth can-i get pods --as=system:serviceaccount:test:serviceaccount:app1
>>>
no
```

Da questi esempi è possibile osservare alcuni comportamenti e limitazioni delle risorse RBAC: i Role e RoleBinding devono esistere nello stesso namespace, e i RoleBinding possono fare riferimento solo alle risorse Role. I RoleBinding possono agganciare un ClusterRole, ma la loro visibilità rimarrà legata alle risorse inerenti un namespace.

Che cosa abbiamo imparato

- Come funziona la gestione dell'autenticazione e dell'autorizzazione in generale, con un focus sul contesto Kubernetes.
- Quali risorse abbiamo a disposizione, come utenti e ServiceAccount, e come questi siano differenti.
- Quali sono le modalità con cui possiamo assegnare dei permessi per accedere alle risorse all'interno del cluster e permettere di compiere determinate azioni, attraverso il concetto di Role e RoleBinding.

Templating

L'industria tecnologica vuole davvero credere di essere una meritocrazia, quando non lo è. Dobbiamo venire a patti con i bias inconsci e riconoscere che molti degli assunti che facciamo sono, senza saperlo, influenzati dai nostri pregiudizi.

– Rebecca Parsons, CTO @ ThoughtWorks

Kubernetes non fornisce alcun meccanismo di creazione dei file YAML *da solo*: questi sono normalmente scritti dagli utenti che lo utilizzano e non permettono molta logica, di per sé, per adattare le risorse a determinati flussi di lavoro, ma sono file statici che definiscono in maniera chiara quali sono le risorse da creare e quali le modalità di rilascio, sfruttando questi template. E se nascesse l'esigenza di avere, magari tramite una soluzione esterna, dei parametri da passare nei file per rendere i propri *modelli* adatti a più contesti o situazioni? Nel mondo di Kubernetes, esistono diversi strumenti che possiamo utilizzare, e tra questi ci sono Helm e Kustomize: uno è il gestore di pacchetti per Kubernetes e include anche funzionalità di creazione di *manifest*, mentre Kustomize è un *tool* perfetto per chi lavora bene con la filosofia dichiarativa di Kubernetes.

Helm

Quando esegui un'applicazione su Kubernetes potresti dover distribuire così tanti oggetti come Deployment, ConfigMap, Secret, che definirli semplicemente in un file potrebbe non essere la via più veloce, soprattutto se la configurazione avviene più volte. Helm è il gestore di pacchetti per Kubernetes che serve a fornire una soluzione per la gestione dei pacchetti, la sicurezza e la configurabilità della tua applicazione durante la sua installazione su un cluster Kubernetes. Può essere utilizzato tramite riga di comando, permettendoci di eseguire tutte le attività di creazione e gestione delle risorse dell'applicazione associata.

Installazione

Cominciamo dall'inizio, ossia dall'installazione: Helm è disponibile sul repository GitHub ufficiale (<https://github.com/helm/helm/releases>), da cui possiamo scaricare la versione più recente e installarla, a seconda del sistema operativo con cui stiamo lavorando. Attualmente la versione più recente e stabile è la 3.10.3:

The screenshot shows the Helm v3.11.2 release page. At the top, it says "Helm v3.11.2" and "Latest". Below that, a message encourages users to upgrade for the best experience. It then invites the community to grow by joining discussions on Slack, hanging out on Zoom, and testing charts on ArtifactHub. A section titled "Installation and Upgrading" provides download links for common platform binaries:

- MacOS amd64 ([checksum](#) / 404938fd2c6eff9e0dab830b0db943fca9e1572cd3d7ee40904705760faa390f)
- MacOS arm64 ([checksum](#) / f61a3aa55827de2d8c64a2063fd744b618b443ed063871b79f52069e90813151)
- Linux amd64 ([checksum](#) / 781d826daec584f9d50a01f0f7dadfd25a3312217a14aa2fbb85107b014ac8ca)
- Linux arm ([checksum](#) / 444b65100e224beee0a3a3a54cb19dad37388fa9217ab2782ba63551c4a2e128)
- Linux arm64 ([checksum](#) / 0a60baac83c3106017666864e664f52a4e16fdb578ac009f9a85456a9241c5db)
- Linux i386 ([checksum](#) / dee028554da99415eb19b4b1fd423db390f84d49e4c4cbc3df5d6f658ec7f38)

Figura 11.1 Download della release di Helm.

Per installarlo, è sufficiente selezionare l'eseguibile corretto e procedere seguendo le modalità previste dal sistema operativo a disposizione; per esempio, per Windows o Linux, sarà sufficiente estrarre dall'archivio in formato .zip i file in una cartella a piacere e aggiungere il percorso di questa alla variabili di ambiente PATH per poterlo utilizzare da riga di comando. Per MacOS, è possibile sfruttare brew o Chocolatey per installarlo.

Una volta configurato, per verificarne il corretto funzionamento, potremo aprire un terminale e digitare il seguente comando in attesa di un messaggio simile a quello riportato:

Listato 11.1 Test dell'installazione di Helm

```
helm list
>>>
NAME      NAMESPACE      REVISION      UPDATED      STATUS      CHART      APP      VERSION
...
```

Al momento, la lista restituita è ovviamente vuota, ma non lo sarà una volta che avremo iniziato a lavorarci. Prima di procedere con la parte pratica, diamo una rapida occhiata alle risorse di Helm: i Chart.

Chart

Un Chart in Helm non è altro che una versione pacchettizzata della tua applicazione. È, infatti, un insieme di file e directory che segue alcune specifiche per descrivere le risorse da installare in Kubernetes, all'interno di uno o più file YAML: possiamo definire al suo interno le informazioni sulla versione, il nome, la descrizione e così via dell'applicazione, e di conseguenza del Chart. Quando lavoriamo con Helm, tutti i file che il Chart dovrà contenere saranno inclusi in una cartella chiamata `templates`, mentre i valori che devono essere sostituiti saranno all'interno del file `values.yaml`. Questo file viene letto e va a sovrascrivere tutti i parametri con gli argomenti che vogliamo applicare alle risorse da implementare, e già questo ci dà un'informazione molto importante: tramite un Chart possiamo

parametrizzare la nostra installazione, di modo che sia adatta per più contesti: se, per esempio, volessimo creare delle risorse che siano adatte a più fasi di sviluppo, come test e produzione, potremmo pensare di astrarre queste componenti attraverso il file `values.yaml`, così da sostituirle con le informazioni corrette al momento opportuno.

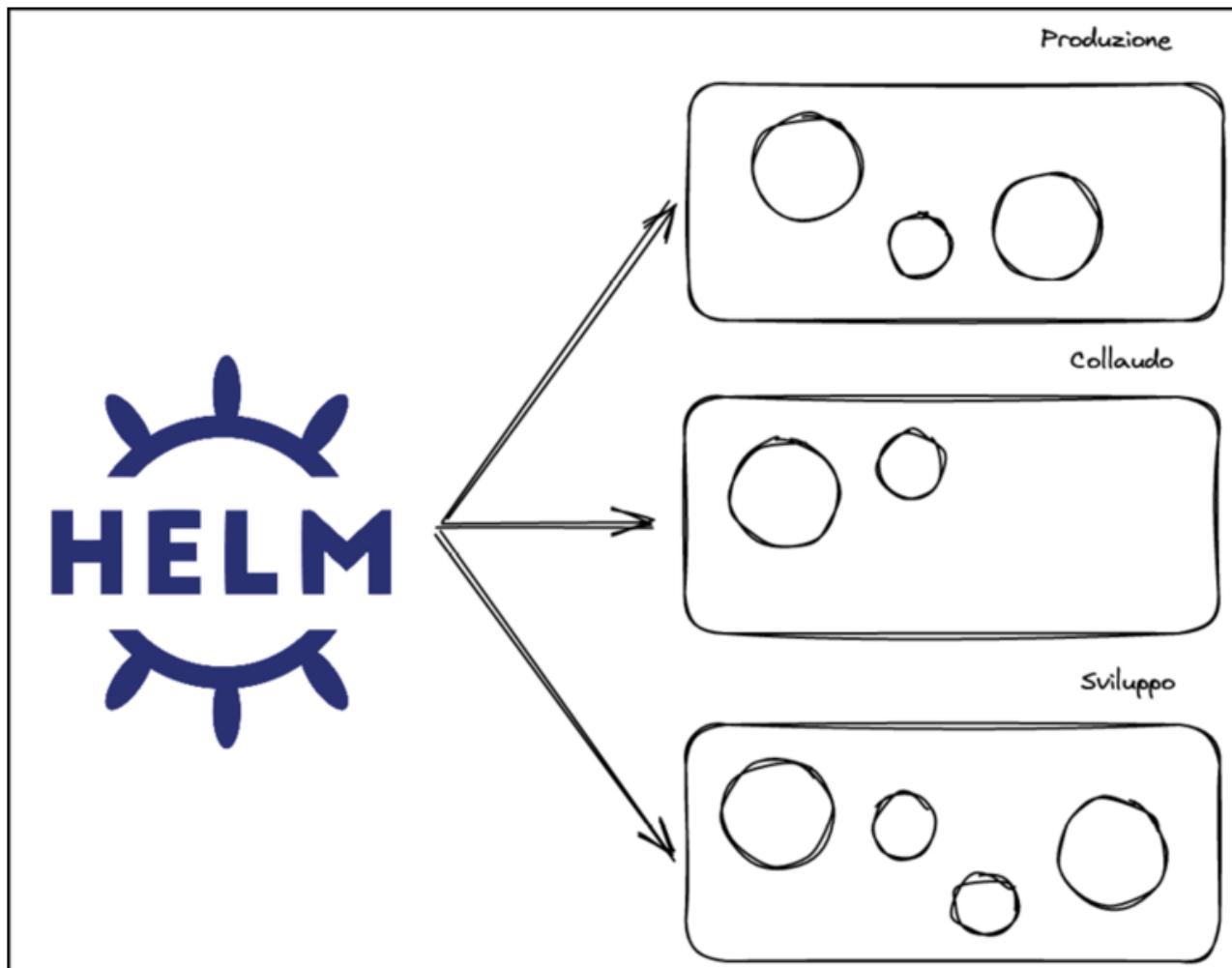


Figura 11.2 Come è possibile utilizzare Helm.

Per poter prendere confidenza con questo oggetto, l'approccio migliore è quello di vedere un esempio pratico: creiamo il primo Chart, in modo da comprenderne la struttura dei file su cui andremo a lavorare. Per farlo, è sufficiente eseguire il comando `helm create` e assegnare un nome al Chart:

Listato 11.2 Creazione di un Chart

```
helm create my-chart
>>>
Creating my-chart
...
```

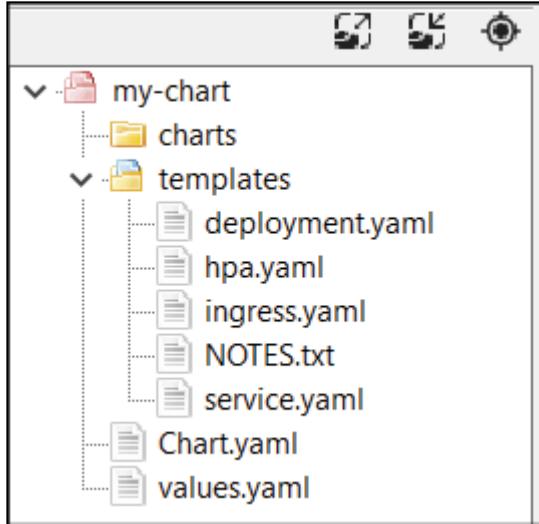


Figura 11.3 Esempio di struttura di un Chart.

Ora che abbiamo creato il Chart, diamo un'occhiata alla sua struttura per vedere che cosa c'è dentro alla cartella e ai relativi file. I primi due file che ci interessano sono `Chart.yaml` e `values.yaml`, che definiscono le informazioni generiche del Chart e quali parametri possono essere valorizzati al suo interno al momento dell'implementazione.

Listato 11.3 Esempio di Chart.yaml

```

apiVersion: v2
name: buildachart
description: A Helm chart for Kubernetes

# A chart can be either an 'application' or a 'library' chart.
#
# Application charts are a collection of templates that can be packaged into versioned
archives
# to be deployed.
#
# Library charts provide useful utilities or functions for the chart developer. They're
included as
# a dependency of application charts to inject those utilities and functions into the
rendering
# pipeline. Library charts do not define any templates and therefore cannot be deployed.
type: application

# This is the chart version. This version number should be incremented each time you make
changes
# to the chart and its templates, including the app version.
version: 0.1.0

# This is the version number of the application being deployed. This version number should
be
# incremented each time you make changes to the application.
appVersion: 1.16.0

```

La prima parte include la versione dell'API utilizzata dal Chart (campo obbligatorio), il suo nome e una descrizione. La sezione successiva descrive il tipo di Chart (viene segnata `application` per impostazione predefinita), la versione del Chart che distribuirai e la versione dell'applicazione (che dovrebbe essere incrementata man mano che apporti modifiche). La parte più importante del grafico è la directory chiamata `templates`: contiene tutte le configurazioni per l'applicazione che verrà distribuita

nel cluster. Come visibile nell'immagine, questa applicazione ha un Deployment, un Ingress, un Service e un ServiceAccount predefiniti. Questa directory include anche una directory di test, che include un test di connessione all'app. C'è infine un'altra directory, chiamata `charts`, che è vuota: consente di aggiungere Chart dipendenti che sono necessari per distribuire un'applicazione. Alcuni grafici Helm per le applicazioni hanno fino a quattro Chart aggiuntivi per l'applicazione principale: quando ciò accade, il file `values.yaml` viene aggiornato con i valori per ciascun Chart in modo che le applicazioni vengano configurate e distribuite contemporaneamente.

Un'altra nota degna di attenzione riguarda il file `values.yaml`: per personalizzare il tuo Chart Helm, devi modificare questo file che, per impostazione predefinita, ha il seguente aspetto:

Listato 11.4 Esempio di values.yaml

```
# Default values for buildachart.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.

replicaCount: 1

image:
  repository: nginx
  pullPolicy: IfNotPresent

imagePullSecrets: []
nameOverride: ""
fullnameOverride: ""

serviceAccount:
  # Specifies whether a service account should be created
  create: true
  # Annotations to add to the service account
  annotations: {}
  # The name of the service account to use.
  # If not set and create is true, a name is generated using the fullname template
  name:

podSecurityContext: {}
  # fsGroup: 2000

securityContext: {}
  # capabilities:
  #   drop:
  #     - ALL
  # readOnlyRootFilesystem: true
  # runAsNonRoot: true
  # runAsUser: 1000

service:
  type: ClusterIP
  port: 80

ingress:
  enabled: false
  annotations: {}
    # kubernetes.io/ingress.class: nginx
    # kubernetes.io/tls-acme: "true"
  hosts:
    - host: chart-example.local
      paths: []
  tls: []
    # - secretName: chart-example-tls
    #   hosts:
    #     - chart-example.local
```

```

resources: {}
  # We usually recommend not to specify default resources and to leave this as a conscious
  # choice for the user. This also increases chances charts run on environments with little
  # resources, such as Minikube. If you do want to specify resources, uncomment the
  following
  # lines, adjust them as necessary, and remove the curly braces after 'resources:'.
  # limits:
  #   cpu: 100m
  #   memory: 128Mi
  # requests:
  #   cpu: 100m
  #   memory: 128Mi

nodeSelector: {}

tolerations: []

affinity: {}

```

A partire dall'alto, puoi vedere che `replicaCount` è impostato automaticamente a uno, il che significa che verrà istanziato un solo Pod. La sezione relativa all'immagine ha due cose che dobbiamo guardare: il repository da cui stiamo estraendo l'immagine e la `pullPolicy`, che è impostata su `IfNotPresent`, il che significa che l'immagine verrà aggiornata se non già presente nel cluster. Successivamente, diamo un'occhiata ai parametri `imagePullSecrets` e via dicendo: questo in particolare ci permette di specificare una risorsa di tipo `Secret` che contiene una password o una chiave API per recuperare l'immagine tramite un registro privato. Poi ci sono `nameOverride` e `fullnameOverride`: dal momento in cui hai eseguito `helm create`, il suo nome (`buildachart`) è stato aggiunto a una serie di file di configurazione. Se devi rinominare un Chart, questa sezione è il posto migliore per farlo, in modo da non perdere nessuna sostituzione nei file di configurazione.

Listato 11.5 Parziale di values.yaml per il pull dell'immagine

```

# Default values for buildachart.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.

replicaCount: 1

image:
  repository: nginx
  pullPolicy: IfNotPresent

imagePullSecrets: []
nameOverride: ""
fullnameOverride: ""

```

Gli `account di servizio`, o `ServiceAccount`, forniscono un'identità all'utente associato all'esecuzione nel Pod all'interno del cluster. Se viene lasciato vuoto, il nome verrà generato in base al nome completo utilizzando il file `helpers.tpl`. Il consiglio, genericamente parlando, è quello di impostare sempre un account di questo tipo in modo che l'applicazione venga associata direttamente a un utente specifico.

Listato 11.6 Parziale di values.yaml relativo ai Service Account

```

serviceAccount:
  # Specifies whether a service account should be created
  create: true
  # Annotations to add to the service account
  annotations: {}
  # The name of the service account to use.

```

```

# If not set and create is true, a name is generated using the fullname template
name:

### Esempio
serviceAccount:
  # Specifies whether a service account should be created
  create: true
  # Annotations to add to the service account
  annotations: {}
  # The name of the service account to use.
  # If not set and create is true, a name is generated using the fullname template
  Name: tomriddle

```

Sempre all'interno del file `values.yaml`, puoi configurare i dettagli relativi alla sicurezza del Pod per impostare limiti su quale tipo di gruppo di filesystem utilizzare o su quale utente può e non può essere utilizzato. Comprendere queste opzioni è importante per proteggere un Pod.

Listato 11.7 Parziale di `values.yaml` relativo alla sicurezza

```

podSecurityContext: {}
  # fsGroup: 2000

securityContext: {}
  # capabilities:
  #   drop:
  #     - ALL
  # readOnlyRootFilesystem: true
  # runAsNonRoot: true
  # runAsUser: 1000

```

Relativamente alla connessione di rete, ci sono due diversi tipi di opzioni nel file di default: uno utilizza una rete di servizio locale con un indirizzo ClusterIP tramite il Service, l'altro configura un Ingress. La scelta di questi valori dipende da come vogliamo rendere l'applicazione raggiungibile: solo all'interno del cluster o anche dall'esterno.

Listato 11.8 Parziale di `values.yaml` relativo alla connessione

```

service:
  type: ClusterIP
  port: 80

ingress:
  enabled: false
  annotations: {}
    # kubernetes.io/ingress.class: nginx
    # kubernetes.io/tls-acme: "true"
  hosts:
    - host: chart-example.local
      paths: []
  tls: []
    # - secretName: chart-example-tls
    #   hosts:
    #     - chart-example.local

```

Infine, come ultima (ma non per importanza), abbiamo la sezione relativa alle risorse: Helm ti consente di allocare in modo esplicito le risorse hardware da associare agli oggetti Kubernetes. Puoi configurare la quantità minima e massima di risorse che un Chart può richiedere, in termini di millicores per la CPU (espressa con `m`) e di Mebibyte o Gibibyte per la memoria. Le ultime tre voci sono molto specifiche e riguardano l'infrastruttura di Kubernetes e come distribuire i Pod sui vari nodi: il consiglio è quello di tornare a dare un'occhiata qui dopo aver letto il Capitolo 9.

Listato 11.9 Esempio di `values.yaml` relativo alle risorse

```

resources: {}
  # We usually recommend not to specify default resources and to leave this as a conscious
  # choice for the user. This also increases chances charts run on environments with little
  # resources, such as Minikube. If you do want to specify resources, uncomment the
following
  # lines, adjust them as necessary, and remove the curly braces after 'resources:'.
  # limits:
  #   cpu: 100m
  #   memory: 128Mi
  # requests:
  #   cpu: 100m
  #   memory: 128Mi

nodeSelector: {}

tolerations: []

affinity: {}

```

Mebibyte vs megabyte

Il mebibyte (MiB) è un'unità di misura dell'informazione o della quantità di dati e una delle differenze con in megabyte (MB) è puramente matematica: il MB è uguale a 10 alla sesta, mentre il MiB è uguale a 2 alla ventesima, ristabilendo quindi un nuovo multiplo per i byte.

Così come avviene per le immagini Docker, anche Helm ha un registry pubblico dove poter cercare un Chart: è il caso di *ArtifactHub*, progetto open source che espone tutti i progetti appartenenti al mondo della CNCF (Riferimento al progetto: <https://www.cncf.io/projects/artifact-hub/>). Se volessi utilizzare un Chart per installare Mongo, uno dei database non relazionali tra i più conosciuti, potremmo cercare all'interno del sito un Chart per l'ultima versione disponibile:

Figura 11.4 Pagina iniziale di ArtifactHub.

Di default, quando Helm è installato, non conosce alcun repository, motivo per cui è necessario aggiungere quelli che ci interessano per poter lavorare con i Chart desiderati. Per poter quindi lavorare con Mongo, sarebbe sufficiente aggiungere un repository di riferimento alla nostra istanza e installare la risorsa desiderata, tenendo presente che non tutti i repository pubblicati sono affidabili: i primi due

mostrati nella Figura 11.4 sono stati aggiunti da utenti e non organizzazioni, per cui le immagini potrebbero non essere verificate. In questo caso, per andare sul sicuro, sfruttiamo una delle organizzazioni più conosciute in ambito *open source* per il rilascio di immagini, ossia Bitnami, e aggiungiamo questo repository al nostro cluster, tramite il seguente comando:

Listato 11.10 Aggiunta del repository

```
helm repo add bitnami https://charts.bitnami.com/bitnami  
>>>
```

Per procedere all'installazione, sarà invece necessario eseguire il comando `helm install` assegnando un nome al Chart (ossia, `mongo-release1`) e segnalando qual è quello da utilizzare (nel nostro caso, `bitnami/mongodb`).

Listato 11.11 Installazione di Mongo

```
helm install mongo-release1 bitnami/mongodb  
>>>
```

Questo comando ci permette anche di installare diverse versioni di Mongo, specificando una release diversa per ogni Chart installabile. L'elenco completo dei rilasci installabili è disponibile utilizzando il comando seguente:

Listato 11.12 Elenco delle release disponibili

```
helm list  
>>>  
NAME          NAMESPACE      REVISION    ...      STATUS      CHART  
APP VERSION  
mongo-release1      k8s-training      1          deployed  
mongodb-13.9.1      6.0.5
```

A questo punto, una volta che Mongo sarà stato istanziato tramite il Chart, sarà possibile vederne le risorse attive, così come faremmo con una "normale" installazione tramite un *manifest* di Kubernetes:

Listato 11.13 Aggiunta del repository

```
kubectl get deployments  
>>>  
NAME          READY   UP-TO-DATE   AVAILABLE   AGE  
mongo-release1-mongodb   1/1       1           1           99s  
  
kubectl get pods  
>>>  
NAME          READY   STATUS     RESTARTS  
AGE  
mongo-release1-mongodb-8d5cfb765-kj741   1/1       Running   0  
118s
```

L'istanza di Mongo è stata installata correttamente con dei parametri di default, che possono certamente essere personalizzati a seconda dell'esigenza (ci arriveremo tra un secondo). Invece, per disinstallare un Chart, è sufficiente utilizzare il comando `helm uninstall`:

Listato 11.14 Rimozione del Chart

```
helm uninstall mongo-release1  
# oppure  
helm delete mongo-release1  
>>>  
release "mongo-release1" uninstalled
```

Mongo è però un esempio piuttosto semplice, e non rende bene l'idea delle potenzialità: proviamo a installare Wordpress con Helm.

Installare Wordpress tramite Helm

WordPress è la piattaforma di blogging e gestione dei contenuti più popolare al mondo. Potente ma semplice, utilizzata da tutti, dagli studenti alle aziende globali, per creare siti web belli e funzionali. Per installarla tramite Helm, Bitnami ha messo a disposizione un Chart che è possibile utilizzare per configurare un'istanza del celebre blog in un attimo: eseguiamo il seguente comando e configuriamo il repository Bitnami con la relativa istanza di Wordpress da installare.

Listato 11.15 Configurazione del repo e installazione di Wordpress

```
$ helm repo add my-repo https://charts.bitnami.com/bitnami
$ helm install my-release my-repo/wordpress
>>>
NAME: my-release
LAST DEPLOYED: Fri Mar 17 12:00:05 2023
NAMESPACE: kube-public
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
CHART NAME: wordpress
CHART VERSION: 15.2.55
APP VERSION: 6.1.1

** Please be patient while the chart is being deployed **
```

Your WordPress site can be accessed through the following DNS name from within your cluster:

my-release-wordpress.kube-public.svc.cluster.local (port 80)

To access your WordPress site from outside the cluster follow the steps below:

1. Get the WordPress URL by running these commands:

NOTE: It may take a few minutes for the LoadBalancer IP to be available.
Watch the status with: 'kubectl get svc --namespace kube-public -w my-release-wordpress'

```
export SERVICE_IP=$(kubectl get svc --namespace kube-public my-release-wordpress --template "{{ range (index .status.loadBalancer.ingress 0) }}{{ . }}{{ end }}")
echo "WordPress URL: http://$SERVICE_IP/"
echo "WordPress Admin URL: http://$SERVICE_IP/admin"
```

2. Open a browser and access WordPress using the obtained URL.

3. Login with the following credentials below to see your blog:

```
echo Username: user
echo Password: $(kubectl get secret --namespace kube-public my-release-wordpress -o jsonpath=".data.wordpress-password" | base64 -d)
```

Apparentemente, siamo nella stessa situazione di prima: tutto ha funzionato ed è andato liscio come l'olio, e in effetti è proprio così. Questo però è il caso più semplice davanti al quale ci possiamo trovare, perché capiterà molto spesso di dover *personalizzare* la soluzione in base alle esigenze: se, per esempio, volessimo usare una versione particolare di Wordpress, piuttosto che specificare un utente e una password, o il nome del blog, potremmo farlo tramite i *parametri*. Questi si trovano all'interno del file `values.yaml`, che potremo modificare assegnando dei valori diversi da quelli predefiniti:

Listato 11.16 Modifica dei parametri di default del Chart di Wordpress

```

...
## @param wordpressUsername WordPress username
##
wordpressUsername: Tom
## @param wordpressPassword WordPress user password
## Defaults to a random 10-character alphanumeric string if not set
##
wordpressPassword: "mypassword"
## @param existingSecret Name of existing secret containing WordPress credentials
## NOTE: Must contain key `wordpress-password`
## NOTE: When it's set, the `wordpressPassword` parameter is ignored
##
existingSecret: ""
## @param wordpressEmail WordPress user email
##
wordpressEmail: tom.riddle@google.com
## @param wordpressFirstName WordPress user first name
##
wordpressFirstName: Tom
## @param wordpressLastName WordPress user last name
##
wordpressLastName: Riddle
## @param wordpressBlogName Blog name
##
wordpressBlogName: Don't call me Blog!
## @param wordpressTablePrefix Prefix to use for WordPress database tables
##
...

```

Nel caso di esempio, mostriamo solo una parte dei parametri che Bitnami ha messo a disposizione per questo Chart, anche se la lista è ben più lunga: l'elenco completo è disponibile sul repository GitHub ufficiale, a questo indirizzo:

<https://github.com/bitnami/charts/tree/main/bitnami/wordpress>.

Parametrizzazione: che maledista!

Può sembrare una follia quella di specificare tutti i parametri di configurazione dell'applicazione che stiamo utilizzando, ma la realtà è che questo ci consente di adattarne le capacità a seconda del contesto: immagina di astrarre il tuo lavoro e lavorarci una sola volta per renderlo disponibile su più piattaforme, solo cambiando dei valori all'interno di un unico file, centralizzato. Non reinventare la ruota, piuttosto rendila funzionante una sola volta!

Creare un Chart custom

Arriverà il momento in cui vorremmo distribuire la nostra applicazione su Kubernetes e Helm potrebbe farci davvero comodo: potrebbe quindi essere utile vedere un esempio di come rendere un'applicazione qualsiasi pronta per essere distribuita tramite Chart.

Utilizziamo un'app Node.js, un popolare framework di sviluppo per *web application*, per costruire un Chart che ci permetta di installarla facilmente su Kubernetes. Per farlo, sfrutteremo il seguente Dockerfile.

Listato 11.17 Dockerfile per il deploy di un'applicazione Node.js

```

FROM node:10 AS ui-build
WORKDIR /usr/src/app
COPY my-app/ ./my-app/
RUN cd my-app && npm install @angular/cli && npm install && npm run build

FROM node:10 AS server-build
WORKDIR /root/
COPY --from=ui-build /usr/src/app/my-app/dist ./my-app/dist
COPY package*.json ./
```

```

RUN npm install
COPY server.js .

EXPOSE 3000

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

```

Una volta eseguita la build dell'immagine, verificato che l'applicazione funziona correttamente, possiamo pubblicare l'immagine su un qualsiasi repository pubblico, da cui sia poi possibile recuperare l'immagine: possiamo, per esempio, sfruttare il DockerHub e *pushare* l'immagine con i seguenti comandi.

Listato 11.18 Pubblicare l'immagine su DockerHub

```

# login
docker login # tag the image
docker tag angular-node-image myuser/angular-node-webapp # push the image
docker push myuser/angular-node-webapp

```

Adesso che l'immagine è pubblica, possiamo procedere a creare le risorse necessarie all'installazione dell'applicazione su Kubernetes: in questo caso, è facile immaginarsi che avremo bisogno di un Deployment per contenere l'immagine Node.js e di un Service che permetta di accedervi esternamente.

Listato 11.19 Esempio di manifest.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: angular-webapp
    name: angular-webapp
spec:
  replicas: 5
  selector:
  matchLabels:
    app: angular-webapp
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
    labels:
      app: angular-webapp
      spec:
    containers:
      - image: docker.io/myuser/angular-node-webapp
    name: webapp
    imagePullPolicy: Always
    resources: {}
  ports:
    - containerPort: 3080
status: {}

---
apiVersion: v1
kind: Service
metadata:
  name: angular-webapp
  labels:
    run: angular-webapp
spec:
  ports:

```

```

- port: 3080
  protocol: TCP
selector:
  app: angular-webapp
type: NodePort

```

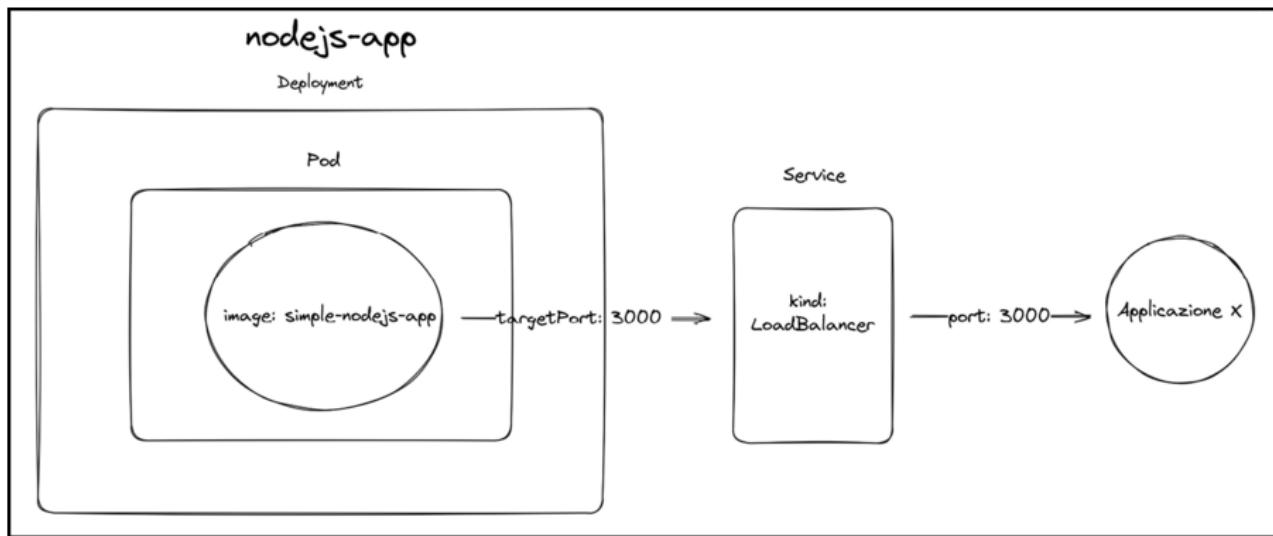


Figura 11.5 Architettura dell'applicazione costruita tramite il Chart.

Come visto in precedenza, per creare un Helm Chart è sufficiente eseguire il comando `helm create` e assegnargli un nome: questo creerà una serie di file che servono a distribuire l'applicazione su Kubernetes, tra cui `deployment.yaml` e `service.yaml`. Questi file sono quelli con cui ci interessa lavorare, mentre per il momento `hpa.yaml` o `serviceaccount.yaml` non sono necessari, per cui possono essere rimossi o ignorati. All'interno del file `values.yaml` andiamo a specificare le informazioni che ci interessa configurare, come il numero di repliche presenti all'interno del Deployment, piuttosto che l'immagine, come nel seguente caso:

Listato 11.20 Configurazione del file values.yaml

```

# Default values for angular-node-chart.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.

replicaCount: 10

image:
  repository: docker.io/bbachin1/angular-node-webapp
  pullPolicy: Always

application:
  name: angular-webapp

```

Per rendere poi il Deployment *parametrizzabile*, è sufficiente includere tra parentesi graffe nel file relativo i valori che dovranno essere letti dal file `values.yaml` e sostituiti al momento di inizializzazione del Chart su Kubernetes.

Listato 11.21 Parametri del file deployment.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:

```

```

    app: angular-webapp
    name: {{.Values.application.name}}
spec:
  replicas: {{.Values.replicaCount}}
  selector:
    matchLabels:
      app: angular-webapp
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
    labels:
      app: angular-webapp
      spec:
      containers:
        - image: {{.Values.image.repository}}
    name: webapp
    imagePullPolicy: {{.Values.image.pullPolicy}}
    resources: {}
    ports:
      - containerPort: 3080
status: {}

```

Listato 11.22 Parametri del file service.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: {{.Values.application.name}}
  labels:
    run: angular-webapp
spec:
  ports:
    - port: 3080
      protocol: TCP
  selector:
    app: angular-webapp
  type: NodePort

```

Come ultimo step, andiamo a lavorare con il file `chart.yaml`, il quale espone i metadati del Chart, come nome, versione, descrizione, e così via: non si tratta di un'operazione obbligatoria, ma fortemente suggerita per mantenere coerente il lavoro fatto, e magari per renderlo anche consistente se utilizziamo delle piattaforme di versionamento del software dove tracciamo i cambiamenti dei file con cui lavoriamo!

Listato 11.23 Esempio del file chart.yaml

```

apiVersion: v2
name: angular-node-chart
description: A Helm chart for Kubernetes

# A chart can be either an 'application' or a 'library' chart.
#
# Application charts are a collection of templates that can be packaged into versioned
archives
# to be deployed.
#
# Library charts provide useful utilities or functions for the chart developer. They're
included as
# a dependency of application charts to inject those utilities and functions into the
rendering
# pipeline. Library charts do not define any templates and therefore cannot be deployed.
type: application

```

```

# This is the chart version. This version number should be incremented each time you make
changes
# to the chart and its templates, including the app version.
# Versions are expected to follow Semantic Versioning (https://semver.org/)
version: 0.1.0

# This is the version number of the application being deployed. This version number should
be
# incremented each time you make changes to the application. Versions are not expected to
# follow Semantic Versioning. They should reflect the version the application is using.
appVersion: 1.16.0

```

Per “impacchettare” tutto quello che abbiamo fatto finora, è sufficiente eseguire il comando omonimo, che raggruppa i file all’interno di un archivio in formato `.tgz`:

Listato 11.24 Package del Chart

```
helm package angular-node-chart
```

Questo ci permetterà di distribuire la nostra applicazione e installarla su qualsiasi cluster in maniera semplice e veloce: per configurarla in un’altra istanza Kubernetes con il comando `helm install`, dovremo solo sostituire il repository con il nome dell’archivio che abbiamo generato in precedenza:

Listato 11.25 Installazione di un Chart in formato `.tgz` in locale

```
helm install release1 angular-node-chart-0.1.0.tgz
```

L’esempio è completo, anche se c’è un dettaglio che potrebbe fare la differenza per chi usa il nostro codice, o anche per noi se passa un po’ di tempo tra quando lo creiamo per la prima volta e quando lo riprendiamo in mano: tra i file presenti nella cartella del Chart, può esistere un file sotto `templates` chiamato `NOTES.txt`, che permette di descrivere con del semplice testo delle informazioni che verranno stampate dopo l’installazione e durante la visualizzazione dello stato di una versione. Questo file può essere utilizzato per visualizzare delle note sull’utilizzo dell’applicazione o sui passaggi successivi all’installazione per facilitarne la configurazione: per esempio, si potrebbero fornire delle istruzioni per la connessione a un database o l’indirizzo per l’accesso a un’interfaccia utente web. Poiché questo file viene stampato sullo *standard output* del terminale, si consiglia di mantenere il contenuto breve e puntare al `README.md` per maggiori dettagli implementativi.

Listato 11.26 Risultato

```

NAME: release
LAST DEPLOYED: Tue Feb 16 20:16:18 2023
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
1. Get the application URL by running these commands:
kubectl get svc
kubectl cluster-info

```

Kustomize

Kustomize è uno degli strumenti più utili nell’ecosistema Kubernetes per semplificare la distribuzione delle tue soluzioni, consentendo di creare un’intera applicazione Kubernetes a partire da singoli pezzi, senza toccare i file di configurazione YAML dei singoli componenti. Kustomize funziona grazie alla riga di comando e si integra con gli oggetti Kubernetes: infatti, a partire dalla versione 1.14 di `kubectl`, consente di apportare modifiche usando il paradigma *dichiarativo* alle configurazioni senza toccare il manifest YAML. Per esempio, puoi combinare risorse che provengono da fonti diverse, mantenere le

tue personalizzazioni (o *kustomizations*, a seconda dei casi) nel codice sorgente e creare *sovraposizioni* per contesti specifici, creando un file che collega tutto insieme o, facoltativamente, include “sostituzioni” per i singoli parametri.

Se fin qui hai un po’ di confusione, nessun problema: vedrai che “sporcandosi” un po’ le mani, sarà più chiaro. Partiamo dall’installazione: se stai utilizzando l’ultima versione di `kubectl` (a oggi, la 1.26), dovrà avere già Kustomize integrato. Se stai utilizzando una versione di `kubectl` precedente alla 1.14 o desideri semplicemente installare Kustomize senza `kubectl`, puoi seguire le istruzioni riportate di seguito per installarlo. In alcuni casi, i comandi differiranno tra la versione *standalone* e quella integrata: verranno riportati i comandi per completare la parte pratica con entrambe le modalità.

Per installarlo tramite il file binario, è possibile eseguire il seguente comando su qualunque sistema operativo supporti `curl`.

Listato 11.27 Installazione Kustomize tramite file binario

```
curl -s "https://raw.githubusercontent.com/kubernetes-sigs/kustomize/master/hack/install_kustomize.sh" | bash
```

Altrimenti, un’altra opzione è utilizzare Go: se usi questo linguaggio di programmazione e hai una versione superiore alla 1.13, puoi seguire queste istruzioni:

Listato 11.28 Installazione Kustomize tramite Go

```
unset GOPATH  
unset GO111MODULES  
git clone git@github.com:kubernetes-sigs/kustomize.git # clone del repository ufficiale di kustomize  
cd kustomize # cambio cartella e mi posiziono in quella del repository appena clonato  
  
git checkout kustomize/v5.0.0 # scelgo di utilizzare la versione 5.0.0 (è possibile usare quella presente sul branch principale)  
make kustomize # build del binario  
~/go/bin/kustomize version # eseguo il comando kustomize version
```

Le ultime due alternative prevedono l’uso di Chocolatey, un gestore di pacchetti molto popolare per Windows e MacOS, oppure brew per i soli sistemi MacOS:

Listato 11.29 Installazione Kustomize tramite Chocolatey

```
choco install kustomize
```

Listato 11.30 Installazione Kustomize tramite brew

```
brew install kustomize
```

Ora che Kustomize è installato, vediamo un esempio pratico e sfruttiamo sempre Wordpress: diciamo che abbiamo due file che descrivono due risorse separate per mettere in esecuzione questa applicazione, ossia un Deployment per ospitare il container principale e un Service per poter raggiungere tramite il browser il blog; possiamo sfruttare Kustomize per sfruttare i file YAML a nostra disposizione ed eseguire Wordpress con pochi comandi. Partiamo dai file: quelli riportati di seguito sono quelli di riferimento per il nostro caso d’uso:

Listato 11.31 Esempio di Deployment di Wordpress

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: wordpress  
  labels:  
    app: wordpress  
spec:
```

```

selector:
  matchLabels:
    app: wordpress
strategy:
  type: Recreate
template:
  metadata:
    labels:
      app: wordpress
spec:
  containers:
    - image: wordpress:6.1.1-apache
      name: wordpress
      ports:
        - containerPort: 80
          name: wordpress
      volumeMounts:
        - name: wordpress-persistent-storage
          mountPath: /var/www/html
  volumes:
    - name: wordpress-persistent-storage
  emptyDir: {}

```

Listato 11.32 Esempio di Service di Wordpress

```

apiVersion: v1
kind: Service
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  ports:
    - port: 80
  selector:
    app: wordpress
  type: LoadBalancer

```

Creiamo una cartella `wordpress` (o con il nome che preferite) e al suo interno creiamo due file, uno per ogni risorsa: quando avremo tutti i file all'interno della directory, potremo eseguire il comando seguente per verificare che le risorse vengano create correttamente:

Listato 11.33 Creazione delle risorse per Wordpress

```

kubectl apply -f wordpress/
>>>
deployment.apps/wordpress created
service/wordpress created

```

Ottimo! Quindi i nostri file sono a posto. Cancelliamo queste risorse e iniziamo a sperimentare con Kustomize: un po' come visto in precedenza con Helm, questo strumento ci dà la possibilità di definire di cosa eseguire la build all'interno del cluster tramite un unico file. Questo file si chiama `kustomization.yaml` e ci permette di aggiungere anche diversi livelli di logica.

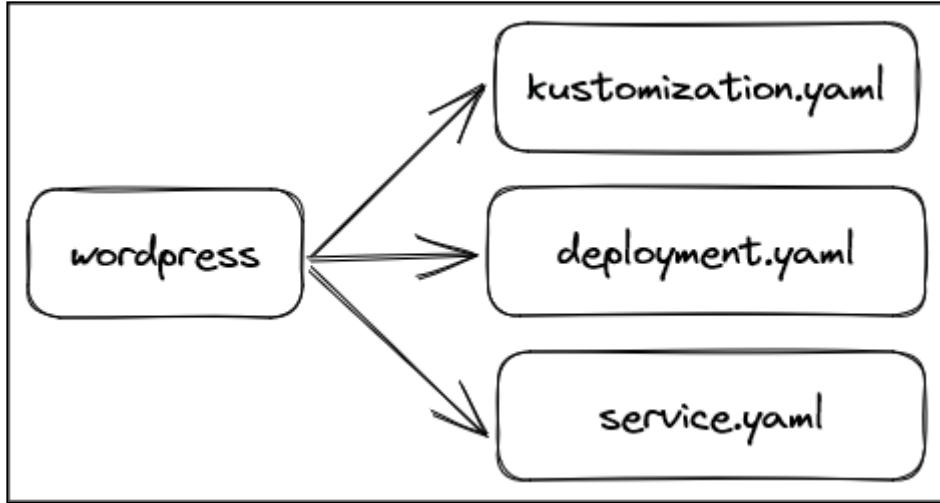


Figura 11.6 Esempio di file per kustomizzare un'applicazione come Wordpress.

Partiamo da un caso semplice: all'interno di entrambi i file, le risorse hanno delle label `app` che riportano il valore `wordpress`; questo ci permette, attraverso il file `kustomization.yaml` di definire quali sono le risorse che è necessario eseguire all'interno del cluster Kubernetes sfruttando sia i file che abbiamo creato sia la label che deve utilizzare per selezionare le risorse giuste, per creare un solo file che potremo eseguire tramite un solo comando:

Listato 11.34 Esempio di file wordpress/kustomization.yaml

```

commonLabels:
  app: my-wordpress
resources:
- deployment.yaml
- service.yaml

```

Nella prima parte specifichiamo che la label `app` andrà sovrascritta con una personalizzata, o *kustomizzata*: questo ci permetterà di applicare delle modifiche annotando le risorse specificate con una nuova etichetta, come definito nella seconda parte. Per prendere confidenza con Kustomize, eseguiamo il comando `kustomize build`, che ci permette di assemblare il file YAML: il risultato che ci aspettiamo è di ottenere un unico YAML con entrambe le risorse e la nuova label `app` applicata:

Listato 11.35 Esempio di build

```

kustomize build wordpress/
# oppure, se usi kubectl, non è necessario specificare il comando build e basta usare
kubectl kustomize
kubectl kustomize wordpress/

```

L'output prodotto sarà simile al seguente:

Listato 11.36 Esempio di build

```

apiVersion: v1
kind: Service
metadata:
  labels:
    app: my-wordpress
    name: wordpress
spec:
  ports:
  - port: 80
  selector:

```

```

app: my-wordpress
type: LoadBalancer
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wordpress
  labels:
    app: my-wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: my-wordpress
    spec:
      containers:
        - image: wordpress:6.1.1-apache
          name: wordpress
        ports:
          - containerPort: 80
            name: wordpress
      volumeMounts:
        - name: wordpress-persistent-storage
          mountPath: /var/www/html
      volumes:
        - name: wordpress-persistent-storage
          emptyDir: {}

```

Questo file è il risultato dell'unione dei diversi file (non dell'applicazione al cluster); può essere salvato in un file apposito ed eseguito successivamente per creare tutte le risorse necessarie a Wordpress per partire. Un secondo, ci stiamo dimenticando un pezzo: Wordpress ha bisogno di un database per funzionare! Per fortuna, abbiamo la definizione di un Deployment, di un Service (per collegarci da Wordpress) e di un Secret (per le credenziali) per MySQL che fa al caso nostro: creiamo una cartella `mysql` allo stesso livello della cartella `wordpress`.

Listato 11.37 Esempio di Deployment di MySQL

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
  labels:
    app: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - image: mysql:8.0.32
          name: mysql
          env:
            - name: MYSQL_ROOT_PASSWORD

```

```

        valueFrom:
secretKeyRef:
    name: mysql-cred
key: password
ports:
    - containerPort: 3306
        name: mysql
        volumeMounts:
            - name: mysql-persistent-storage
                mountPath: /var/lib/mysql
volumes:
    - name: mysql-persistent-storage
        emptyDir: {}

```

Listato 11.38 Esempio di Service di MySQL

```

apiVersion: v1
kind: Service
metadata:
    name: mysql
    labels:
        app: mysql
spec:
    ports:
        - port: 3306
    selector:
        app: mysql

```

Listato 11.39 Esempio di Secret di MySQL

```

apiVersion: v1
kind: Secret
metadata:
    name: mysql-cred
type: Opaque
data:
    # Default password: "admin".
    password: YWRtaW4=

```

Come prima, creiamo un file `kustomization.yaml`, dove indichiamo le risorse che vogliamo vengano eseguite.

Listato 11.40 Esempio di file mysql/kustomization.yaml

```

resources:
- deployment.yaml
- service.yaml
- secret.yaml

```

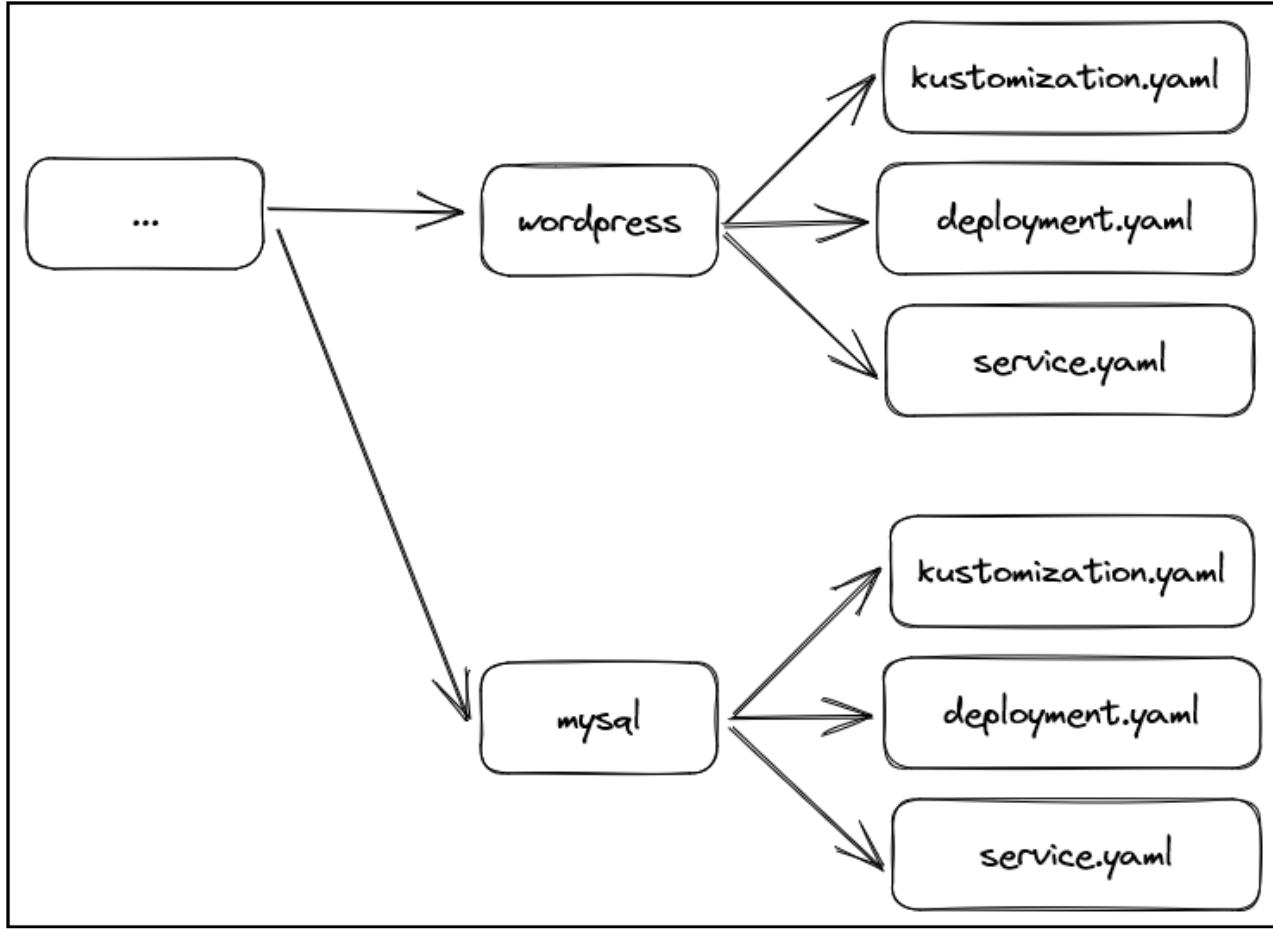


Figura 11.7 Struttura complessiva dell'installazione di Wordpress con un database MySQL.

Nel file `kustomization.yaml` precedente nella cartella Wordpress rimuoviamo il campo `commonLabels` (perché non vogliamo che vengano messe alle risorse relative a Wordpress) e creiamo un'ulteriore cartella, chiamata `base`: questa dovrà includere quelle precedenti e un file `kustomization.yaml`, dove descriveremo le due cartelle da utilizzare e anche le label da applicare a tutte le risorse.

Listato 11.41 Esempio di file base/kustomization.yaml

```

commonLabels:
  app: my-wordpress
bases:
- ./wordpress
- ./mysql

```

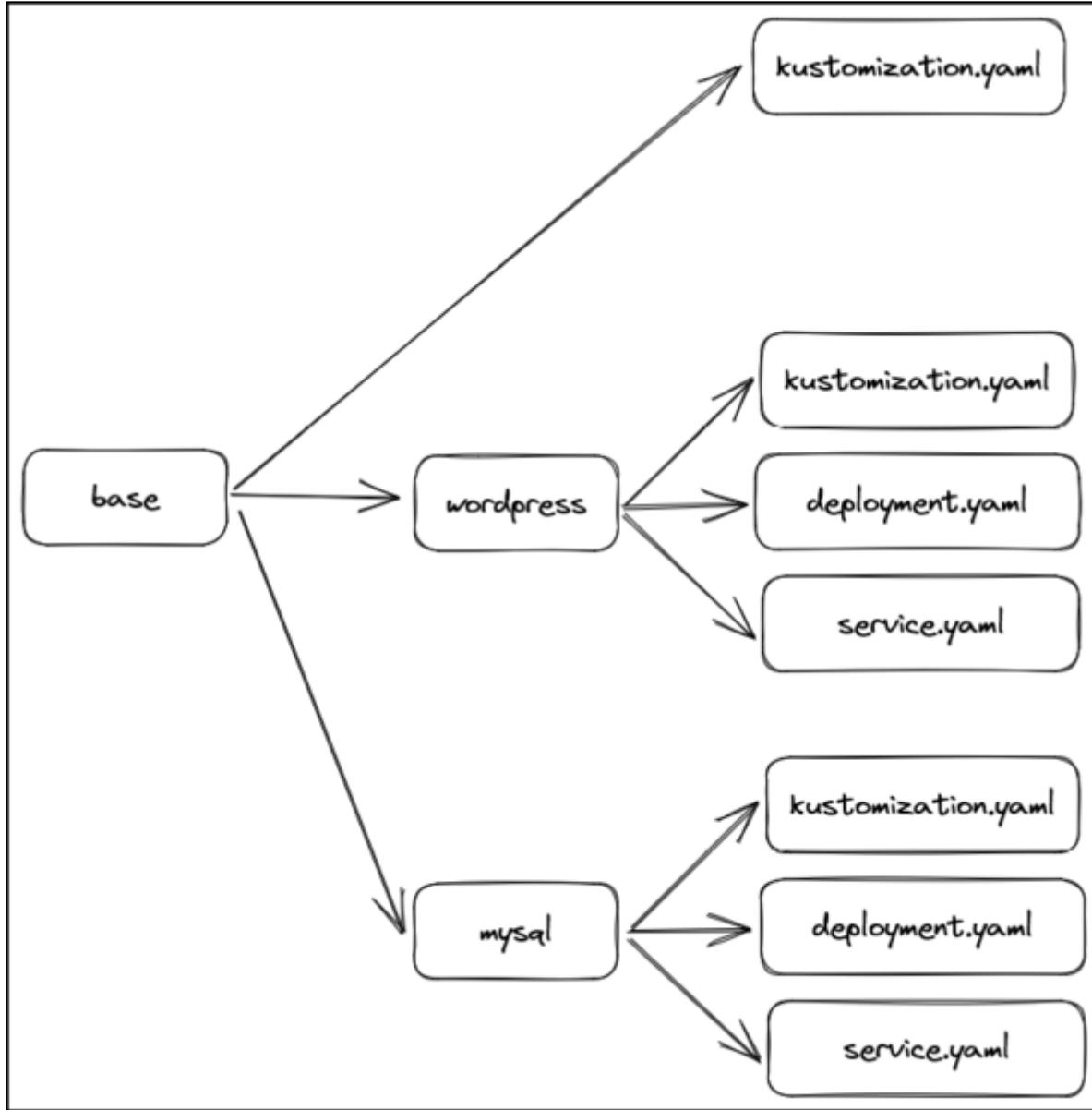


Figura 11.8 Aggiornamento della struttura precedente.

Ora che abbiamo “spostato” le etichette in questo file `kustomization.yaml` principale e definito le due directory di base con cui stiamo lavorando, se eseguiamo `kustomize build`, possiamo vedere che tutti i file sono assemblati e la label è stata aggiunta correttamente a tutte le risorse.

Listato 11.42 Output del comando `kustomize build`

```

apiVersion: v1
data:
  password: YWRtaW4=
kind: Secret
metadata:
  labels:
    app: my-wordpress
  name: mysql-cred
type: Opaque
---
apiVersion: v1
kind: Service

```

```

metadata:
  labels:
    app: my-wordpress
  name: mysql
spec:
  ports:
  - port: 3306
  selector:
    app: my-wordpress
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: my-wordpress
  name: wordpress
spec:
  ports:
  - port: 80
  selector:
    app: my-wordpress
  type: LoadBalancer
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: my-wordpress
  name: wordpress
spec:
  selector:
    matchLabels:
      app: my-wordpress
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: my-wordpress
    spec:
      containers:
      - image: wordpress:6.1.1-apache
        name: wordpress
      ports:
      - containerPort: 80
        name: wordpress
      volumeMounts:
      - mountPath: /var/www/html
        name: wordpress-persistent-storage
      volumes:
      - emptyDir: {}
        name: wordpress-persistent-storage
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: my-wordpress
  name: mysql
spec:
  selector:
    matchLabels:
      app: my-wordpress
  strategy:
    type: Recreate

```

```

template:
  metadata:
    labels:
      app: my-wordpress
  spec:
    containers:
      - env:
          - name: MYSQL_ROOT_PASSWORD
            valueFrom:
              secretKeyRef:
                key: password
                name: mysql-cred
            image: mysql:5.6
            name: mysql
        ports:
          - containerPort: 3306
            name: mysql
        volumeMounts:
          - mountPath: /var/lib/mysql
            name: mysql-persistent-storage
    volumes:
      - emptyDir: {}
        name: mysql-persistent-storage

```

Dopo aver verificato che il risultato prodotto sia quello che volevamo ottenere, possiamo applicare queste modifiche tramite il comando `kubectl apply`, di modo che le risorse vengano aggiunte al cluster. In questo caso, mentre nel primo comando eseguiamo la `build` con Kustomize e poi passiamo esplicitamente tramite la `pipe`(il simbolo “|”) il risultato a `kubectl` come fosse un file, il parametro `-k` del secondo comando permette di utilizzare Kustomize per eseguire la build e passare l’output a `kubectl`, che lo utilizzerà per creare le risorse:

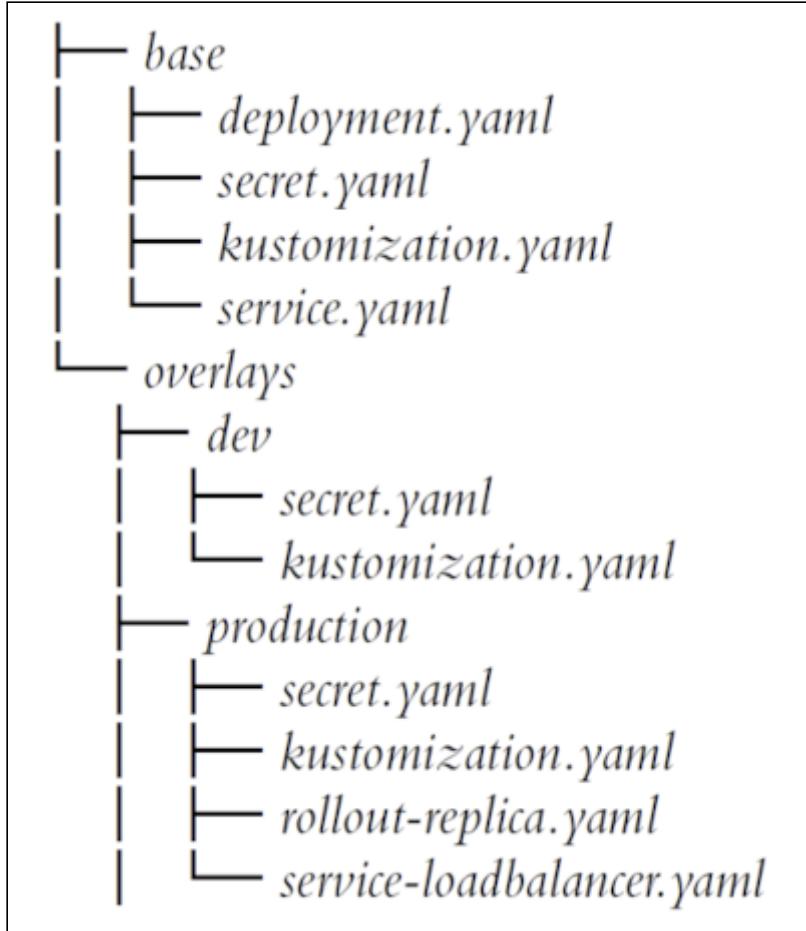
Listato 11.43 Esempio di applicazione delle risorse kustomizzate

```

kustomize build base/ | kubectl apply -f -
# oppure
kubectl apply -k base/

```

Questo è solo uno dei tanti esempi di funzionalità che ci permettono, attraverso Kustomize, di lavorare con i file YAML che descrivono delle risorse Kubernetes; come visto anche attraverso gli esempi, un team di sviluppo può prendere diversi file e aggiornare i singoli componenti a seconda dei diversi contesti, mantenendo una base consolidata e separata dalle personalizzazioni che si vuole applicare. Oltre all’utilizzo di una *baseline*, Kustomize permette anche la definizione di livelli di patch da applicare a specifiche risorse: in questo caso, sfruttiamo delle risorse per eseguire il cosiddetto *overlay*, ossia dei file che descrivono risorse specifiche a seconda delle configurazioni o dello stage di sviluppo in cui ci troviamo. Prendiamo un caso d’uso reale: immaginiamo di avere un’applicazione che viene descritta tramite un Deployment, un Secret e un Service, che costituiscono quindi la sua base; a seconda che l’applicazione sia rilasciata per eseguire dei test di sviluppo (che chiameremo `dev`), oppure per metterla a disposizione dell’utente finale (che chiameremo `prod`), possiamo creare una cartella aggiuntiva a quella base, chiamata `overlays`. Qui inseriremo tutti file che devono personalizzare (o *kustomizzare*) l’applicazione: avremo quindi 2 sottocartelle di `base`, una per ogni stage di sviluppo, con le relative risorse.



Per esempio, se volessimo modificare la password da utilizzare per l'applicazione, potremmo utilizzare la seguente definizione: il primo file rispecchierà la password utilizzata di default in fase di build del progetto, mentre il secondo andrà a definire la versione “patchata”, ossia le modifiche da applicare alla risorsa Secret.

Listato 11.44 File YAML base/secret.yaml

```

apiVersion: v1
kind: Secret
metadata:
  name: mysql-cred
type: Opaque
data:
  # Default password: "admin".
  password: YWRtaW4=

```

Listato 11.45 File YAML dev/secret.yaml

```

apiVersion: v1
kind: Secret
metadata:
  name: mysql-cred
type: Opaque
data:
  # Password: mydevpassword
  password: bXlkZXZwYXNzd29yZA==

```

Il seguente esempio mostra in che modo Kustomize dovrà utilizzare questo file per applicare alla configurazione di base la patch, utilizzando una strategia `patchesStrategicMerge`, che consente di

definire e sovrapporre file YAML parziali sopra quelli presenti nella base: ciò significa che verranno “calcolate” le differenze tra i due file e verrà applicata solo la/le differenza/e presente/i nel file sotto `dev` al file di base.

Listato 11.46 File YAML `dev/kustomization.yaml`

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
bases:
- ../../base
patchesStrategicMerge:
- secret.yaml
```

A differenza di quanto visto prima, la `build` attraverso Kustomize andrà eseguita specificando le cartelle da utilizzare per eseguire l'overlay, così che le patch specificate nei diversi file vengano correttamente applicate:

Listato 11.47 Esempio di applicazione degli overlay alla baseline

```
kustomize build overlays/dev | kubectl apply -f -
# oppure, se usi kubectl, non è necessario specificare il comando build e basta usare
kubectl kustomize
kubectl apply -k overlays/dev
```

Helm vs Kustomize

Ora che i due strumenti sono stati illustrati, potrebbe sorgere la domanda: quando devo usare uno o l'altro, oppure esiste uno strumento migliore? In realtà, si tratta di una scelta estremamente personale, e molto spesso non si tratta neanche di decidere quale utilizzare, ma in che modo sfruttare entrambi per uno stesso flusso di lavoro. Facciamo un piccolo riassunto delle puntate precedenti, e rivediamo la differenza tra la programmazione *dichiarativa* e quella *imperativa*; mentre la prima descrive un approccio dove, in Kubernetes, i file YAML che descrivono le risorse racchiudono anche il modo in cui queste verranno rilasciate nel cluster, nel secondo paradigma abbiamo la possibilità di descrivere i cambiamenti di stato che le risorse potranno avere a seconda di alcune condizioni. Già attraverso queste due definizioni, possiamo distinguere e associare Helm a un approccio imperativo, mentre Kustomize sposa in pieno l'approccio dichiarativo, come d'altronde fa Kubernetes nativamente. Ma quali sono i vantaggi di utilizzo di uno strumento piuttosto che l'altro?

Helm utilizza diversi file YAML per creare il *manifest* finale, il che significa che potrebbero esserci molte parentesi e pezzi di file non-YAML, che servono a esprimere della logica. Infatti, l'idea dietro al chart è quella di utilizzare questo oggetto come unico per il rilascio del software e di sfruttare le restanti risorse per aggiungere della parametrizzazione o della logica condizionale ai diversi file. Questa è una funzionalità che con Kustomize non è possibile avere: le modifiche sono applicate su una base specifica, e non su espressioni che vengono risolte in fase di build. In Kustomize non ci sono istruzioni condizionali, o iterative che ne permettono una maggior complessità; questo significa anche avere a che fare con un livello di astrazione più alto, per cui dovremo lavorare con diversi file in diverse cartelle, e probabilmente anche con diversi chart. “Da un grande potere derivano grandi responsabilità”, per cui con Helm abbiamo maggior libertà di manovra sulla logica da applicare ai nostri file, ma questo significa anche una maggior gestione; non a caso, Helm ha una curva di apprendimento piuttosto ripida e può volerci un po' per abituarsi.

Kustomize è uno strumento dichiarativo, che funziona direttamente attraverso i file YAML, e racconta un po' la stessa storia del comando `sed` che si trova nei sistemi Unix, che applica una sostituzione “a

comando” a determinate stringhe: “attraversa” una risorsa Kubernetes per aggiungere, rimuovere o aggiornare le opzioni di configurazione direttamente nelle loro definizioni. Il vantaggio è sicuramente l’integrazione in Kubernetes, che ne permette l’utilizzo con `kubectl` e quindi la creazione anche di flussi di integrazione e rilascio continuo senza dover aggiungere strumenti alla nostra cassetta degli attrezzi; non è inoltre complesso da utilizzare, perché la sua semplicità di approccio rispecchia anche la filosofia adottata da Kubernetes per la definizione delle sue risorse. Lo svantaggio è sicuramente nella possibilità di applicare della logica: qui parliamo infatti di modifiche nel senso di patch, quindi definizioni che vengono aggiornate secondo delle regole ferree (vedi la strategia di *merge* applicata nell’esempio precedente), piuttosto che la possibilità di poter esprimere delle condizioni, come con Helm.

Tuttavia, è doveroso dire che spesso entrambi gli strumenti vengono utilizzati come parte integrante di un flusso più grande, dove ognuno ha la propria responsabilità.

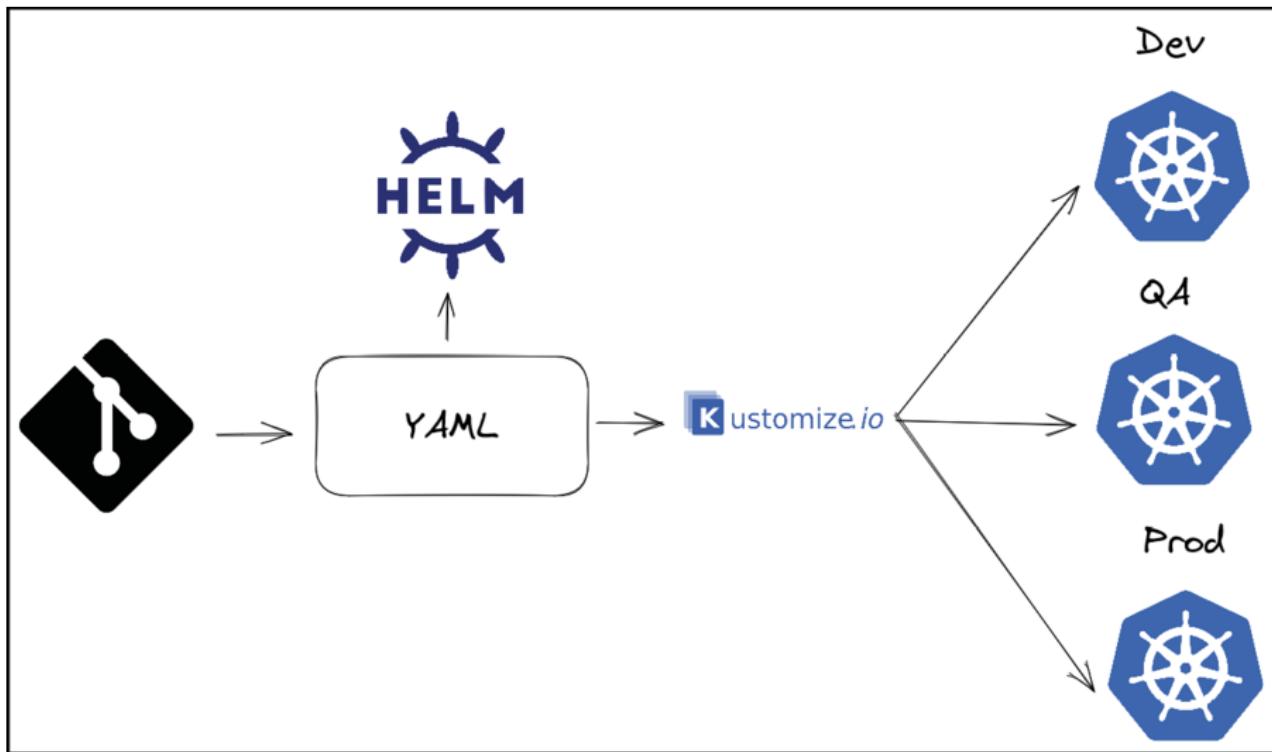


Figura 11.9 Come utilizzare Helm e Kustomize in uno scenario con diversi ambienti di rilascio.

Il diagramma in Figura 11.9 mostra un caso d’uso comune di un flusso di lavoro di esempio per integrare Kustomize all’interno del processo di sviluppo e rilascio di un prodotto, iniziando con un evento Git. L’evento può essere un `push`, il `merge` del codice o la creazione di un nuovo ramo. In questo caso, Helm viene utilizzato per generare i file yaml e Kustomize correggerà con valori specifici dell’ambiente in base agli eventi. Per esempio: se il ramo è `master` ed è legato all’ambiente di produzione, allora Kustomize applicherà i valori applicabili alla produzione.

Per riassumere, uno è un motore di *template*, l’altro un motore di *patch* o *overlay*: quando devi scegliere quale strumento utilizzare, e in che occasione, dovresti chiederti se il tuo obiettivo è quello di rendere le cose più semplici, sia in termini di complessità lavorativa quotidiana, sia in termini di quanta complessità stai aggiungendo al tuo lavoro in un dato momento. Potrebbe arrivare un momento in cui capiremo che dovremo far lavorare insieme Kustomize e Helm, come nella figura precedente: questa è una possibilità; potremo anche dover utilizzare l’uno piuttosto che l’altro per esigenze aziendali. L’esigenza stabilisce lo strumento, e non il contrario!

Che cosa abbiamo imparato

- Che cos'è Helm, come funziona e quali approcci possiamo utilizzare per aggiungere della logica ai nostri file YAML.
- Che cos'è Kustomize, come si usa e come possiamo eseguire delle patch sui file che descrivono le risorse Kubernetes.
- Quali sono le differenze tra i due strumenti, e quando adottare uno (o entrambi).

Operatori

Dobbiamo far emergere il potenziale di tutte le persone, il che significa che dobbiamo avere un'attenzione particolare e individualizzata su ogni persona.

– Anne Chow, ex-CTO AT&T, Consigliera @ 3M

In un contesto in cui i servizi stateless condividono lo stesso spazio di quelli stateful, abbiamo potuto apprezzare le loro differenze e il loro livello di applicabilità. Gli *operatori* si inseriscono all'interno del dominio in quanto strumento fondamentale per quei casi d'uso in cui gestire degli applicativi che hanno una propria identità e persistenza è fondamentale: come abbiamo avuto modo di vedere brevemente con gli esempi fatti in precedenza, e come vedremo in seguito, le applicazioni stateful sono indicate per quelle situazioni in cui si vuole poter mantenere il loro stato e la relativa coerenza di dati. Un esempio tipo di StatefulSet è proprio quello relativo a un database, dove i dati sono il cuore pulsante e la loro persistenza con eventuali repliche per resistere anche agli errori è fondamentale. Gli operatori, grazie alla loro definizione, rendono altamente personalizzabili quelle risorse che non possono essere semplicemente “riavviate” in caso di guasti, ma la cui gestione dev'essere associata a un agente esterno, che possa governarle e coordinarle in maniera opportuna.

Che cosa sono

Per dare un'idea delle funzionalità relative agli operatori, e della loro definizione, partiamo da un contesto in cui si lavori solo con applicazioni stateless: come abbiamo visto in precedenza, possiamo facilmente immaginare di creare un Deployment con diverse repliche di Pod insieme a un Service per la comunicazione con l'esterno e una ConfigMap che contenga la relativa configurazione. Se una di queste repliche fosse arrestata, il ReplicationController si occuperà di sostituire immediatamente la copia mancante con una nuova, grazie al meccanismo di controllo che viene eseguito in loop. Nel caso di un aggiornamento, è possibile modificare la configurazione del Deployment, e i Pod saranno automaticamente riavviati per poter applicare correttamente le modifiche. Tutto questo avviene perché Kubernetes controlla periodicamente se lo stato desiderato corrisponde a quello reale e attuale, per poter operare di conseguenza: abbiamo parlato infatti nei primi capitoli del ruolo dello scheduler e di come questo oggetto comunichi con le API per poter agire tempestivamente in caso lo stato del cluster risulti alterato rispetto a quanto presente nel database di etcd.

Quando abbiamo a che fare con applicazioni stateful, è necessario anche valutare la gestione della persistenza dei dati: al loro riavvio, non è sempre sufficiente un processo automatico di recupero dei dati o di gestione del loro ciclo di vita. Se prendiamo come esempio il database PostgreSQL, e decidiamo di avviare un'istanza di questo strumento tramite uno StatefulSet, inserendo un numero di repliche pari a 3, sappiamo già che queste non saranno perfettamente identiche, ma che ognuna di esse avrà il suo stato e la relativa identità.

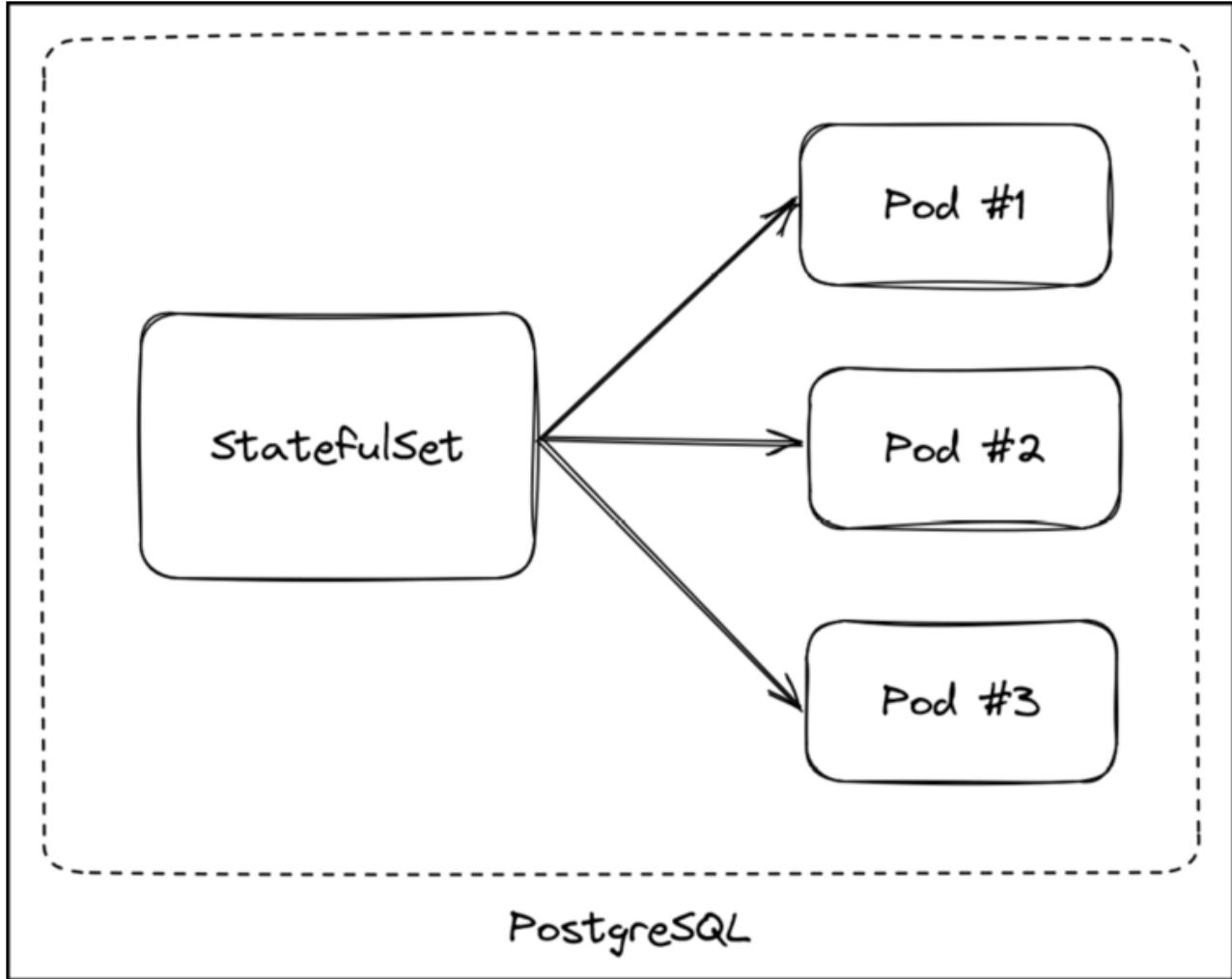


Figura 12.1 StatefulSet per PostgreSQL.

Proprio per questa caratteristica, questo vuol dire che il loro riavvio o il loro aggiornamento deve essere eseguito seguendo un ordine ben preciso, così come i dati al loro interno dovranno essere in qualche modo “sincronizzati”, per far sì che siano coerenti; pensiamo a un database con 3 repliche, di cui una è quella primaria: se questa venisse arrestata, in che modo una delle secondarie dovrebbe prendere il suo posto? Tutti questi dettagli implementativi dipendono molto dal caso d’uso e dal tipo di strumento utilizzato; un database come MySQL, relazionale a tabella, si comporterà in modo diverso da uno come MongoDB, basato su documenti e collezioni, e ognuno di questi avrà dei processi interni differenti per mantenere le informazioni in uno stato congruente. Da queste riflessioni, emerge quindi l’impossibilità di avere un solo approccio per gestire situazioni in cui applicazioni come queste sono strutturalmente diverse, e che spesso richiedono un intervento manuale: semplificare il lavoro automatizzando parte di questi processi, attraverso un unico punto di raccordo, diventa quindi una priorità. Parlare di “intervento manuale” stona un po’ con le funzionalità che Kubernetes offre, tra cui c’è la gestione automatica di aspetti come la resilienza delle applicazioni ai guasti.

Un *operator* serve proprio a rendere automatiche una serie di operazioni che normalmente richiederebbero un intervento umano, e non solo; un esempio classico di come questa risorsa torni particolarmente utile riguarda il caso in cui si abbia a che fare con più di un cluster da gestire e manutenere, rendendo allineati i due ambienti. In questo caso, avere un oggetto che descriva il processo da applicare per sincronizzare i due sistemi in maniera automatica assume un’importanza

fondamentale. Per comprendere come questi funzionino, riprendiamo il concetto di *control loop*, ossia quel meccanismo con cui le risorse che gestiscono il cluster verificano periodicamente lo stato di un'entità e se ci sono differenze che richiedono un'azione. Un operatore è in grado di assolvere a tutti i compiti descritti prima, come aggiornare le immagini utilizzate dai Pod, piuttosto che sostituire una replica che è stata terminata, senza compromettere la funzionalità e l'integrità dell'applicazione.

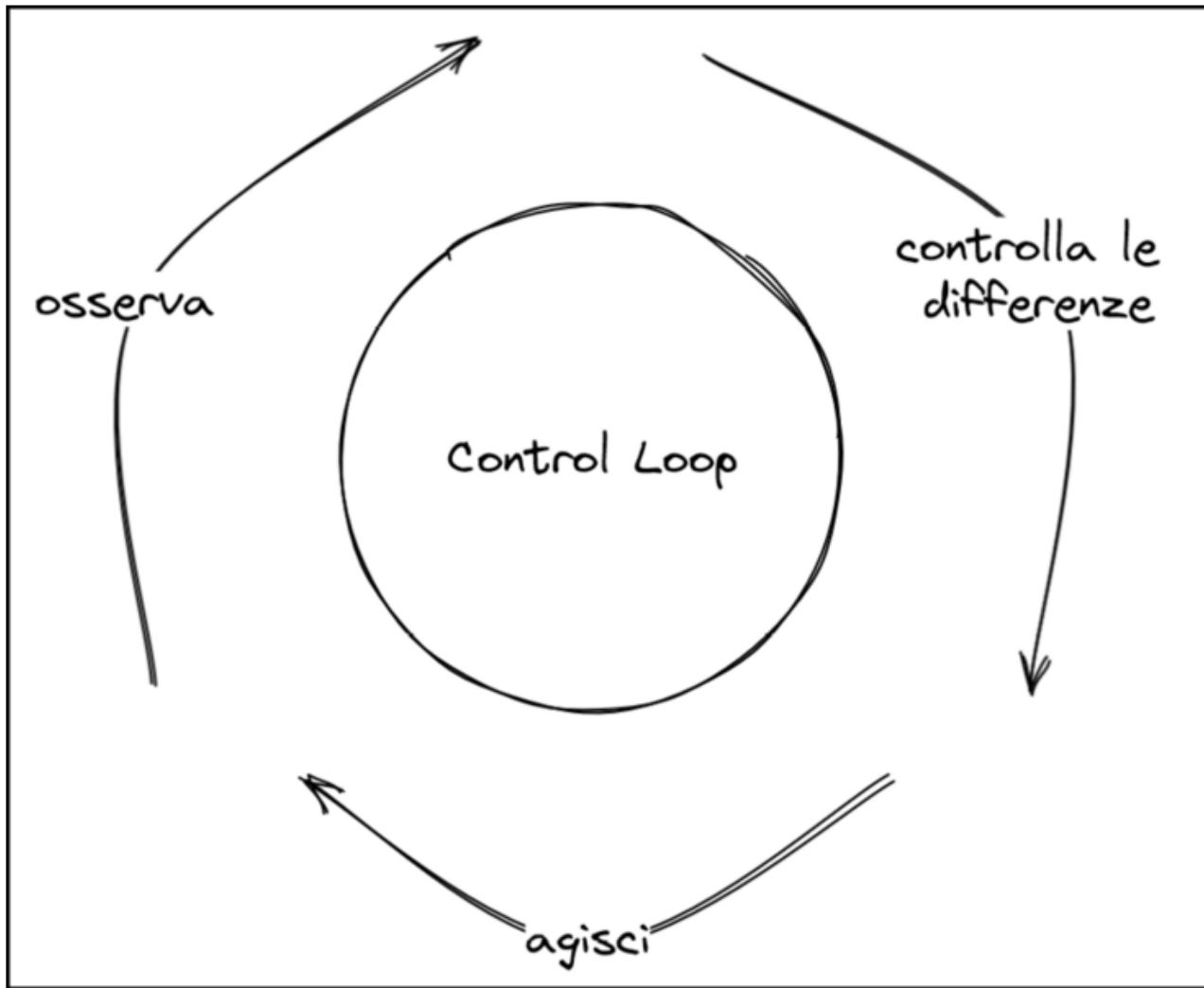


Figura 12.2 Come funziona il meccanismo di control loop.

Questo tipo di risorse si basa sul concetto di CRD, ovvero *Custom Resource Definition*: si tratta semplicemente di un componente Kubernetes *personalizzato* la cui definizione non fa parte delle API standard (o meglio, *core*), ma che hanno una serie di caratteristiche proprie. Queste risorse vengono create a partire da quelle che conosciamo, come StatefulSet, Deployment e via dicendo, per poter aggiungere della logica applicativa in più. Un operatore prende quindi queste risorse, insieme al meccanismo di controllo dello stato delle entità in esecuzione, e le unisce con il dominio specifico delle funzionalità di un certo servizio, per poter fornire un'applicazione completa anche una serie di automatismi di tutti i suoi processi interni. La creazione di questi oggetti per una serie di tecnologie è demandata ai team dietro al servizio stesso: possiamo infatti aspettarci che, se dovessimo aver bisogno di creare un cluster MongoDB su Kubernetes e di avere quindi più repliche primarie e secondarie, le persone che sviluppano questo sistema abbiano messo a disposizione della community un operatore in grado di gestire anche le funzionalità più complesse, come la creazione di un cluster,

come sincronizzarlo o come eseguire un backup. Molti di questi operatori sono disponibili all'interno di un portale specifico, OperatorHub.io, dove le tecnologie più note sono presenti con la relativa documentazione per l'installazione di questi oggetti.

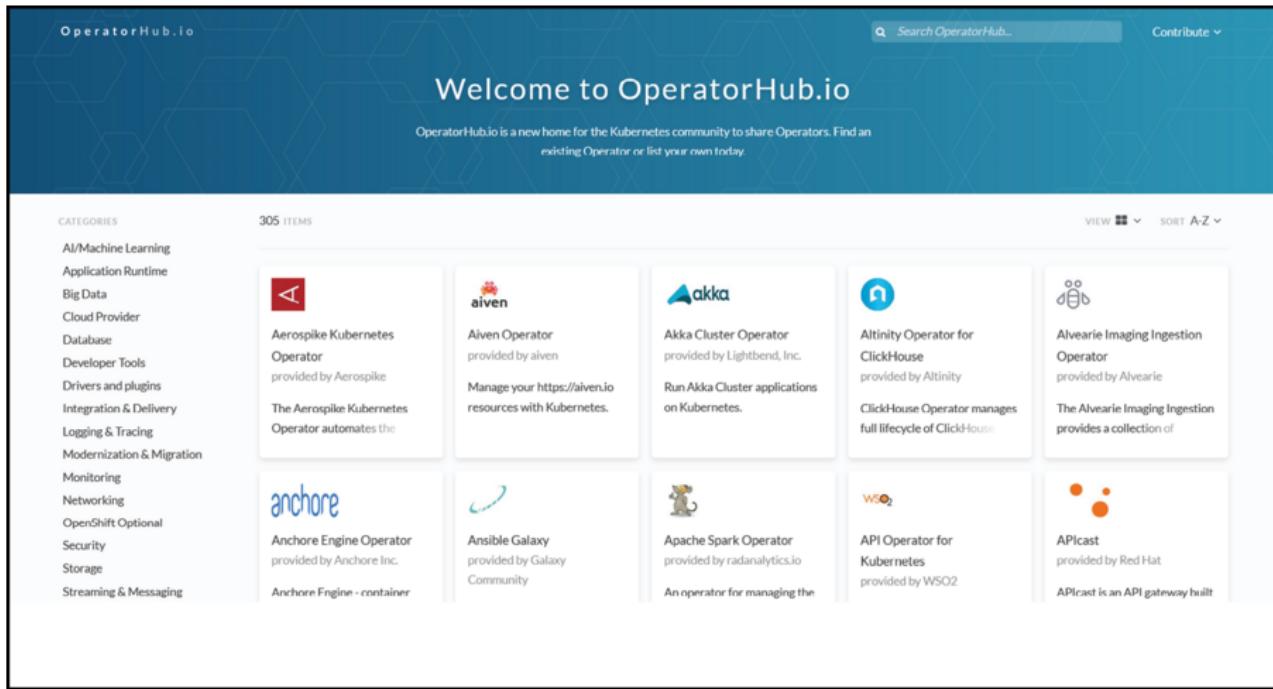


Figura 12.3 Pagina principale di OperatorHub.io.

Questo non significa che non si possa creare il proprio operatore: esiste infatti un SDK che permette alle persone esperte in sviluppo insieme a quelle che conoscono il dominio di creare un operatore su misura per la propria applicazione, e quindi di estendere ciò che Kubernetes offre all'interno del suo portfolio. Tramite il sito <https://sdk.operatorframework.io/>, è possibile scaricare il materiale necessario per creare il proprio operatore, insieme alla documentazione che dettaglia quali sono le modalità per farlo. Non sarà una sorpresa scoprire che, tra gli strumenti utilizzati per poter far nascere uno di questi oggetti, ci siano Go, Java, Ansible, ma soprattutto Helm: tutti questi permettono un alto livello di personalizzazione e soprattutto di automazione dei processi che descrivono. Questo vuol dire, partendo da dei campi *custom* che definiscono le nostre risorse e come vogliamo che siano descritte, e un meccanismo di controllo, sarà possibile creare un *operator*.

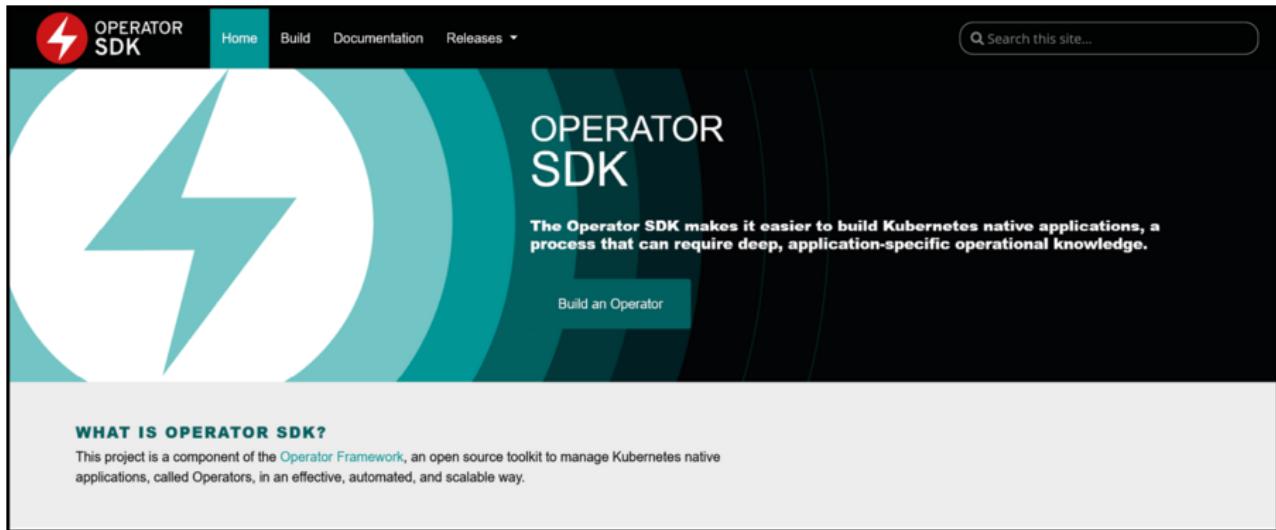


Figura 12.4 Pagina principale dell'Operator SDK.

Come funziona

Ora che abbiamo definito come nascono gli operatori e qual è il loro scopo, potrebbe sembrare che questo tipo di oggetti sia complicato da utilizzare. In realtà, il loro uso non differisce molto dalle risorse “tradizionali”: questi possono essere definiti tramite un file YAML, che contiene le loro proprietà, e che avvierà una serie di oggetti all’interno del cluster Kubernetes. La differenza sta nelle API: se prendessimo, per esempio, Memcached, noteremmo che, nella prima parte della sua specifica, una risorsa di questo tipo utilizza un’API diversa:

Listato 12.1 Esempio parziale di una risorsa Memcached

```
apiVersion: cache.example.com/v1alpha1
kind: Memcached
metadata:
  name: memcached-sample
spec:
  size: 3
...
```

Questa definizione fa dunque riferimento a una risorsa *custom* che contiene tutti i dettagli sui campi che questo oggetto può avere, di che tipo, ed eventualmente la descrizione: un esempio parziale della `CustomResourceDefinition` di Memcached è il seguente: all’interno del campo `validation` vengono descritti tutti i campi che possono essere utilizzati per descrivere un’istanza di Memcached, e molti di questi li abbiamo già incontrati con altre risorse. È il caso di `apiVersion`, `kind` e `metadata`, che sono comuni a tutti gli oggetti con cui abbiamo lavorato.

Listato 12.2 Esempio parziale della CustomResourceDefinition di Memcached

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: memcacheds.cache.example.com
spec:
  group: cache.example.com
  names:
    kind: Memcached
    listKind: MemcachedList
```

```

plural: memcacheds
singular: memcached
scope: Namespaced
subresources:
  status: {}
validation:
  openAPIV3Schema:
    description: Memcached is the Schema for the memcacheds API
  properties:
    apiVersion:
      description: 'APIVersion defines the versioned schema of this representation
                    of an object. Servers should convert recognized schemas to the latest
                    internal value, and may reject unrecognized values. More info:
                    https://git.k8s.io/community/contributors/devel/api-conventions.md#resources'
      type: string
    kind:
      description: 'Kind is a string value representing the REST resource this
                    object represents. Servers may infer this from the endpoint the client
                    submits requests to. Cannot be updated. In CamelCase. More info:
                    https://git.k8s.io/community/contributors/devel/api-conventions.md#types-kinds'
      type: string
    metadata:
      type: object
    spec:
      description: MemcachedSpec defines the desired state of Memcached
      properties:
        size:
          description: 'INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
                        Important: Run "operator-sdk generate k8s" to regenerate code after
                        modifying this file Add custom validation using kubebuilder tags:
                        https://book-v1.book.kubebuilder.io/beyond\_basics/generating\_crd.html'
          format: int32
          type: integer
        required:
        - size
      type: object
...

```

Nel caso, invece, di proprietà specifiche di quella risorsa, quasi sempre dovremo fare riferimento al campo `spec`, il quale ci permetterà di fornire dei parametri alla risorsa `custom`: nel caso di Memcached, all'interno di questo troveremo un attributo chiamato `size`, che serve per descrivere la dimensione del cluster (campo `description` con le relative informazioni), che dovrà essere di tipo intero (campo `format` e `type` per definirlo) e sarà obbligatorio (campo `required`). La parte interessante sta nel fatto che questa CRD sia stata definita da qualcun'altro al posto nostro e che, per utilizzarla, non ci richiederà molto sforzo: utilizzando un esempio simile a quanto visto in precedenza, sarà sufficiente utilizzare il comando `kubectl apply` per creare le risorse che definiscono un'istanza Memcached. Questo vuol dire che anche tutti i comandi visti finora per gestire le risorse del cluster restano validi; è possibile cancellare un oggetto creato a partire da una CRD, così come aggiornarlo o crearne di nuovi.

Esempi

Per installare un operatore, che sia presente su OperatorHub.io o su altre piattaforme, avremo bisogno di configurare anche l'oggetto *Operator Lifecycle Manager* (abbreviato in OLM), un componente di Operator Framework, che deve essere presente nel cluster. OLM rende gli operatori disponibili per l'installazione da parte degli utenti in base al concetto di *cataloghi*, che sono repository di operatori pacchettizzati per l'utilizzo con OLM. Questo vuol dire che, quando avremo installato OLM, ogni volta che vorremo provare un nuovo operatore potremo sfruttare questa risorsa per aviarla con pochi e semplici comandi.

Un modo rapido per installare OLM su un cluster Kubernetes con le impostazioni predefinite consiste nell'eseguire questo comando:

Listato 12.3 Configurazione di OLM

```
kubectl create -f https://raw.githubusercontent.com/operator-framework/operator-lifecycle-
manager/master/deploy/upstream/quickstart/crds.yaml
>>>

customresourcedefinition.apiextensions.k8s.io/catalogsources.operators.coreos.com created
customresourcedefinition.apiextensions.k8s.io/clusterserviceversions.operators.coreos.com
created
customresourcedefinition.apiextensions.k8s.io/installplans.operators.coreos.com created
customresourcedefinition.apiextensions.k8s.io/operatorconditions.operators.coreos.com
created
customresourcedefinition.apiextensions.k8s.io/operatorgroups.operators.coreos.com created
customresourcedefinition.apiextensions.k8s.io/operators.operators.coreos.com created
customresourcedefinition.apiextensions.k8s.io/subscriptions.operators.coreos.com created

kubectl create -f https://raw.githubusercontent.com/operator-framework/operator-lifecycle-
manager/master/deploy/upstream/quickstart/olm.yaml
>>>
namespace/olm created
namespace/operators created
serviceaccount/olm-operator-serviceaccount created
clusterrole.rbac.authorization.k8s.io/system:controller:operator-lifecycle-manager created
clusterrolebinding.rbac.authorization.k8s.io/olm-operator-binding-olm created
deployment.apps/olm-operator created
deployment.apps/catalog-operator created
clusterrole.rbac.authorization.k8s.io/aggregate-olm-edit created
clusterrole.rbac.authorization.k8s.io/aggregate-olm-view created
operatorgroup.operators.coreos.com/global-operators created
operatorgroup.operators.coreos.com/olm-operators created
clusterserviceversion.operators.coreos.com/packageserver created
catalogsource.operators.coreos.com/operatorhubio-catalog created
```

Questo eseguirà l'avvio di OLM, che consiste in due operatori: `catalog-operator` e `olm-operator`, due Pod che saranno in esecuzione su un namespace chiamato `olm`. Ognuno dei due comandi va a creare una serie di risorse note: nel primo caso, vengono aggiunte le risorse CRD a quelle disponibili nelle API di Kubernetes, così che se ne persista la definizione in etcd per permetterne poi la gestione. Tramite il secondo comando, vengono invece create le entità relative al suo funzionamento, come i ruoli, il namespace dove operare, i Deployment dei due Pod e le risorse custom, tra cui `catalogsource`.

Una volta terminato, OLM sarà pronto per essere utilizzato.

Percona per MongoDB

Percona è un'azienda che si occupa di software dal 2006: nasce in America e si occupa soprattutto del mondo legato ai database, come MariaDB, MySQL e InnoDB. In questo frangente, hanno prodotto diversi strumenti utili alla loro gestione e spesso rilasciati come software open source, tra cui *Percona Server for MongoDB*: si tratta di un operatore che permette di avviare una soluzione enterprise di un cluster MongoDB. Questa risorsa è stata creata seguendo le *best practice* definite dal team di sviluppo per il suo rilascio e configurazione così da mettere in esecuzione un cluster a più repliche con il supporto a funzionalità come il backup programmato o manuale, lo sharding e la gestione della rotazione delle credenziali. Anche il team di Percona ha utilizzato l'Operator SDK messo a disposizione della community per creare questo prodotto.

Per installarlo, è possibile utilizzare sia Helm che `kubectl`: tramite i comandi messi a disposizione nella documentazione ufficiale, è possibile installare sia l'operatore, che il server Percona. Partiamo da

questo semplice caso d'uso: installiamo l'operatore tramite il comando seguente.

Listato 12.4 Configurazione di Percona

```
helm install my-op percona/psmdb-operator

# oppure

kubectl apply --server-side -f https://raw.githubusercontent.com/percona/percona-server-mongodb-operator/main/deploy/bundle.yaml
>>>
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbbackups.psmdb.percona.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbrestores.psmdb.percona.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/perconaservermongodbs.psmdb.percona.com serverside-applied
role.rbac.authorization.k8s.io/percona-server-mongodb-operator serverside-applied
serviceaccount/percona-server-mongodb-operator serverside-applied
rolebinding.rbac.authorization.k8s.io/service-account-percona-server-mongodb-operator serverside-applied
deployment.apps/percona-server-mongodb-operator serverside-applied
```

Questo andrà a configurare l'operatore, costituito da un Deployment con un Pod, insieme ad alcuni ruoli che ne consentono l'esecuzione: sarà questo oggetto a gestire l'intero ciclo di vita delle risorse relative a Percona e quindi delle istanze che andremo a creare. A questo punto, possiamo procedere con l'installazione del server Percona: per eseguire un caso d'uso di esempio, è possibile utilizzare il seguente file, che descrive un cluster minimale per MongoDB: questo utilizza la risorsa `PerconaServerMongoDB`, che abbiamo importato in precedenza per creare un cluster chiamato `my-cluster` a partire dall'immagine fornita, con una sola replica e con lo *sharding* attivo.

Listato 12.5 Cluster Percona per MongoDB

```
apiVersion: psmdb.percona.com/v1
kind: PerconaServerMongoDB
metadata:
  name: my-cluster
spec:
  crVersion: 1.14.0
  image: percona/percona-server-mongodb:5.0.11-10
  allowUnsafeConfigurations: true
  upgradeOptions:
    apply: disabled
    schedule: "0 2 * * *"
  secrets:
    users: my-cluster
  replsets:
    - name: rs0
      size: 1
      volumeSpec:
        persistentVolumeClaim:
          resources:
            requests:
              storage: 1Gi
  sharding:
    enabled: true
    configsvrReplSet:
      size: 1
      volumeSpec:
        persistentVolumeClaim:
          resources:
            requests:
              storage: 1Gi
```

```
mongos:  
size: 1
```

Significato di sharding

Lo *sharding* è un tipo di partizionamento dei database che separa quelli di grandi dimensioni in parti più piccole, più veloci e più facilmente gestibili. Queste parti più piccole sono chiamate anche *frammenti di dati* o *shard*. La parola frammento significa “una piccola parte di un tutto”.

Per creare questo cluster, utilizziamo il comando `kubectl apply` e noteremo che, come nel caso dell'installazione dell'operatore, verrà creata una risorsa `custom`.

Listato 12.6 Configurazione di un cluster Percona

```
kubectl apply -f my-cluster.yaml  
>>>  
perconaservermongodb.psmdb.percona.com/my-cluster created  
  
kubectl get pods  
>>>  
NAME                               READY   STATUS    RESTARTS   AGE  
my-cluster-cfg-0                  1/1     Running   0          39m  
my-cluster-mongos-0                1/1     Running   0          38m  
my-cluster-rs0-0                  1/1     Running   0          38m  
percona-server-mongodb-operator-65977f75d5-9rdm9  1/1     Running   0          39m
```

Come visibile dall'output, sono state create (oltre al Pod dell'operatore) diverse risorse: una chiamata `my-cluster-rs0-0`, che corrisponde alla replica del cluster, in questo caso (per forza) primaria, essendo l'unica. Dal momento che è stato abilitato lo *sharding*, sarà necessario avere un componente che gestisca le richieste in ingresso per poter “smistare” i dati: è il caso di `mongos`, il secondo Pod creato. Questo lavora come un punto di accesso alla persistenza dei dati. Infine, il Pod `my-cluster-cfg-0` rappresenta il Pod di configurazione del server.

Ora che il cluster è funzionante, come collegarsi al database? Per farlo, è possibile recuperare le credenziali contenute all'interno del Secret che è stato creato all'avvio di questa risorsa e utilizzare il terminale di un client per MongoDB per collegarsi.

Listato 12.7 Secret del cluster Percona

```
kubectl get secret minimal-cluster -o yaml  
>>>  
apiVersion: v1  
data:  
  MONGODB_BACKUP_PASSWORD: cktDR0p0V1Y2eTFCdFoxOHZ2aQ==  
  MONGODB_BACKUP_USER: YmFja3Vw  
  MONGODB_CLUSTER_ADMIN_PASSWORD: QUFrZlpRSHRCd3hsWUVUYw==  
  MONGODB_CLUSTER_ADMIN_USER: Y2x1c3RlckFkbWlu  
  MONGODB_CLUSTER_MONITOR_PASSWORD: Tj10aENKQ3ZUcUdGYzVENGV3  
  MONGODB_CLUSTER_MONITOR_USER: Y2x1c3Rlck1vbml0b3I=  
  MONGODB_DATABASE_ADMIN_PASSWORD: cEdYR2R4R3BJTVRNckc3SVY=  
  MONGODB_DATABASE_ADMIN_USER: ZGF0YWJhc2VBZG1pbg==  
  MONGODB_USER_ADMIN_PASSWORD: RGd1Zk5DdXdxWU1vVzNsZWg=  
  MONGODB_USER_ADMIN_USER: dXNlckFkbWlu  
kind: Secret  
metadata:  
  name: my-cluster  
  namespace: default  
  type: Opaque
```

Come visibile nell'output precedente, all'interno della risorsa sono stati definiti diversi utenti con le relative password: quello chiamato `MONGODB_BACKUP_USER`, utilizzato per operazioni di backup e restore; `MONGODB_CLUSTER_ADMIN_USER`, che ha pieni poteri sul cluster come amministratore;

MONGODB_CLUSTER_MONITOR_USER, l'utente che ha i permessi per monitorare lo stato del cluster; MONGODB_DATABASE_ADMIN_USER e MONGODB_USER_ADMIN_USER, che sono gli utenti che hanno, rispettivamente, permessi di scrittura e lettura di ogni database insieme a backup e restore, e di sola gestione del proprio database.

Infine, nei comandi seguenti, utilizziamo un'immagine Percona come "ponte" per connetterci al cluster, e poi definiamo la stringa di connessione verso il cluster indicando lo username, la password e il nome del Service che permette di raggiungere il cluster, insieme al nome assegnato all'insieme di repliche dello stesso:

Listato 12.8 Connessione tramite client al cluster Percona

```
kubectl run -i --rm --tty percona-client --image=percona/percona-server-mongodb:4.4.16-16 --restart=Never -- bash -il

>>>
If you don't see a command prompt, try pressing enter.
[mongodb@percona-client /]$ 

mongo "mongodb+srv://userAdmin:DgufNCuwWYMoW3leh@my-cluster-rs0.default.svc.cluster.local/admin?replicaSet=rs0&ssl=false"
>>>

Percona Server for MongoDB shell version v4.4.16-16
connecting to: mongodb://my-cluster-rs0-0.my-cluster-rs0.default.svc.cluster.local:27017/admin?
compressors=disabled&gssapiServiceName=mongodb&replicaSet=rs0&ssl=false
Implicit session: session { `id` : UUID(<<4b709937-81fb-44ce-bc14-05087cf24f81>>) }
Percona Server for MongoDB server version: v5.0.11-10
WARNING: shell and server versions do not match
Welcome to the Percona Server for MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
    https://www.percona.com/doc/percona-server-for-mongodb
Questions? Try the support group
    https://www.percona.com/forums/questions-discussions/percona-server-for-mongodb
rs0:PRIMARY>
```

L'output ci mostra l'avvenuto collegamento al cluster e, in particolare, alla replica rs0 primaria, da cui potremo operare. Questo semplice e veloce esempio ci è servito a mostrare le potenzialità di questo strumento, ma soprattutto per evidenziare quali sono i benefici nell'utilizzare un operatore che possa includere tutta la gestione di un applicativo non solo al suo avvio, ma durante il suo ciclo di vita e all'occorrenza di specifiche funzionalità, che fanno parte della logica stessa del servizio.

Che cosa abbiamo imparato

- Questo capitolo è una breve introduzione su un tema estremamente complesso e potente: gli operatori sono strumenti molto utili per poter rendere giustizia al lavoro di gestione di applicazioni con diverse funzionalità, con le loro complessità e le loro personalizzazioni.
- Dopo una panoramica su che cosa siano gli operatori, abbiamo visto un breve esempio sfruttando l'OperatorHub e utilizzando Percona per MongoDB come caso di studio per avviare un cluster a più repliche.

Kubernetes: casi d'uso

Non mi piace la parola ‘equilibrio’. Per me, questo concetto in qualche modo evoca un conflitto tra lavoro e famiglia... finché pensiamo a queste cose come contrastanti, non saremo mai felici. La vera felicità viene dall'integrazione... del lavoro, della famiglia, di se stessi, della comunità.

– Padmasree Warrior, ex-CTO @ Cisco, CEO @ Fable

Questo capitolo è inteso come lo spazio perfetto per chi ha visto tutti gli esempi finora proposti, ma vuole andare oltre: se hai in mente un progetto che sfrutta Docker che vorresti poter eseguire su un cluster Kubernetes, oppure hai a disposizione un file `docker-compose.yml` e hai bisogno che funzioni anche su questo orchestratore, sei nel posto giusto. Con gli esempi che andremo a vedere, avrai modo di sperimentare di più dei casi d'uso reali, che ti portino a comprendere più nel pratico non solo gli oggetti visti finora, ma soprattutto come approcciarsi a un problema come quello descritto poco fa, dove lo stato dell'arte è che hai a disposizione qualcosa, ma non sei ancora nel punto in cui puoi decollare verso il tuo cluster.

Tutti i casi d'uso proposti fanno riferimento ad applicazioni conosciute, di modo che sia anche più semplice approcciarsi agli argomenti proposti, perché il cuore degli esempi che porteremo avanti è una logica mentale con cui poter affrontare un processo che sia di migrazione o anche di primo approccio al mondo di Kubernetes. Attraverso quello che vi verrà quindi proposto, vedremo come guardare diverse situazioni dalla prospettiva di un *software architect*, per comprendere meglio quali sono le domande che ci dobbiamo porre di fronte a questa sfida e come approcciarle al meglio.

Da Docker a Kubernetes

Il primo esempio che vediamo riguarda il mondo di Docker: questo perché non sempre si ha a disposizione un file `docker-compose.yml` o si conosce questo mondo, ma magari si ha comunque la necessità o la curiosità di avvicinarsi a Kubernetes con il minor numero di intoppi possibile.

In questo caso, partiamo da un esempio volutamente semplice, che però ci servirà a comprendere quali sono gli strumenti di cui abbiamo bisogno per poter migrare a Kubernetes: useremo un'applicazione scritta in Flask, un framework piuttosto noto di Python, che esporrà una semplice pagina Web come la seguente:



Hello World!

Figura 13.1 Pagina principale dell'applicazione.

Il Dockerfile è il seguente: l'immagine utilizzata è quella di Python 3.8, che utilizza la cartella /app come cartella di lavoro principale, all'interno della quale verrà copiato il file requirements.txt che contiene le librerie che Python dovrà installare (in questo caso, solo Flask) e poi le installerà tramite il comando pip install. Infine, verrà creato un *entrypoint* tramite il comando python e le specifiche relative all'esecuzione dell'applicazione Flask:

Listato 13.1 Dockerfile dell'applicazione Python

```
FROM python:3.8-slim-buster

LABEL maintainer="serena.sensini@gmail.com"

WORKDIR /app

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

COPY . .

ENTRYPOINT ["python"]

CMD ["-m", "flask", "run", "--host=0.0.0.0"]
```

Per testarla, eseguiamo il comando seguente e, aprendo il browser, otterremo lo stesso risultato mostrato nella Figura 13.1.

Listato 13.2 Build ed esecuzione dell'immagine Docker

```
docker build -t flask-app .
docker run -p 5000:5000 flask-app
>>>
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
  WARNING: This is a development server. Do not use it in a production deployment. Use a
  production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.17.0.2:5000
```

Porta 5000

Le applicazioni vengono esposte di default sulla porta 5000: questo è il motivo per cui all'interno del Dockerfile non è stato specificato, mentre nel comando docker run abbiamo dovuto definirlo tramite l'opzione -p.

Ora che l'applicazione è funzionante, abbiamo bisogno di "tradurla" in oggetti che siano utili su Kubernetes. Partiamo quindi dalla componente applicativa: tra i diversi controller che abbiamo a disposizione, quale possiamo sfruttare, considerato che l'applicazione è stateless e che non richiederà più di un Pod? Questo è il caso perfetto per utilizzare un Deployment, dove possiamo specificare eventuali porte, variabili di ambiente, l'immagine da utilizzare... A proposito di immagine: le risorse in Kubernetes utilizzano il DockerHub come repository per eseguire il pull delle immagini, motivo per cui dovremo in qualche modo rendere pubblica questa immagine.

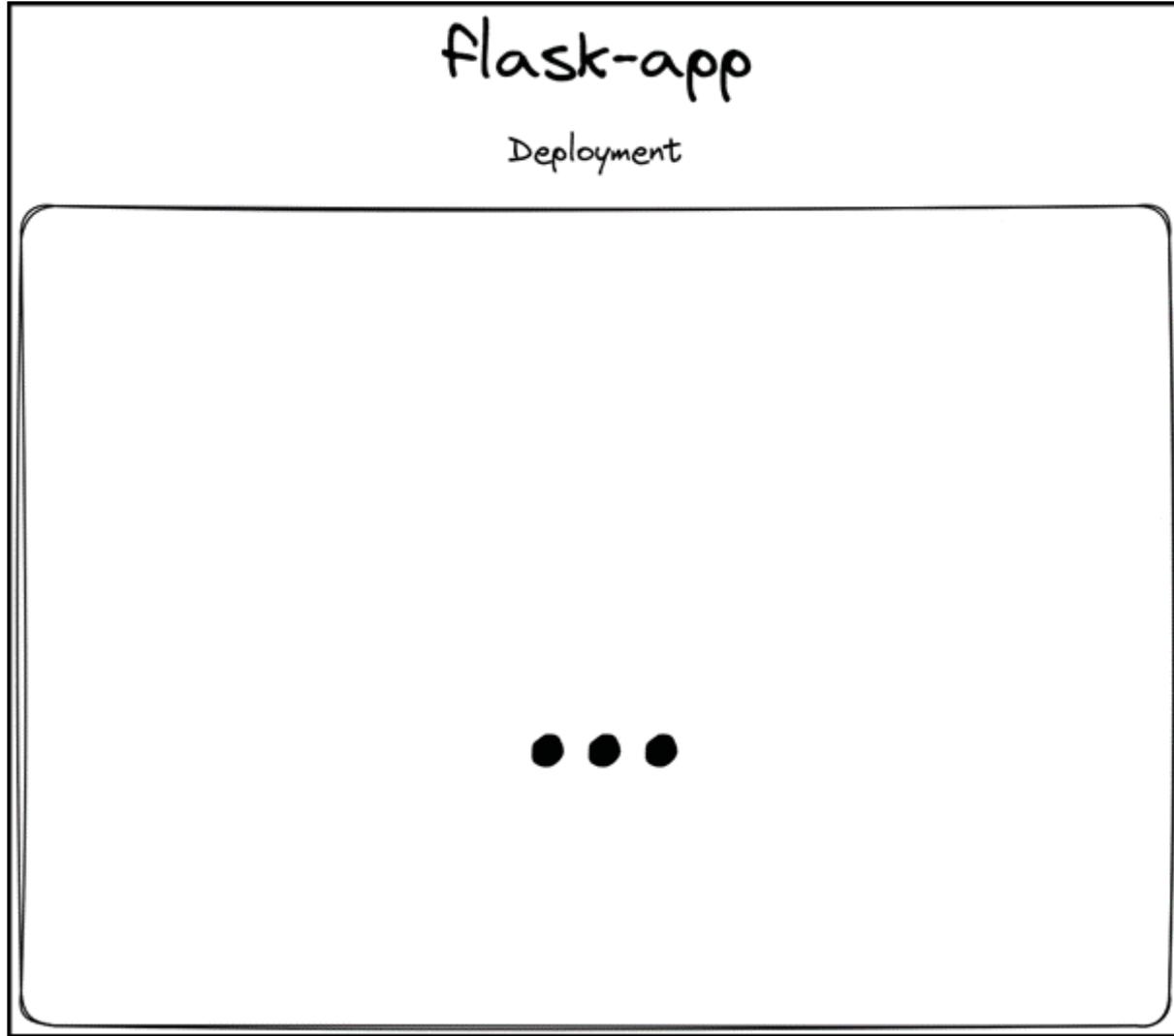


Figura 13.2 Schema iniziale dell'applicazione Flask.

La via più semplice è quella di eseguire il *push* dell'immagine sullo stesso DockerHub tramite il proprio account, nel modo che segue.

Listato 13.3 Login e push dell'immagine su DockerHub

```

docker login
>>>
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a
Docker ID, head over to https://hub.docker.com to create one.
Username: ssensini
Password:
Login Succeeded

docker push ssensini/simple-flask-app
>>>
Using default tag: latest
The push refers to repository [docker.io/ssensini/simple-flask-app]
7d56086c08b8: Pushed
375a79a61abf: Pushed
1d43d9675117: Pushed
452f1f17f45b: Pushed
ade4cdb42598: Mounted from library/python
a3af7ad05be9: Mounted from library/python

```

```
b42c4e0a74fd: Mounted from library/python
9a771a2f7675: Mounted from library/python
7b6f75f8765b: Mounted from library/python
latest: digest: sha256:87aecd4aecd01329e57a755286afe2c08b179bf3398be1f059290d32663d183
size: 2202
```

Ora che l'immagine è pubblica, possiamo provare a eseguire un container partendo da questa immagine, per vedere se funziona correttamente.

Listato 13.4 Login e push dell'immagine su DockerHub

```
docker run -d -it -p 5000:5000 --name flask-app-from-dh ssensini/simple-flask-app
>>>
80d9150fc42b8b035e5e7e5a4efcea1541b1df95bf5da15ede180a9438372b82

docker ps -a
>>>
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
PORTS             NAMES
80d9150fc42b      ssensini/simple-flask-app   "python -m flask run..."   15 seconds ago   Up 14
seconds          0.0.0.0:5000->5000/tcp    flask-app-from-dh
```

Funziona, ottimo! Iniziamo quindi a scrivere il Deployment che ci servirà per utilizzare l'immagine appena “pushata” sul registry pubblico: cominciamo con la definizione della risorsa, creando anche label che ci permettano di associare tutti gli eventuali oggetti che creeremo per questa applicazione. All'interno del template definiamo la struttura del container: aggiungiamo una variabile di ambiente che indichi la versione attuale dell'applicazione, più il riferimento all'immagine di cui dovrà eseguire il pull Kubernetes per poterla avviare. La porta che esporremo sarà poi la 5000, come visto in precedenza durante l'avvio del container Docker:

Listato 13.5 Creazione del Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: flask-app
  name: flask-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: flask-app
  template:
    metadata:
      labels:
        app: flask-app
    spec:
      containers:
        - env:
            - name: APP_VERSION
              value: "1.0.0"
            image: ssensini/simple-flask-app
            name: flask-app
            ports:
              - containerPort: 5000
      restartPolicy: Always
```

Creiamo quindi la risorsa Deployment tramite il relativo comando `kubectl` e proviamo a raggiungerla tramite il browser al consueto indirizzo `localhost:5000`.

Listato 13.6 Avvio del Deployment

```
kubectl apply -f ./deployment.yaml  
deployment.apps/flask-app created
```

Connessione non riuscita

Firefox non può stabilire una connessione con il server localhost:5000.

- Il sito potrebbe essere non disponibile o sovraccarico. Riprovare fra qualche istante.
- Se non è possibile caricare alcuna pagina, controllare la connessione di rete del computer.
- Se il computer o la rete sono protetti da un firewall o un proxy, assicurarsi che Firefox abbia i permessi per accedere al Web.

[Riprova](#)

Figura 13.3 Problema con la connessione all'applicazione.

L'applicazione non è visibile, e il motivo è semplice: il Deployment è raggiungibile solo all'interno del cluster Kubernetes, fintanto che non creiamo un oggetto che ci permetta di "collegarci": in questo caso, un Service è quello di cui abbiamo bisogno.

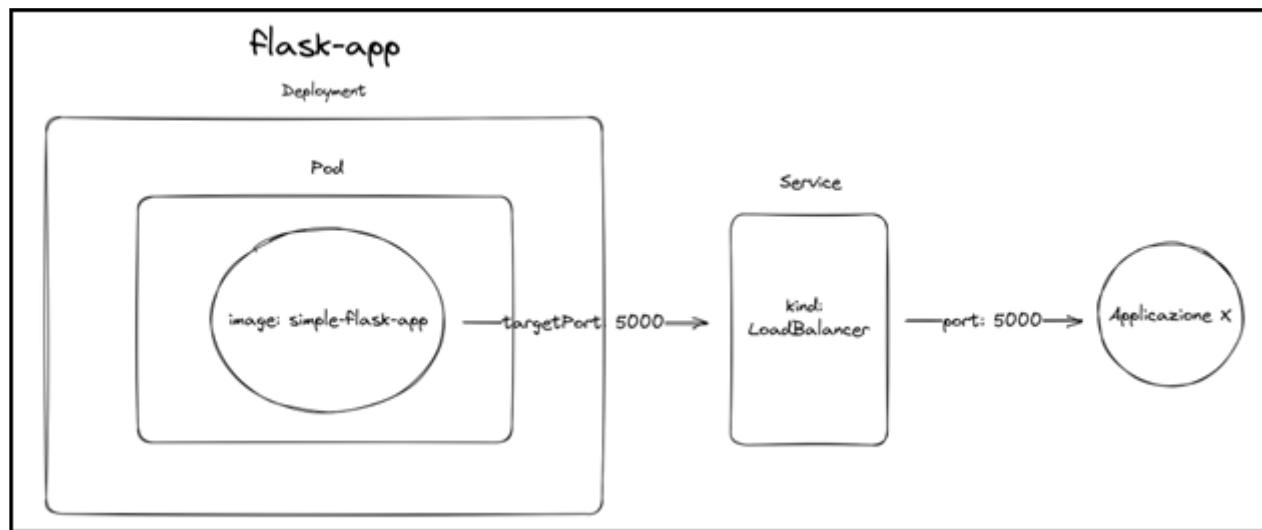


Figura 13.4 Schema aggiornato dell'applicazione.

Definiamo quindi un Service che esponga la porta 5000 del container sulla porta 5000, e utilizziamo un semplice LoadBalancer per farlo.

Listato 13.7 Definizione del Service

```
apiVersion: v1  
kind: Service  
metadata:  
  creationTimestamp: null  
  labels:  
    app: flask-app
```

```

name: flask-app
spec:
  ports:
    - name: "5000"
      port: 5000
      targetPort: 5000
  selector:
    app: flask-app
  type: LoadBalancer
  externalTrafficPolicy: Cluster

```

Completato questo step, è creato il Service tramite il consueto `kubectl apply`, l'applicazione è raggiungibile anche da browser. Questo esempio, apparentemente banale, ci ha portato a vedere con attenzione una serie di passaggi che ci permettono di “passare” da Docker a Kubernetes con alcuni accorgimenti, come il push dell’immagine su DockerHub. Nelle prossime sezioni, con tecnologie diverse (così da “movimentare” un po’ la parte pratica), vedremo come rendere un’applicazione Docker in Kubernetes sfruttando altri oggetti, come ConfigMap e Secret, piuttosto che gestirne la persistenza o anche utilizzando un’immagine di un repository privato.

Da Docker Compose a Kubernetes

In questa sezione, esamineremo come trasformare il file `docker-compose.yml` relativo all’immagine di Wikipedia in un file YAML pronto per l’utilizzo su Kubernetes. Il file relativo a Docker Compose è disponibile nella pagina ufficiale della documentazione di Wiki.js a questo indirizzo:

<https://docs.requarks.io/install/docker> (e ne troverai una copia all’interno del repository GitHub del manuale). Diamo un’occhiata più da vicino a questo file, per comprendere quali sono le entità in gioco:

Listato 13.8 File `docker-compose` di Wiki.js

```

version: <<3>>
services:
  db:
    image: postgres:11-alpine
    environment:
      POSTGRES_DB: wiki
      POSTGRES_PASSWORD: wikijsrocks
      POSTGRES_USER: wikijs
    logging:
      driver: "none"
    restart: unless-stopped
    volumes:
      - db-data:/var/lib/postgresql/data

  wiki:
    image: ghcr.io/requarks/wiki:2
    depends_on:
      - db
    environment:
      DB_TYPE: postgres
      DB_HOST: db
      DB_PORT: 5432
      DB_USER: wikijs
      DB_PASS: wikijsrocks
      DB_NAME: wiki
    restart: unless-stopped
    ports:
      - "80:3000"

volumes:
  db-data:

```

In questo esempio, abbiamo a che fare con due servizi: uno è relativo all'immagine PostgreSQL versione 11, database utilizzato per la persistenza della nostra applicazione, e l'altra riguarda Wiki.js, con relativa configurazione per comunicare con il database. In entrambi i servizi vediamo infatti delle variabili di ambiente che servono a descrivere la configurazione di base del database in un caso, mentre nell'immagine chiamata `wiki` le variabili di ambiente descrivono le informazioni che permetteranno all'applicazione di persistere i dati della documentazione che andremo a inserire su Wikipedia. Queste informazioni saranno infatti salvate all'interno di un volume chiamato `db-data`, che permetterà di recuperare in qualsiasi momento le pagine inserite e salvate attraverso Wiki.js.

Possiamo quindi riassumere l'architettura complessiva con lo schema in Figura 13.5.

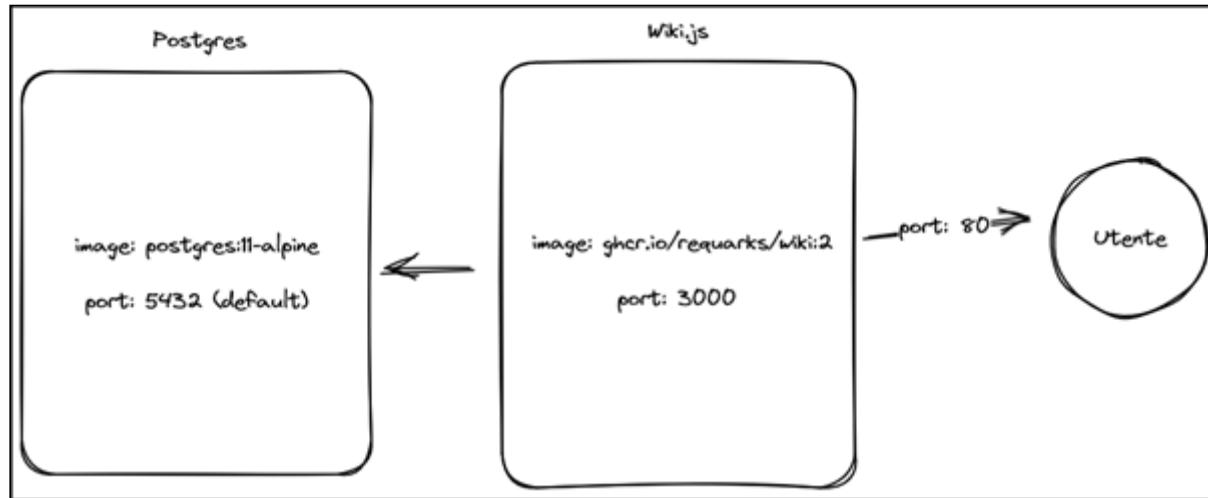


Figura 13.5 Come i diversi componenti comunicano.

Una volta che siamo a questo punto, abbiamo due alternative per quanto riguarda la migrazione dei nostri servizi da Docker Compose a Kubernetes: la prima è quella di “tradurre” l'architettura finora descritta sfruttando gli oggetti messi a disposizione dal nostro orchestratore, cercando di individuare responsabilità e proprietà adatte. Per esempio, come riporto il container di Wiki.js in Kubernetes? Posso sfruttare un Deployment, che istanzierà un Pod per contenere l'istanza dell'immagine applicativa, così come posso replicare lo stesso meccanismo per PostgreSQL. Per mettere in comunicazione i due container, è possibile sfruttare un Service, che esponga le porte necessarie alla comunicazione tra due, e così via. Si tratta di un processo apparentemente noioso, ma che ci dà una grande visione di insieme del funzionamento complessivo della soluzione.

Un'altra alternativa è quella di sfruttare un tool che, in maniera automatica, converte il file `docker-compose.yml` in una serie di oggetti che rappresentano quanto raccontato prima. Questa è sicuramente una strada più facile, soprattutto se non abbiamo molto tempo a disposizione, ma attenzione: strumenti come questi eseguono una traduzione meccanica, non tenendo conto che alcune di queste entità, una volta tradotte, hanno bisogno di una configurazione più complessa di quella che è possibile descrivere attraverso Docker Compose.

Procediamo, però, per gradi, e usiamo una strategia mista per apprezzare entrambi gli approcci: iniziamo quindi a scoprire cos'è Kompose, e come funziona. Kompose supporta la conversione dei file di Docker Compose in oggetti Kubernetes e OpenShift. Non solo: ti consente di selezionare più file Docker Compose contemporaneamente, e per impostazione predefinita in Kompose verranno generati dei file YAML per descrivere gli oggetti tradotti, anche se avrai un'opzione alternativa per generare dei file JSON, piuttosto che un chart per Helm.

La procedura per installare Kompose è piuttosto semplice, anche se dipende dal sistema operativo. Le istruzioni sono disponibili sul sito ufficiale (<https://kompose.io/>) ed è sufficiente eseguire un paio di comandi per averlo a disposizione all'interno del proprio laptop. Di seguito vengono riportate le istruzioni per installare l'ultima versione disponibile (attualmente la 1.27) sulla WSL di Windows, su MacOS o su Linux:

Listato 13.9 Installazione di Kompose per WSL, Linux o MacOS

```
# Linux  
curl -L https://github.com/kubernetes/kompose/releases/download/v1.27.0/kompose-linux-amd64  
-o kompose  
  
# macOS  
curl -L https://github.com/kubernetes/kompose/releases/download/v1.27.0/kompose-darwin-  
amd64 -o kompose  
  
chmod +x kompose  
sudo mv ./kompose /usr/local/bin/kompose
```

La nota simpatica è che l'installazione è così semplice che sul sito stesso le persone che hanno contribuito al suo progetto lo descrivono a prova di gatto.

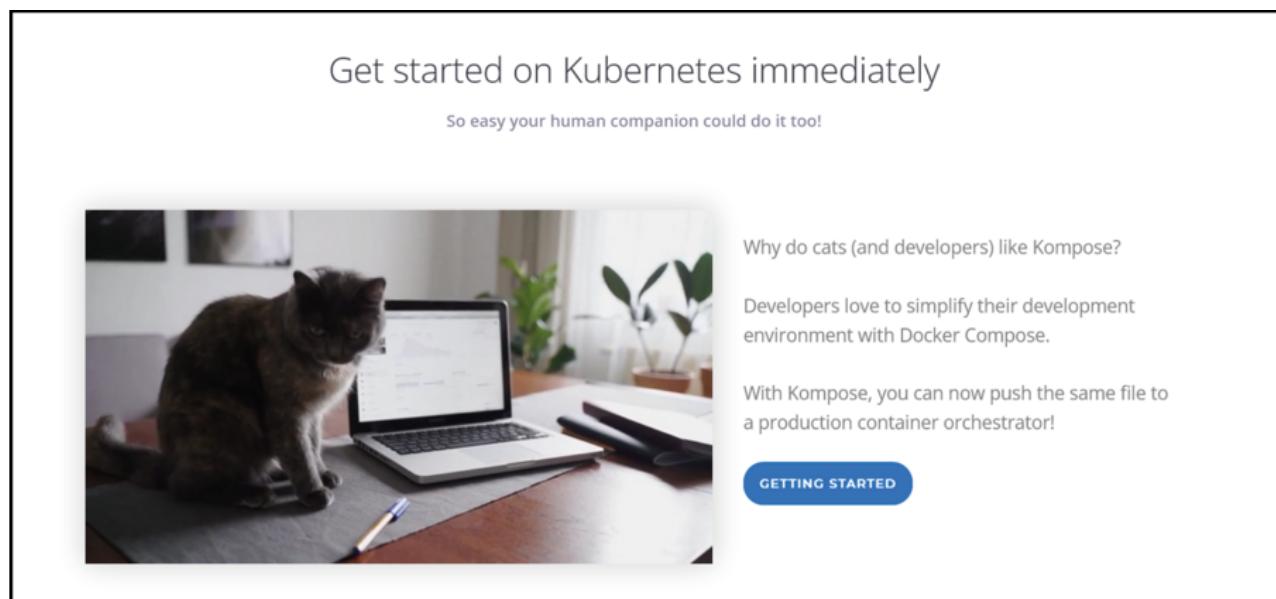


Figura 13.6 Estratto di una pagina della documentazione di Kompose.

Tornando persone serie, per poter convertire un file `docker-compose.yml` in qualcosa che anche Kubernetes possa eseguire attraverso i suoi oggetti, abbiamo bisogno di eseguire il comando che segue.

Listato 13.10 Conversione del file docker-compose.yml in oggetti K8S

```
kompose convert  
>>>  
WARN Restart policy 'unless-stopped' in service db is not supported, convert it to 'always'  
WARN Restart policy 'unless-stopped' in service wiki is not supported, convert it to  
'always'  
WARN Service "db" won't be created because 'ports' is not specified  
INFO Kubernetes file "wiki-service.yaml" created  
INFO Kubernetes file "db-deployment.yaml" created  
INFO Kubernetes file "db-data-persistentvolumeclaim.yaml" created  
INFO Kubernetes file "wiki-deployment.yaml" created
```

Se diamo un'occhiata a quanto prodotto da Kompose, notiamo che sono stati creati 4 oggetti: un file per il Service di Wiki.js, un Deployment per ogni applicazione e una PVC per memorizzare i dati del database. Il lavoro eseguito non è male, tuttavia richiede una serie di modifiche da effettuare. Diamo un'occhiata alla conversione, partendo dalla creazione del Deployment e del PVC di PostgreSQL.

Listato 13.11 Creazione del Deployment per Wiki.js

```
kubectl apply -f db-data-persistentvolumeclaim.yaml
kubectl apply -f db-deployment.yaml
>>>
2023-01-05 16:33:50
2023-01-05 16:33:50 PostgreSQL Database directory appears to contain a database; Skipping initialization
2023-01-05 16:33:50
2023-01-05 16:33:50 2023-01-05 15:33:50.065 UTC [1] LOG:  listening on IPv4 address
"0.0.0.0", port 5432
2023-01-05 16:33:50 2023-01-05 15:33:50.065 UTC [1] LOG:  listening on IPv6 address "::",
port 5432
2023-01-05 16:33:50 2023-01-05 15:33:50.075 UTC [1] LOG:  listening on Unix socket
"/var/run/postgresql/.s.PGSQL.5432"
2023-01-05 16:33:50 2023-01-05 15:33:50.107 UTC [22] LOG:  database system was shut down at
2023-01-05 15:33:43 UTC
2023-01-05 16:33:50 2023-01-05 15:33:50.123 UTC [1] LOG:  database system is ready to
accept connections
```

Perfetto, il database è in esecuzione ed è funzionante: possiamo anche entrare all'interno dell'istanza tramite l'utenza specificata nelle variabili di ambiente collegandoci con il terminale del container tramite i seguenti comandi.

Listato 13.12 Connessione al Pod di PostgreSQL

```
kubectl exec --stdin --tty db-7f695fdd7b-7smwp -- /bin/bash
db-7f695fdd7b-7smwp:/# psql -U wikijs -d wiki
psql (11.18)
Type "help" for help.
wiki=#
```

Occupiamoci ora della parte applicativa: Kompose, durante la fase di “traduzione”, non ha creato un Service relativo al database, non avendo trovato una porta esplicitata nel file. Mentre questo per Docker Compose non rappresenta un problema, per Kubernetes significa non potersi collegare a PostgreSQL correttamente: in effetti, se provassimo a creare i due Deployment senza che il Service per il database, risultato sarebbe il seguente.

Listato 13.13 Creazione del Deployment di Wiki.js

```
kubectl apply -f wiki-deployment.yaml
```

Listato 13.14 Tentativo di comunicazione tra PostgreSQL e Wiki.js

```
2023-01-05 16:33:59 Loading configuration from /wiki/config.yml... OK
2023-01-05 16:34:00 2023-01-05T15:34:00.105Z [MASTER] info:
=====
2023-01-05 16:34:00 2023-01-05T15:34:00.108Z [MASTER] info: = Wiki.js 2.5.295
=====
2023-01-05 16:34:00 2023-01-05T15:34:00.108Z [MASTER] info:
=====
2023-01-05 16:34:00 2023-01-05T15:34:00.108Z [MASTER] info: Initializing...
2023-01-05 16:34:01 2023-01-05T15:34:01.185Z [MASTER] info: Using database driver pg for
postgres [OK]
2023-01-05 16:34:01 2023-01-05T15:34:01.193Z [MASTER] info: Connecting to database...
2023-01-05 16:34:06 2023-01-05T15:34:06.220Z [MASTER] error: Database Connection Error:
EAI_AGAIN undefined:undefined
2023-01-05 16:34:06 2023-01-05T15:34:06.220Z [MASTER] warn: Will retry in 3 seconds...
```

```
[Attempt 1 of 10]
2023-01-05 16:34:09 2023-01-05T15:34:09.221Z [MASTER] info: Connecting to database...
...
```

Il Deployment va in errore e prova a collegarsi al database per un totale di dieci volte, salvo poi riavviare il Pod. Notare che nei log è presente come stringa di connessione `undefined:undefined`, ossia il corrispettivo dell'indirizzo IP (o dell'hostname) e della relativa porta, al momento sconosciute. Per far sì che i due possano comunicare, abbiamo bisogno di un Service: andiamo quindi a crearne uno sulla falsariga di quelli visti nei capitoli precedenti, di modo che PostgreSQL esponga la porta 5432 attraverso questo layer.

Listato 13.15 Creazione del Service per PostgreSQL

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    name: wiki
  name: db
spec:
  ports:
    - name: "5432"
      port: 5432
      targetPort: 5432
  selector:
    name: wiki
```

Una volta creato questo Service, il Deployment di Wiki.js sarà in grado di collegarsi correttamente:

Listato 13.16 Output del Deployment di Wiki.js

```
2023-01-05 16:34:17 2023-01-05T15:34:17.234Z [MASTER] error: Database Connection Error:
ECONNREFUSED 10.110.83.114:5432
...
2023-01-05 16:34:34 2023-01-05T15:34:34.257Z [MASTER] info: Connecting to database...
2023-01-05 16:34:34 2023-01-05T15:34:34.281Z [MASTER] info: Database Connection Successful
[ OK ]
2023-01-05 16:34:34 2023-01-05T15:34:34.348Z [MASTER] warn: DB Configuration is empty or
incomplete. Switching to Setup mode...
2023-01-05 16:34:34 2023-01-05T15:34:34.349Z [MASTER] info: Starting setup wizard...
2023-01-05 16:34:34 2023-01-05T15:34:34.630Z [MASTER] info: Starting HTTP server on port
3000...
2023-01-05 16:34:34 2023-01-05T15:34:34.630Z [MASTER] info: HTTP Server on port: [ 3000 ]
2023-01-05 16:34:34 2023-01-05T15:34:34.640Z [MASTER] info: HTTP Server: [ RUNNING ]
2023-01-05 16:34:34 2023-01-05T15:34:34.640Z [MASTER] info:
2023-01-05 16:34:34 2023-01-05T15:34:34.640Z [MASTER] info:
2023-01-05 16:34:34 2023-01-05T15:34:34.640Z [MASTER] info: Browse to http://YOUR-SERVER-
IP:3000/ to complete setup!
2023-01-05 16:34:34 2023-01-05T15:34:34.641Z [MASTER] info:
2023-01-05 16:34:34 2023-01-05T15:34:34.642Z [MASTER] info:
```

Adesso non soltanto nei log è visibile l'indirizzo del Service relativo al database con la porta, ma riesce anche a collegarsi correttamente. Proviamo dunque a creare il Service di Wiki.js generato da Kompose e accedere all'applicazione tramite il browser all'indirizzo `localhost:3000`. Il risultato sarà una pagina vuota, come se la nostra applicazione non fosse raggiungibile: allora apriamo il Service creato da Kompose relativo a Wiki.js, e notiamo una cosa.

Listato 13.17 Output del Service di Wiki.js

```
apiVersion: v1
kind: Service
metadata:
  annotations:
```

```

kompose.cmd: kompose convert
  kompose.version: 1.27.0 (b0ed6a2c9)
creationTimestamp: null
labels:
  io.kompose.service: wiki
name: wiki
spec:
  ports:
    - name: "80"
      port: 80
      targetPort: 3000
  selector:
    io.kompose.service: wiki
status:
  loadBalancer: {}

```

Kompose ha erroneamente mappato le porte del container e dell'host usando le informazioni disponibili nel file `docker-compose.yml`: mentre la porta a destra nella direttiva `.services.wiki.ports` rappresenta la porta del container, quella sinistra rappresenta la porta su cui vogliamo raggiungere l'applicazione in locale. Il Service è stato creato per essere in ascolto sulla porta 80 del container ed esporre la 3000; al contrario, abbiamo bisogno di collegare il Service alla porta 3000 del container ed esporre su una uguale o diversa!

Listato 13.18 Output del Deployment di Wiki.js

```

version: "3"
services:
  ...
  wiki:
    ...
    ports:
      - "80:3000"

```

Modifichiamo quindi il Service come di seguito, cancelliamo il vecchio e ricreiamo da zero per poi riprovarci a collegare tramite browser all'applicazione.

Listato 13.19 Aggiornamento del Service di Wiki.js

```

apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    name: wiki
  name: wiki
spec:
  ports:
    - name: "3000"
      port: 3000
      targetPort: 3000
  selector:
    name: wiki
  type: LoadBalancer
  externalTrafficPolicy: Cluster

```

Listato 13.20 Sostituzione del Service di Wiki.ks

```

kubectl delete svc wiki
kubectl apply -f wiki-service.yaml
>>>
service/wiki created

```

NOTA

Un Service può mappare qualsiasi porta in entrata attraverso il campo `targetPort`, anche se, per impostazione predefinita e per comodità, `targetPort` è sempre impostato sullo stesso valore del campo `port`.

Ora la nostra applicazione è in funzione e possiamo cominciare a fruirne... A questo esempio, abbastanza completo, manca un ultimo dettaglio da esaminare: nei log iniziali, durante la fase di conversione, ci era stato restituito un altro *warning* relativo alla gestione di un eventuale riavvio del Pod.

Listato 13.21 Conversione del file `docker-compose.yml` in oggetti K8S

```
kompose convert
>>>
WARN Restart policy 'unless-stopped' in service db is not supported, convert it to 'always'
WARN Restart policy 'unless-stopped' in service wiki is not supported, convert it to
'always'
```

Questa piccola nota può farci riflettere molto su come vogliamo gestire il ciclo di vita del Pod e, più in generale, come istruire il Deployment: in Kubernetes il campo `restartPolicy` viene impostato di default a `Always`, ma la specifica presente nel `docker-compose.yml` indica un qualcosa di diverso: se il Pod va in errore, allora deve essere riavviato, altrimenti si suppone che sia stato arrestato dall'utente. In Kubernetes, nel caso del Deployment, non ci sono alternative.

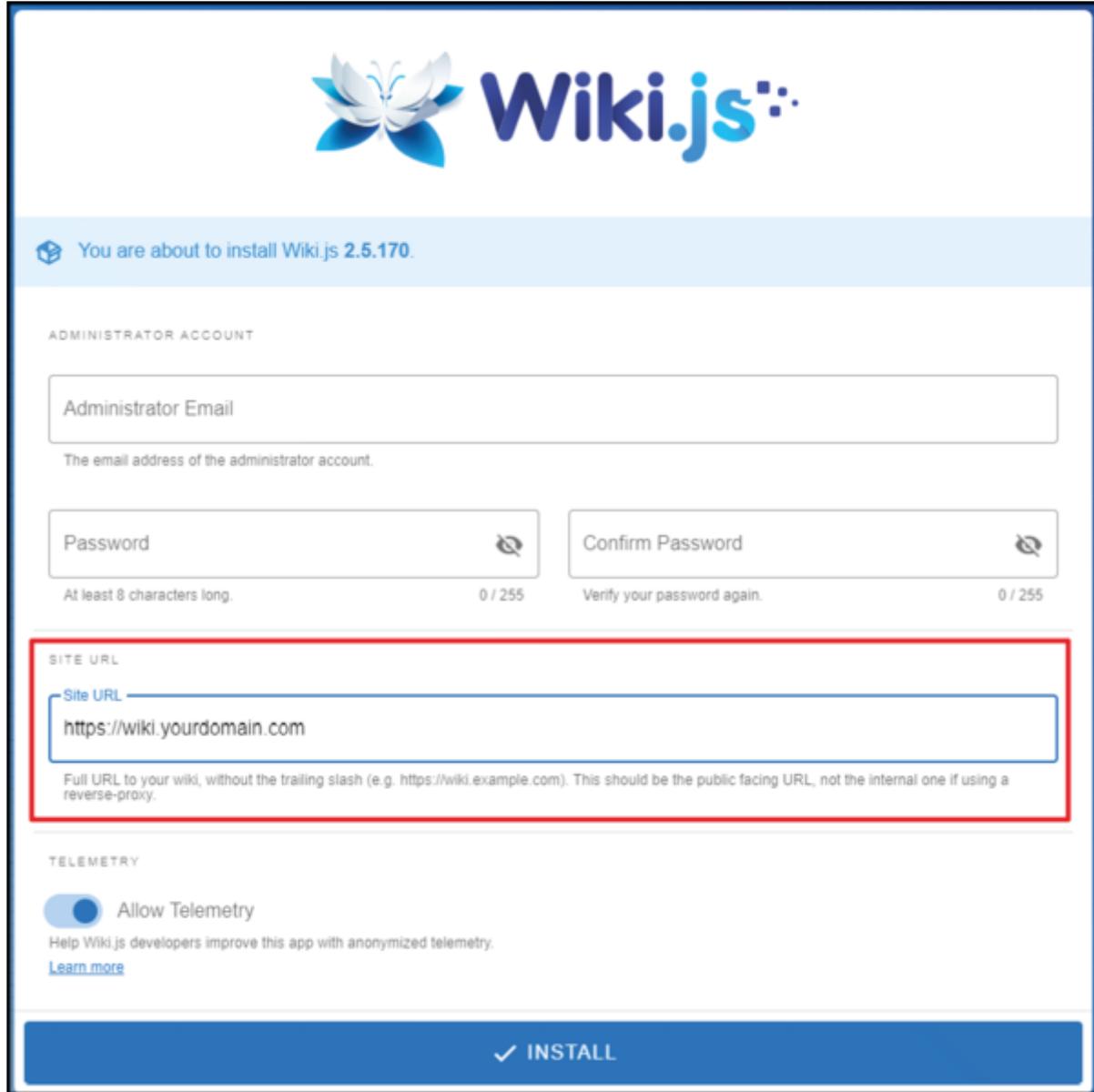


Figura 13.7 Pagina principale di Wiki.js.

Figura 13.8 Documentazione sul template di un Pod.

L'esempio appena sviluppato ci ha mostrato molte delle capacità di Kompose, che tuttavia richiedono un po' di familiarità con gli oggetti con cui stiamo lavorando: non è detto che la traduzione dal file `docker-compose.yaml` alle risorse Kubernetes sia perfetta e funzionante, ma richiede comunque un'attenta analisi di quanto prodotto per poterla eseguire senza intoppi. Ci sono infatti altre operazioni che potremmo voler compiere per sfruttare a pieno le potenzialità di Kubernetes, come utilizzare dei Secret per memorizzare le informazioni relative all'accesso del database da parte di Wiki.js e recuperarle da questo oggetto, piuttosto che esporle all'interno di variabili di ambiente, o anche una lettura più approfondita della definizione del PVC per comprendere se le dimensioni o le modalità di accesso sono quelle desiderate e sufficienti. Attenzione quindi ad affidarsi troppo a certi automatismi!

Avviare un cluster MongoDB

Qualche capitolo fa abbiamo visto come eseguire MongoDB tramite uno StatefulSet, a scopo di esempio: vediamo però come renderlo persistente, come configurare una replica come primaria e come configurare anche una liveness probe per testarne lo stato di integrità. Partiamo quindi da una semplice definizione del controller che ospiterà il cluster: assegnamo un nome a questa risorsa nel campo relativo ai metadati, una label `app` utile per raggruppare anche le altre risorse che andremo a creare e poi avremo il container, con l'immagine di MongoDB nella versione 6.0.4, all'interno del quale verrà avviato il demone `mongod`.

Listato 13.22 Definizione dello StatefulSet per MongoDB

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongo
spec:
  selector:
    matchLabels:
      app: mongo
  serviceName: "mongo"
  replicas: 3
  template:
    metadata:
```

```

labels:
  app: mongo
spec:
  containers:
    - name: mongodb
      image: mongo:6.0.4
      command:
        - mongod
        - --replSet
        - rs0
    - "--bind_ip"
      - "0.0.0.0"
    ports:
      - containerPort: 27017
        name: peer

```

Per comunicare con MongoDB, creiamo anche un Service **headless** che, come visto, ci permetta di raggiungere le diverse istanze:

Listato 13.23 Service headless per MongoDB

```

apiVersion: v1
kind: Service
metadata:
  name: mongo
spec:
  ports:
    - port: 27017
      name: peer
  clusterIP: None
  selector:
    app: mongo

```

Una volta che questo sarà messo in esecuzione tramite il comando `kubectl apply`, potremo definire la replica che fungerà da sito primario per il cluster: entriamo nel Pod `mongo-0` (o uno qualsiasi che vogliamo assuma questo ruolo) ed eseguiamo il seguente comando.

Listato 13.24 Configurazione del sito primario

```

kubectl exec -it mongo-0 mongosh # "mongosh" è per MongoDB con versione 6+, altrimenti
mongo è sufficiente
>>>
test> rs.initiate( {
  _id: "rs0",
  members:[ { _id: 0, host: "mongo-0.mongo:27017" } ]
});
>>>
{ ok: 1 }

```

Il comando `rs.initiate()` indica a MongoDB di avviare un insieme di repliche del cluster con il nome di `rs0` utilizzando il Service `mongo-0.mongo` come replica primaria. Eseguendo successivamente `rs.status()` per mostrare lo stato delle repliche del cluster, il risultato dovrebbe essere quello che segue.

Listato 13.25 Stato delle repliche di MongoDB

```

test> rs.status()
>>>
{
  set: 'rs0',
  date: ISODate("2023-02-17T18:02:25.972Z"),
  myState: 1,
  term: Long("1"),
  syncSourceHost: '',

```

```

syncSourceId: -1,
heartbeatIntervalMillis: Long("2000"),
majorityVoteCount: 1,
writeMajorityCount: 1,
votingMembersCount: 1,
writableVotingMembersCount: 1,
...
members: [
{
    id: 0,
    name: 'mongo-0.mongo:27017',
    health: 1,
    state: 1,
    stateStr: 'PRIMARY',
    uptime: 213,
    optime: { ts: Timestamp({ t: 1676656936, i: 17 }), t: Long(<<1>>) },
    optimeDate: ISODate(<<2023-02-17T18:02:16.000Z>>),
    lastAppliedWallTime: ISODate(<<2023-02-17T18:02:16.519Z>>),
    lastDurableWallTime: ISODate(<<2023-02-17T18:02:16.519Z>>),
    syncSourceHost: '',
    syncSourceId: -1,
    infoMessage: 'Could not find member to sync from',
    electionTime: Timestamp({ t: 1676656936, i: 2 }),
    electionDate: ISODate("2023-02-17T18:02:16.000Z"),
    configVersion: 1,
    configTerm: 1,
    self: true,
    lastHeartbeatMessage: ''
}
],
ok: 1,
...

```

Come visibile dall'output, al momento è presente solo un membro come "riconosciuto" all'interno del cluster; quindi indichiamo, sempre tramite la *shell* di Mongo, gli altri due Pod come repliche secondarie per il cluster tramite la funzione `rs.add()`:

Listato 13.26 Configurazione delle repliche secondarie

```

test> rs.add("mongo-1.mongo:27017");
>>>
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1676658467, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1676658467, i: 1 })
}
...
test> rs.add("mongo-2.mongo:27017");
>>>
{
  ok: 1,
...

```

A questo punto, avremo un cluster completo di tre repliche, di cui una primaria e due secondarie, il che ci garantisce una maggioranza in caso di perdita di uno dei nodi e quindi un'alta disponibilità del database.

Listato 13.27 Status del cluster aggiornato

```

test> rs.status();
>>>
{
  set: 'rs0',
  date: ISODate("2023-02-17T18:28:13.070Z"),
  myState: 1,
  term: Long("1"),
  syncSourceHost: '',
  syncSourceId: -1,
  heartbeatIntervalMillis: Long("2000"),
  majorityVoteCount: 2,
  writeMajorityCount: 2,
  votingMembersCount: 3,
  writableVotingMembersCount: 3,
...
members: [
  {
    _id: 0,
    name: 'mongo-0.mongo:27017',
    health: 1,
    state: 1,
    stateStr: 'PRIMARY',
    uptime: 91,
    optime: { ts: Timestamp({ t: 1676658491, i: 1 }), t: Long(<<1>>) },
    optimeDate: ISODate(<<2023-02-17T18:28:11.000Z>>),
    lastAppliedWallTime: ISODate(<<2023-02-17T18:28:11.707Z>>),
    lastDurableWallTime: ISODate(<<2023-02-17T18:28:11.707Z>>),
    syncSourceHost: '',
    syncSourceId: -1,
    infoMessage: 'Could not find member to sync from',
    electionTime: Timestamp({ t: 1676658456, i: 2 }),
    electionDate: ISODate("2023-02-17T18:27:36.000Z"),
    configVersion: 5,
    configTerm: 1,
    self: true,
    lastHeartbeatMessage: ''
  },
  {
    _id: 1,
    name: 'mongo-1.mongo:27017',
    health: 1,
    state: 2,
    stateStr: 'SECONDARY',
...

```

Come puoi vedere, abbiamo utilizzato gli hostnames specifici della replica per aggiungerli al nostro cluster Mongo e, a questo punto, abbiamo finito. Il nostro MongoDB replicato è attivo e funzionante. Questo esempio di per sé è abbastanza completo, ma noi vogliamo qualcosa di più: vogliamo vedere come poter automatizzare queste operazioni preliminari che abbiamo eseguito *dopo* che lo StatefulSet era stato creato: per farlo, aggiungiamo un container che esegua la configurazione iniziale già svolta prima che i Pod salgano, grazie a una ConfigMap che conterrà il nostro script di inizializzazione. Questo ci permetterà una certa libertà di manovra e soprattutto ci renderà più semplice il lavoro quando vorremo ripetere l'esperimento; modifichiamo quindi il file che definisce lo StatefulSet aggiungendo il riferimento a questa risorsa.

Listato 13.28 Definizione dello StatefulSet per MongoDB

```

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongo
spec:

```

```

selector:
  matchLabels:
    app: mongo
  serviceName: "mongo"
  replicas: 3
  template:
    metadata:
      labels:
        app: mongo
  spec:
    containers:
      - name: mongodb
        image: mongo:6.0.4
        command:
          - bash
          - /config/init.sh
        ports:
          - containerPort: 27017
            name: peer
        volumeMounts:
          - name: config
            mountPath: /config
    volumes:
      - name: config
        configMap:
          name: "mongo-init"

```

Il container utilizzerà una ConfigMap chiamata `mongo-init`, che contiene uno script: questo essenzialmente descriverà le stesse operazioni eseguite in precedenza, ma in maniera più ordinata: dopo aver controllato che il Service di MongoDB sia disponibile, proverà a collegarsi alla replica principale (che abbiamo scelto essere `mongo-0`) e la imposterà come replica primaria, per poi settare le altre come secondarie.

Listato 13.29 Definizione della ConfigMap mongo-init

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: mongo-init
data:
  init.sh: |
    #!/bin/bash
    until /usr/bin/mongosh --eval 'printjson(db.serverStatus())'; do
      echo "Sto controllando lo stato del database...."
      sleep 2
    done
    echo "OK."
  HOST=mongo-0.mongo:27017
  until /usr/bin/mongosh --host=${HOST} --eval 'printjson(db.serverStatus())'; do
    echo "Mi sto collegando all'istanza Mongo..."
    sleep 2
  done
  echo "Connesso."
  if [[ "${HOSTNAME}" != 'mongo-0' ]]; then
    until /usr/bin/mongosh --host=${HOST} --eval="printjson(rs.status())" | grep -v
"no replset config has been received"; do
      echo "In attesa di configurazione"
      sleep 2
    done
    echo "Sto aggiungendo le repliche."
    /usr/bin/mongosh --host=${HOST} --eval="printjson(rs.add('${HOSTNAME}.mongo'))"
  fi
  if [[ "${HOSTNAME}" == 'mongo-0' ]]; then
    echo "Configurazione del nodo primario"

```

```

/usr/bin/mongosh --eval="printjson(rs.initiate(\n  {'_id': 'rs0', 'members': [{}],\n   'host': 'mongo-0.mongo:27017'}))"\n  fi\n\n  echo "Configurazione completata."\nwhile true; do\n  sleep 3600\ndone

```

Kubernetes pattern: configurazione

Tra i pattern architetturali, ce ne sono alcuni che ricalcano le strategie utilizzate in Kubernetes per configurare la propria applicazione: l'esempio appena visto rende la configurazione facile da comprendere e gestire. La ConfigMap può essere letta come un intero file nel container dell'applicazione, ma anche facilmente modificata all'occorrenza. Inoltre, la stessa ConfigMap può essere aggiunta facilmente su container diversi. Con questo modello, la configurazione è "debolmente accoppiata" dall'applicazione, rispettando il pattern *loose coupling* molto spesso adottato nelle architetture enterprise.

L'ultimo step potrebbe essere quello di aggiungere allo StatefulSet delle probes: queste ci permettono di controllare se l'applicazione è attiva e funzionante, o se Kubernetes deve intervenire per riavivarla. In questo caso, ci torna utile la configurazione di una *liveness probe*, che può verificare lo stato del cluster Mongo per avere la certezza che tutto stia andando come ci si aspetta. Aggiungiamo quindi la *livenessProbe* nell'esempio precedente che eseguirà il comando `db.serverStatus()` tramite la *shell* di Mongo per testare la corretta esecuzione del database.

Listato 13.30 Aggiunta della liveness probe allo StatefulSet per MongoDB

```

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongo
spec:
  selector:
    matchLabels:
      app: mongo
  serviceName: "mongo"
  replicas: 3
  template:
    metadata:
      labels:
        app: mongo
  spec:
    containers:
      - name: mongodb
        image: mongo:6.0.4
        command:
          - bash
          - /config/init.sh
        ports:
          - containerPort: 27017
            name: peer
        volumeMounts:
          - name: config
            mountPath: /config
    livenessProbe:
      exec:
        command:
          - /usr/bin/mongosh
          - --eval
          - db.serverStatus()
        initialDelaySeconds: 10
        timeoutSeconds: 10

```

```

volumes:
  - name: config
    configMap:
      name: "mongo-init"

```

Dopo aver combinato StatefulSet, volumi persistenti e probes, disponiamo di un'installazione MongoDB pronta per Kubernetes scalabile e configurabile; questo esempio ci torna particolarmente utile perché, sebbene riguardasse MongoDB, i passaggi per la creazione di uno StatefulSet per gestire altre soluzioni che abbiano delle caratteristiche simili a quelle descritte diventa più semplice, ed è possibile seguire lo stesso filo logico adottato finora anche per altri sistemi di gestione di dati, o per comprendere la scelta dietro ad applicazioni che hanno adottato lo stesso tipo di risorsa, per esempio per raccogliere i log prodotti dai diversi container, come avviene con Fluentd.

Backup di un database Postgres

Buon proposito dell'anno: eseguire un backup dei database per non trovarci mai più nella situazione di dover spiegare perché ancora non ci avevamo pensato. Scherzi a parte, eseguire il backup dei tuoi dati è di fondamentale importanza per quando le cose andranno storte: per questo caso d'uso, andremo a vedere come programmare dei backup ogni 12 ore di un database PostgreSQL avviato all'interno del nostro cluster e renderli persistenti all'interno di un volume, così che si abbia una copia esterna e aggiuntiva dei dati a disposizione del database.

Disclaimer

Per rendere l'esempio più completo, viene riportato anche un esempio di avvio di un'istanza Postgres su Kubernetes, con le dovute semplificazioni sul tema: non è infatti oggetto dell'esempio vedere come creare un'istanza altamente affidabile, piuttosto che collegarci dall'esterno al database; l'esempio vuole mostrare le potenzialità di un oggetto che permetta la schedulazione di un'attività come il backup su un'istanza già presente. Se quindi hai già un database Postgres presente nel tuo cluster, puoi saltare questa prima parte e procedere con il backup seguendo le indicazioni fornite successivamente.

Partiamo proprio da quest'ultimo: creiamo un'istanza di PostgreSQL definito attraverso i seguenti file YAML che descrivono, rispettivamente, il Deployment con i riferimenti all'immagine di PostgreSQL nella versione 11, il Secret da cui recuperare la password del database e il volume in cui i dati saranno persistiti. Vediamo nel dettaglio ognuno di essi, e partiamo dal Secret: questo conterrà la password per collegarsi al database, che nell'esempio è volutamente definita attraverso il campo `stringData`, così da non dover codificare in formato Base64 la stringa:

Listato 13.31 Esempio del Secret di PostgreSQL

```

kind: Secret
apiVersion: v1
metadata:
  name: postgres-secret-config
stringData:
  password: password
type: Opaque

```

Il Deployment si chiamerà `postgres` e verrà creato a partire dall'immagine `postgres:11` disponibile su DockerHub; per poter comunicare con l'esterno, esporremo la porta 5432, ossia la porta di default di questo sistema di Postgres, e utilizzerà un volume persistente chiamato `postgres-pv-storage`, creato grazie a una PersistentVolumeClaim chiamata `postgres-pv-claim`. I dati verranno salvati all'interno della cartella `/var/lib/postgresql/data/pgdata`, anche questa predefinita, e quindi l'istruzione per il `mount` del volume è su questa directory specifica.

Listato 13.32 Esempio del Deployment di PostgreSQL

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      volumes:
        - name: postgres-pv-storage
          persistentVolumeClaim:
            claimName: postgres-pv-claim
      containers:
        - name: postgres
          image: postgres:11
          imagePullPolicy: IfNotPresent
      ports:
        - containerPort: 5432
      env:
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: postgres-secret-config
              key: password
        - name: PGDATA
          value: /var/lib/postgresql/data/pgdata
      volumeMounts:
        - mountPath: /var/lib/postgresql/data
          name: postgres-pv-storage
```

La PersistentVolumeClaim avrà invece questa definizione: il nome corrisponde a quanto specificato nel Deployment, così come le label: la modalità di accesso è di lettura e scrittura da un solo nodo, dal momento che non si tratta di un database replicato né che intendiamo scalare, e avrà uno spazio di 5Gi.

Listato 13.33 Esempio di PersistentVolumeClaim di PostgreSQL

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: postgres-pv-claim
  labels:
    app: postgres
spec:
  accessModes:
    - ReadWriteOnce
  resources:
  requests:
    storage: 5Gi
```

Ultimo step, per permettere allo strumento con cui eseguiremo il backup di comunicare con l'istanza del database, è la creazione di un Service: in questo caso, adottiamo un NodePort come tipologia, per permettere in futuro all'utente di collegarsi anche attraverso il proprio client locale (per esempio, *TablePlus* o *pgAdmin*) all'istanza di Postgres.

Listato 13.34 Esempio di Service di PostgreSQL

```
kind: Service
apiVersion: v1
metadata:
  name: postgres
  labels:
    app: postgres
spec:
  ports:
    - protocol: TCP
      port: 5432
      targetPort: 5432
  selector:
    app: postgres
  type: NodePort
```

Creati questi oggetti, potremo entrare nel terminale dell'istanza di Postgres e iniziare a lavorare con il database: eseguiamo il comando `kubectl exec` per accedere al container del Deployment e creare una tabella di test al suo interno, così da popolare il database.

Listato 13.35 Avvio del terminale

```
kubectl exec -it postgres-7dbff8c76-thrvt /bin/sh
>>>
```

Listato 13.36 Connessione al database tramite l'utente postgres

```
root@postgres-7dbff8c76-thrvt:/# psql -U postgres
>>>
psql (11.16 (Debian 11.16-1.pgdg90+1))
Type "help" for help.

postgres=#
```

Per creare una tabella, utilizziamo il database principale `postgres` e lo schema `public` con le seguenti query che definiscono la struttura dell'entità e inseriscono dei record di test.

Listato 13.37 Creazione della tabella tessera

```
CREATE TABLE tessera(
  codice int not null,
  totale_punti int not null,
  data_sottoscrizione date not null,
  ultimo_acquisto date not null,
  PRIMARY KEY (codice)
);
>>>
CREATE TABLE
```

Listato 13.38 Inserimento di alcuni record

```
postgres=# INSERT INTO
public.tessera(codice,totale_punti,data_sottoscrizione,ultimo_acquisto) VALUES
(53267,108,'2019-02-09','2019-08-09');
>>>
INSERT 0 1
postgres=# INSERT INTO
public.tessera(codice,totale_punti,data_sottoscrizione,ultimo_acquisto) VALUES
(48664,63,'2018-08-06','2020-03-08');
>>>
INSERT 0 1
>>>
```

Ora che il database è popolato, passiamo alla parte più interessante dell'esempio: la creazione di un backup è un'operazione che in Postgres può essere eseguita tramite `pg_dump` che possiamo pensare di eseguire ogni 12 ore, così da avere una copia sempre aggiornata dei dati. Per rendere quindi questi backup programmati, possiamo sfruttare il concetto di CronJob, e crearne uno che vada a eseguire il backup al posto nostro. Partiamo dall'immagine Docker: ne creiamo una *ad hoc* che, tramite Postgres stesso, esegua il dump che viene descritto all'interno della cartella `pg_backup`. Lo script che verrà eseguito andrà a creare un file con la data di esecuzione del backup nella cartella definita in precedenza e, sfruttando le variabili di ambiente specificate, si collegherà all'istanza Postgres per poi procedere al backup. Una volta terminato con successo, stamperà un messaggio per riportare l'esito positivo e uscirà con un codice di stato pari a 0 a conferma del successo dell'operazione.

Listato 13.39 Dockerfile dell'immagine per il backup di Postgres

```
FROM alpine:3.17

ENV PGHOST='localhost'
ENV PGPORT='5432'
ENV PGDATABASE='postgres'
ENV PGUSER='postgres@postgres'
ENV PGPASSWORD='password'

RUN apk update && \
    apk add postgresql && \
    apk add curl

COPY dumpDatabase.sh .

ENTRYPOINT [ "/bin/sh" ]
CMD [ "./dumpDatabase.sh" ]
```

Listato 13.40 Script per il dump

```
#!/bin/bash

DUMP_FILE_NAME="backupOn`date +%Y-%m-%d-%H-%M`.dump"
echo "Creating dump: $DUMP_FILE_NAME"

cd pg_backup

pg_dump -C -w --format=c --blobs > $DUMP_FILE_NAME

if [ $? -ne 0 ]; then
    rm $DUMP_FILE_NAME
    echo "Backup not created, check db connection settings"
    exit 1
fi

echo "Current backups:"
ls -la

echo 'Successfully Backed Up'
exit 0
```

Questa immagine dovrà essere *buildata* e poi pubblicata su un repository pubblico, come quello di DockerHub: se vuoi farlo attraverso il tuo account Docker, puoi eseguire i seguenti comandi, o altrimenti puoi utilizzare quella già disponibile e riportata nello step successivo.

Listato 13.41 Push sul repository Docker

```
docker login -u MYUSER
docker build -t pg-backup .
```

```

>>>
Sending build context to Docker daemon 8.192kB
Step 1/10 : FROM alpine:3.17
--> b2aa39c304c2
...
Removing intermediate container b1b0dc9a5ab8
--> fdce4861cff8
Step 10/10 : CMD [ "./dumpDatabase.sh" ]
--> Running in aa43e37c633e
Removing intermediate container aa43e37c633e
--> 51a6711c597d
Successfully built 51a6711c597d
Successfully tagged pg-backup:latest

docker tag pg-backup ssensini/pgsql-backup
>>>
...
docker push ssensini/pgsql-backup
>>>
Using default tag: latest
The push refers to repository [docker.io/ssensini/pgsql-backup]
fea05e287ac6: Pushed
a0a48ab1459b: Layer already exists
7cd52847ad77: Layer already exists

```

È giunto il momento di creare il CronJob: questo ci permetterà di definire *quando* eseguire l'istanza di backup di Postgres e su quale. Oltre a fare riferimento al container creato tramite l'immagine descritta precedentemente, e definire la pianificazione ogni dodici ore (schedule definito lo standard pari a `0 */12 * * *`), c'è la definizione delle variabili che permettono al Pod che verrà creato tramite il Job gestito da questa risorsa di collegarsi all'istanza di Postgres, ossia il nome del servizio da utilizzare (variabile `PGHOST`) piuttosto che la porta (variabile `PGPORT`) o l'utente (variabile `PGUSER`).

Listato 13.42 Definizione del CronJob per il backup dell'istanza Postgres

```

apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: batch-backup
spec:
  schedule: "0 */12 * * *"
  jobTemplate:
    spec:
      template:
        spec:
          restartPolicy: OnFailure
          containers:
            - name: postgresql-backup
              image: ssensini/pgsql-backup
              env:
                - name: PGHOST
                  value: "postgres"
                - name: PGPORT
                  value: "5432"
                - name: PGDATABASE
                  value: "postgres"
                - name: PGUSER
                  value: "postgres"
                - name: PGPASSWORD
                  value: "password"
              volumeMounts:
                - mountPath: "/pg_backup"
                  name: backup-volume
  volumes:

```

```

- name: backup-volume
  persistentVolumeClaim:
    claimName: pg-backup-pvc

```

La proprietà `volumeMounts` di un container ci consente di mappare un volume tramite la proprietà `mountPath` del container. I backup verranno eseguiti all'interno della cartella `pg_backup`, per cui avremo bisogno di una `PersistentVolumeClaim` chiamata `pg-backup-pvc`: un esempio di definizione è quella riportata di seguito.

Listato 13.43 Esempio di PVC per il backup

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pg-backup-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi

```

Per riassumere, abbiamo una PVC e abbiamo un CronJob che eseguirà il lavoro al posto nostro per effettuare il backup: una volta creati questi oggetti, non dovremo fare altro che aspettare che scatti la mezzanotte, e verificare che sia presente un Job con stato `Completed`, o anche che nei log del Job sia andato tutto a buon fine:

Listato 13.44 Esempio di output del Job

```

2023-02-20T17:14:50.975Z | Creating dump: backupOn2023-02-20-17-14.dump
2023-02-20T17:14:51.044Z | Current backups:
2023-02-20T17:14:51.045Z | total 28
2023-02-20T17:14:51.045Z | drwxr-xr-x  3 root  root          4096 Feb 20 17:14 .
2023-02-20T17:14:51.045Z | drwxr-xr-x  1 root  root          45 Feb 20 17:14 ..
2023-02-20T17:14:51.045Z | -rw-r--r--  1 root  root        1903 Feb 20 17:10
backupOn2023-02-20-17-10.dump
2023-02-20T17:14:51.045Z | drwx-----  2 root  root          16384 Feb 20 16:33
lost+found
2023-02-20T17:14:51.045Z | Successfully Backed Up

```

Se invece non volessimo attendere la prossima schedulazione e volessimo eseguirlo manualmente, potremmo usare il comando `kubectl create job` e creare un Job manualmente per eseguire un backup ulteriore: questo creerà una risorsa con il nome specificato (che, per semplicità, chiamiamo `batch-manual-001`) a partire dalla definizione del CronJob e avvierà il Pod che esegue lo script di `dump` di Postgres. Il risultato sarà lo stesso: il backup verrà eseguito e le informazioni circa la sua esecuzione saranno disponibili nei log del container.

Listato 13.45 Esecuzione manuale del Job

```

kubectl create job --from=cronjob/batch-backup batch-manual-001
>>>
job.batch/batch-manual-001 created

kubectl logs batch-manual-001-w26kd
>>>
2023-02-20T17:14:50.975Z | Creating dump: backupOn2023-02-20-17-14.dump
2023-02-20T17:14:51.044Z | Current backups:
2023-02-20T17:14:51.045Z | total 28
2023-02-20T17:14:51.045Z | drwxr-xr-x  3 root  root          4096 Feb 20 17:14 .
2023-02-20T17:14:51.045Z | drwxr-xr-x  1 root  root          45 Feb 20 17:14 ..
2023-02-20T17:14:51.045Z | -rw-r--r--  1 root  root        1903 Feb 20 17:10
backupOn2023-02-20-17-10.dump

```

```
2023-02-20T17:14:51.045Z | -rw-r--r--          1 root  root      1903 Feb 20 17:14
backupOn2023-02-20-17-14.dump
2023-02-20T17:14:51.045Z | drwx-----          2 root  root      16384 Feb 20 16:33
lost+found
2023-02-20T17:14:51.045Z | Successfully Backed Up
```

Stack MEAN

Aumentiamo il livello di complessità lavorando con uno stack MEAN: stavolta, quindi, ci avviciniamo un po' di più a quello che è un caso d'uso reale, dove abbiamo un'applicazione composta da diversi componenti, ognuno dei quali ha la propria responsabilità. Ci sarà infatti uno strato dedicato al front-end dell'applicazione, uno per il back-end e uno per la persistenza dei dati: MEAN infatti sta per MongoDB, Express.js, Angular e Node.js, e tramite questa combinazione di tecnologie è possibile infatti definire un ciclo di vita completo di un'applicazione web. Con MongoDB diamo alla nostra applicazione la persistenza dei dati di cui ha bisogno; con Node.js e Express.js creiamo dei servizi che ci mettono a disposizione un back-end per gestire la comunicazione con il database; infine, con Angular, è possibile creare un front-end che renda fruibile quello che c'è sotto al cofano. Parliamo quindi di un'applicazione full-stack, che segue il ciclo di vita dei suoi oggetti fino alla fine. Non c'è bisogno di spaventarsi di fronte alla moltitudine di queste tecnologie o alla difficoltà di avere più servizi che dovranno comunicare tra loro: grazie a Kubernetes, vedremo come creare ognuna delle risorse necessarie alla messa in esecuzione dello stack, come farle comunicare, e come accedervi.

Chiaramente questo tipo di realtà non è l'unico: la combo di queste tecnologie è di per sé conveniente per chi è particolarmente familiare con tecnologie affini a Typescript, ma in realtà si tratta di uno stack piuttosto diffuso come schema implementativo; questo vuol dire che è sufficiente sostituire ogni mattoncino (o servizio) con la tecnologia più congeniale allo sviluppatore per avere lo stesso risultato. Una cosa che infatti non si insegna mai (o si insegna troppo tardi) è che il linguaggio di programmazione con cui si decide di sviluppare una soluzione non dev'essere posto in cima alla lista delle cose da fare: questo deve essere scelto secondo diversi criteri, che dipendono anche dalla familiarità che si ha con uno di questi. In questo caso, si utilizzerà un approccio *top-down*: partendo infatti dalla definizione generica degli oggetti e delle risorse da utilizzare, andremo via via verso il particolare, ossia verso la costruzione delle singole entità e della relativa configurazione. Non a caso una buona base dati è alla base di molte applicazioni. Teniamo sempre presente che quanto vedremo riguarda prettamente Kubernetes e le sue risorse, ma non il codice Node.js o Angular: l'esempio completo sarà comunque disponibile all'interno del repository ufficiale del libro.

Un'applicazione che lavora con queste tecnologie può essere riassunta con questo schema: il database MongoDB si occuperà della parte di persistenza dei dati e andrà a comunicare direttamente con il back-end Express.js, che metterà a disposizione delle API per esporre dei dati che il front-end in Angular potrà mostrare nel browser (Figura 13.9). Già da questa descrizione, possiamo immaginarci alcune risorse Kubernetes che dovremo utilizzare e definire: ci saranno sicuramente dei Deployment per la parte relativa alle applicazioni, uno StatefulSet o un Deployment per MongoDB (discuteremo di questo più avanti), sicuramente un PersistentVolume per la parte di persistenza di MongoDB e dei Service per tutti gli attori, di modo che possano comunicare tra loro.

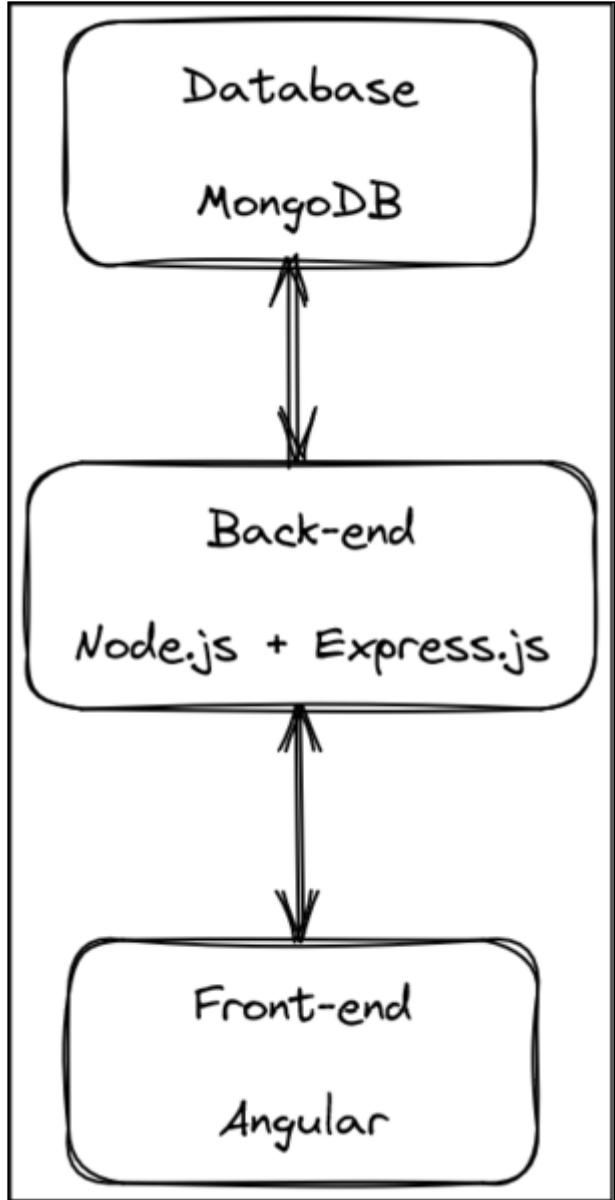


Figura 13.9 Architettura semplificata dei componenti di questo progetto.

Lo schema può quindi essere rivisto in questa chiave, riportando tutte le risorse Kubernetes finora citate (Figura 13.10); magari dovremo affinarlo man mano che ci lavoriamo su, ma possiamo usarlo come strategia generica per iniziare a lavorare sulla definizione delle singole risorse.

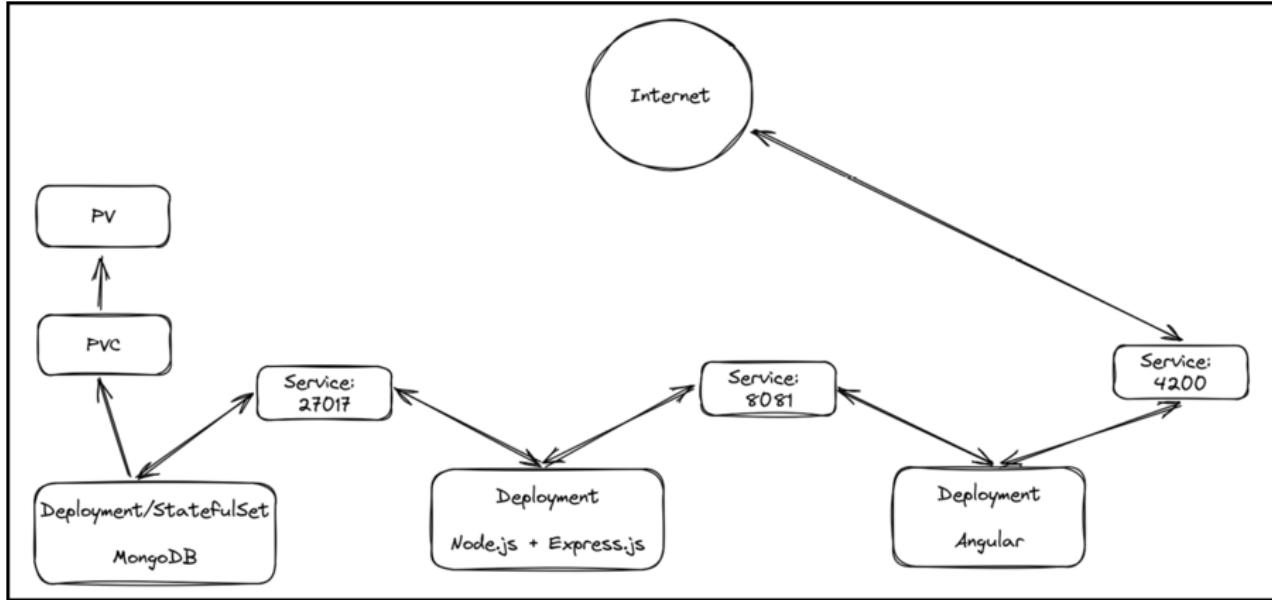


Figura 13.10 Architettura dal punto di vista di Kubernetes con le relative risorse.

Per risparmiarci un po' di lavoro, immaginiamo di avere a disposizione le immagini per la parte di front-end e back-end, mentre per MongoDB utilizzeremo una di quelle messe a disposizione da DockerHub. Come è immaginabile, le risorse da creare sono diverse, ma partiamo da quella che è alla base di tutto: MongoDB è il pezzo fondamentale su cui si regge tutto il resto. Avremo quindi bisogno di definire un controller che ci permetta di gestire questo database, e abbiamo detto che possiamo scegliere tra due diversi oggetti: da una parte, abbiamo il Deployment, che ci permette di creare un Pod per MongoDB con la relativa persistenza; in caso però di errore nel nodo o altre tipologie di problemi, questa replica potrebbe non essere sufficiente per garantire la disponibilità del database, su cui si regge poi tutta l'architettura. Uno StatefulSet, d'altro canto, ci permetterebbe anche di associare un volume e un'identità al database: questo non vuol dire che potremo creare più repliche del database come se lavorassimo con un'applicazione qualsiasi. Dobbiamo tenere conto che quando MongoDB dovrà scrivere dei dati, questi dovranno essere persistiti da qualche parte e ci dev'essere uno strumento che gestisce il traffico di informazioni in ingresso e anche le diverse repliche, che altrimenti rischiano di essere inconsistenti. In questo caso, infatti, dovremo prevedere un qualche meccanismo che stabilisca quale di queste repliche lavora come primaria, e quali come secondarie; questo tipo di configurazione è sicuramente più resiliente ai guasti, ma richiede una definizione più accurata. Vediamo quindi entrambe le configurazioni, tenendo in considerazione che la scelta di adozione di un Deployment per MongoDB è generalmente sconsigliata in quanto questo tipo di controller è adatto ad applicazioni stateless, al contrario della natura dei database; è comunque un'ipotesi che vale la pena di prendere in considerazione se vogliamo semplicemente mettere in piedi un ambiente di sviluppo, i cui dati possono essere eventualmente persi.

Come prima cosa, definiamo la PersistentVolumeClaim e il Deployment, supponendo che il cluster abbia un meccanismo che "stacca" automaticamente un pezzo di storage e crea il relativo volume quando la claim viene eseguita:

Listato 13.46 PVC per MongoDB

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  creationTimestamp: null

```

```

labels:
  app: dbdata
name: dbdata
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
status: {}

```

Listato 13.47 Deployment per MongoDB

```

apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: mongo
    name: mongo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mongo
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
        - args:
            - mongod
            - --port
            - "27017"
        env:
          - name: MONGO_INITDB_DATABASE
            value: mydb
          - name: MONGO_INITDB_ROOT_PASSWORD
            value: root
          - name: MONGO_INITDB_ROOT_USERNAME
            value: root
        image: mongo:6.0.4
      livenessProbe:
        exec:
          command:
            - echo 'db.runCommand("ping").ok' | mongosh mongo:27017/mydb --quiet
        failureThreshold: 5
        initialDelaySeconds: 40
      periodSeconds: 10
      timeoutSeconds: 20
    name: mongo
    ports:
      - containerPort: 27017
    resources: {}
    volumeMounts:
      - mountPath: /data/db
        name: dbdata
  restartPolicy: Always
  volumes:
    - name: dbdata
      persistentVolumeClaim:
        claimName: dbdata

```

Mentre per la definizione della PVC non è necessario aggiungere molte informazioni, visto che la definizione parla da sé, spendiamo due parole per il Deployment di Mongo. in questo caso abbiamo definito all'interno delle specifiche di questo oggetto che il numero di repliche deve essere pari a 1 e che Il Pod è creato avrà una label la cui chiave è pari ad `app` e il cui valore è `mongo`. DoPodiché, all'interno della sezione `containers` definiamo la natura del componente che stiamo per installare: intanto definiamo in che modo MongoDB deve essere avviato tramite il container, e lo facciamo sfruttando il comando `mongod`, che si mette in ascolto sulla porta 27017; a questo punto vengono definite anche delle variabili di ambiente, che servono a creare un primo database con le relative credenziali di accesso per permettere ai servizi di back-end di collegarsi. Come detto in precedenza, l'immagine utilizzata è quella ufficiale disponibile su DockerHub nella versione 6.0.4 e all'interno del file viene anche specificato in che modo deve essere montato il volume per la persistenza dei dati di Mongo. Ultimo ma non ultimo riguarda la configurazione della *liveness probe*, che ci permette di verificare che Il Pod sia correttamente in esecuzione utilizzando le funzioni che MongoDB mette a disposizione.

Se invece volessimo utilizzare uno StatefulSet, potremmo pensare di scriverlo in questo modo: attenendoci a quanto fatto per il Deployment, dobbiamo apportare alcuni cambiamenti, come il tipo di risorsa specificata, il numero di Pod che vogliamo vengano istanziati e il nome del Service che potrà essere utilizzato dalle altre applicazioni per contattare il database; inoltre, il comando che istanzia queste risorse deve tener conto che ci saranno più repliche, e quindi specifichiamo anche a `mongod` che è presente un insieme di repliche del database, che deve chiamare con `rs0` come prefisso. Per il resto, la definizione rimane molto simile a quanto visto nell'esempio precedente: quello che cambierà, sono infatti i Service che i due dovranno utilizzare.

Listato 13.48 StatefulSet per MongoDB

```
apiVersion: "apps/v1"
kind: "StatefulSet"
metadata:
  name: "mongo"
spec:
  serviceName: "mongo"
  replicas: 3
  selector:
    matchLabels:
      app: "mongo"
  template:
    metadata:
      labels:
        app: "mongo"
    spec:
      containers:
        - name: "mongo"
          image: "mongo"
          command:
            - mongod
            - "--replicaSet"
            - "rs0"
            - "--smallfiles"
            - "--noprealloc"
      imagePullPolicy: "Always"
      env:
        - name: "MONGO_INITDB_ROOT_USERNAME"
          value: "root"
        - name: "MONGO_INITDB_ROOT_PASSWORD"
          value: "root"
        - name: "MONGO_INITDB_DATABASE"
```

```

        value: "mydb"
  ports:
    - containerPort: 27017
  volumeMounts:
    - name: "dbdata"
      mountPath: "/data/db"
  volumes:
    - name: "dbdata"
  persistentVolumeClaim:
    claimName: "dbdata"

```

In effetti, nel caso del Deployment, possiamo creare un semplice Service di tipo ClusterIP che si metta in ascolto sulla porta di default di MongoDB e che dirotti il traffico in ingresso sulla stessa porta:

Listato 13.49 Service per il Deployment di MongoDB

```

apiVersion: v1
kind: Service
metadata:
  labels:
    app: mongo
  name: mongo
spec:
  ports:
    - name: "27017"
      port: 27017
      targetPort: 27017
  selector:
    app: mongo

```

Nel caso dello StatefulSet, il Service dovrà invece essere di tipo headless, e quindi non avrà un indirizzo IP all'interno del cluster. In ogni caso, per entrambe le possibilità, non prevediamo di esporre al di fuori del cluster l'istanza di MongoDB:

Listato 13.50 Service per il Deployment di MongoDB

```

apiVersion: v1
kind: Service
metadata:
  name: mongo
  labels:
    app: mongo
spec:
  ports:
    - port: 27017
      targetPort: 27017
  clusterIP: None
  selector:
    app: mongo

```

Ora che MongoDB è stato definito, possiamo passare agli altri due componenti, e iniziamo dal back-end: questo potrà essere definito attraverso un Deployment, trattandosi di un'applicazione stateless; nel caso in cui il Pod andasse in errore o il nodo che lo ospita subisse un guasto, questo potrà essere riavviato senza problemi, a parte il disservizio causato dal riavvio.

In questo caso, avremo quindi un Deployment che sfrutta un'immagine personalizzata che è disponibile sul DockerHub e che espone sulla porta 8081; per verificare che il Pod sia correttamente in esecuzione, viene inoltre configurata una *liveness probe*, che esegue un comando sull'indirizzo principale dell'API per farsi ritornare un codice 200. Questo avrà un ritardo iniziale di 30 secondi, per permettere all'applicazione di completare la relativa inizializzazione, e poi andrà a testare ogni 20 secondi le API; se queste rispondono entro 10 secondi con una risposta positiva, allora il test è superato; altrimenti, dopo aver eseguito tre tentativi senza successo, riavverà il Pod.

Listato 13.51 Deployment per il componente di back-end

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: backend
    name: backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: backend
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: backend
    spec:
      containers:
        - image: ssensini/mean-backend:1.0.0
          livenessProbe:
            exec:
              command:
                - curl
                - -f
                - http://localhost:8081/api/
            failureThreshold: 3
            periodSeconds: 20
            timeoutSeconds: 10
            initialDelaySeconds: 60
          name: backend
          ports:
            - containerPort: 8081
          resources: {}
      restartPolicy: Always
status: {}
```

Il Service esporrà, sempre tramite un ClusterIP, la porta 8081 del container sulla porta 8081 del servizio; anche in questo caso, in effetti, non è necessario esporre le API verso l'esterno del cluster, per cui questa tipologia di risorsa è più che sufficiente.

Listato 13.52 Service per il componente di back-end

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: backend
    name: backend
spec:
  ports:
    - name: "8081"
      port: 8081
      targetPort: 8081
  selector:
    app: backend
```

L'ultima risorsa riguarda il front-end: anche in questo caso, lavoreremo con un Deployment e un Service, che sono molto simili alla definizione di quanto visto finora per il back-end, per cui ne viene riportato solamente il parziale di entrambe le risorse:

Listato 13.53 Deployment per il componente di front-end

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: frontend
    name: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: frontend
  strategy: {}
  template:
    ...
    spec:
      containers:
        - image: ssensini/mean-frontend:1.0.0
          name: frontend
          ports:
            - containerPort: 4200
...
...
```

Listato 13.54 Service per il componente di front-end

```
apiVersion: v1
kind: Service
metadata:
...
  name: frontend
spec:
  ports:
    - name: "4200"
      port: 4200
      targetPort: 4200
...
...
```

La differenza rispetto a quanto visto finora riguarda principalmente la necessità di raggiungere dall'esterno del cluster questo componente, magari esponendolo tramite il protocollo HTTP: dovremo quindi valutare la creazione di un Ingress, che ci permetta di creare un accesso verso il front-end dell'applicazione. Modifichiamo quindi la rappresentazione precedente, aggiungendo un Ingress all'architettura complessiva; inoltre, il Service che ci permette di utilizzare al meglio questa risorsa è il LoadBalancer, quindi modifichiamo l'esempio fatto in precedenza e definiamo l'Ingress come segue.

Listato 13.55 Service di tipo LB per il componente di front-end

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  selector:
    app: frontend
  ports:
    - port: 4200
      targetPort: 4200
  type: LoadBalancer
...
```

Listato 13.56 Ingress per il componente di front-end

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  ...
  name: frontend-ingress
  ...
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/rewrite-target: /
```

```

name: frontend-ingress
spec:
  rules:
    - host: mynodeapp.com
      http:
        paths:
          - backend:
              serviceName: frontend
              servicePort: 4200

```

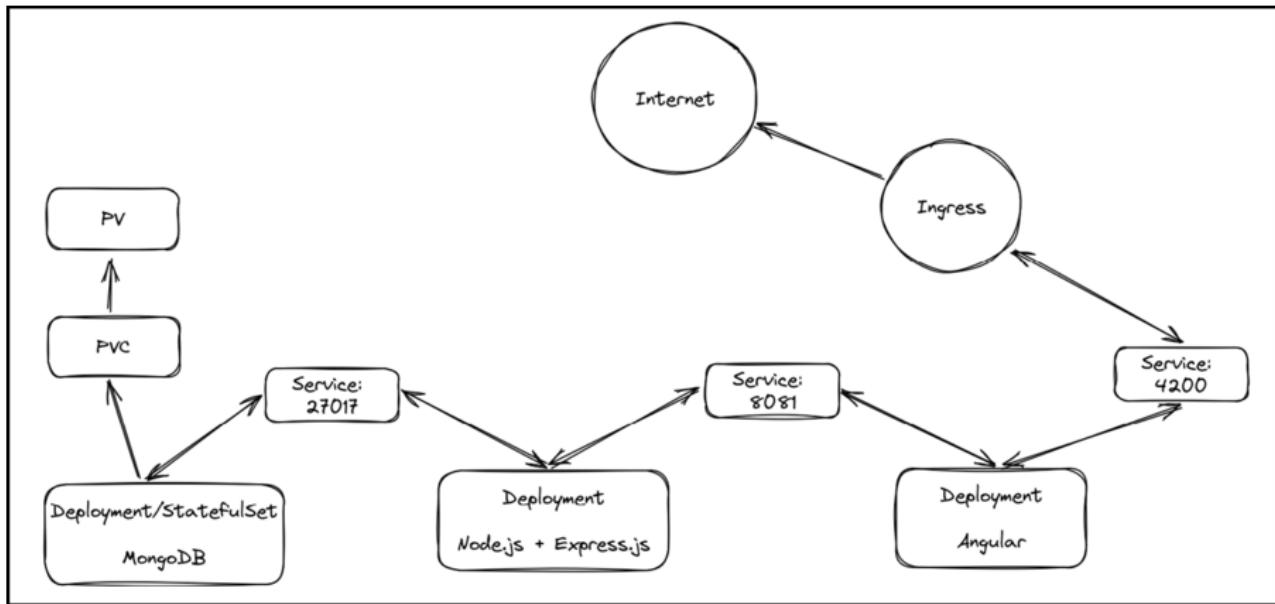


Figura 13.11 Architettura aggiornata.

Le risorse necessarie al funzionamento dell'infrastruttura sono pronte: se la creazione di tutti gli oggetti, nell'ordine con cui li abbiamo definiti, avrà successo, saremo in grado di aprire il browser sull'indirizzo specificato dall'Ingress e vedremo la pagina iniziale dell'applicazione!

Che cosa abbiamo imparato

- Tramite dei casi d'uso reali, abbiamo visto in che modo possiamo partire da una delle tecnologie a nostra disposizione per arrivare all'esecuzione di un'applicazione su Kubernetes.
- Utilizzando un semplice Dockerfile, e ragionando sull'astrazione delle diverse risorse, è possibile riportare il lavoro fatto su un cluster Kubernetes, così come è possibile ottenere lo stesso risultato partendo da un file Docker Compose.
- Abbiamo anche visto degli esempi concreti per ristrutturare un'applicazione Node.js affinché sia eseguibile su un cluster, dividendo i diversi oggetti che la compongono tra le diverse risorse messe a disposizione da questa tecnologia.
- Utilizzando MongoDB come esempio per la persistenza, abbiamo visto in che modo utilizzare risorse come gli StatefulSet.
- Per mettere in pratica le nozioni apprese sui Job, abbiamo fatto un esempio con Postgres e il suo backup.

Best practice per applicazioni enterprise

La nuova generazione di donne leader nell'industria tech deve affrontare numerose sfide che sono nate da un mondo storicamente dominato dagli uomini e che persistono ancora oggi, come il *gender bias* e la *pay inequality*. Tuttavia, ritengo che la sfida principale sia la mancanza di modelli da seguire. Da un lato, più donne leader che fanno sentire la loro voce dimostrano a chiunque abbia pregiudizi che questi non hanno alcun fondamento, poiché sono altrettanto capaci dei loro colleghi uomini. Dall'altro, avere più modelli ai quali ispirarsi può aiutare le nuove generazioni di donne a realizzare il loro potenziale e ad avere il coraggio di intraprendere percorsi di leadership.

– Pamela Gotti, CTO @ Credimi

Se siamo a questo punto del manuale, è perché il grosso del lavoro per comprendere come utilizzare il mondo di Kubernetes è fatto: abbiamo imparato quali sono le risorse principali e come gestirle e abbiamo visto anche diversi esempi pratici per sporcarci un po' le mani. Ci tocca però uscire ancora una volta dalla comfort zone, perché Kubernetes non è adatto ai deboli di cuore: quando lavoreremo in un ambiente che sarà poi utilizzato dall'utente finale, sarà necessario adottare una serie di tecniche che ci permettano di rendere il nostro cluster a prova di qualsiasi carico di lavoro.

In questo capitolo esploreremo, quindi, diversi aspetti che riguardano il modo in cui Kubernetes dovrebbe essere rivisto in un ambiente di *produzione*: queste sono solo alcune delle "pillole" che è possibile maturare utilizzando questo strumento, ma sicuramente l'esperienza la fa da padrona. Vediamo dunque alcuni accenni sulla scalabilità di un'applicazione, su come gestire un piano di disaster recovery o come fare economia con le risorse allocate ai singoli namespace piuttosto che ai Pod.

Scaling

La scalabilità è la capacità di un'applicazione di adattarsi all'aumento del carico di lavoro e indica se un sistema è in grado di crescere o meno. Questa proprietà riguarda non solo oggetti come i controller (per esempio, i Deployment), ma anche il cluster: è possibile infatti gestire la capacità della piattaforma sottostante per adattarsi a una crescita della domanda. È importante comprendere che non tutte le applicazioni sono in grado di scalare e non tutte le applicazioni che hanno questa capacità devono essere scalate.

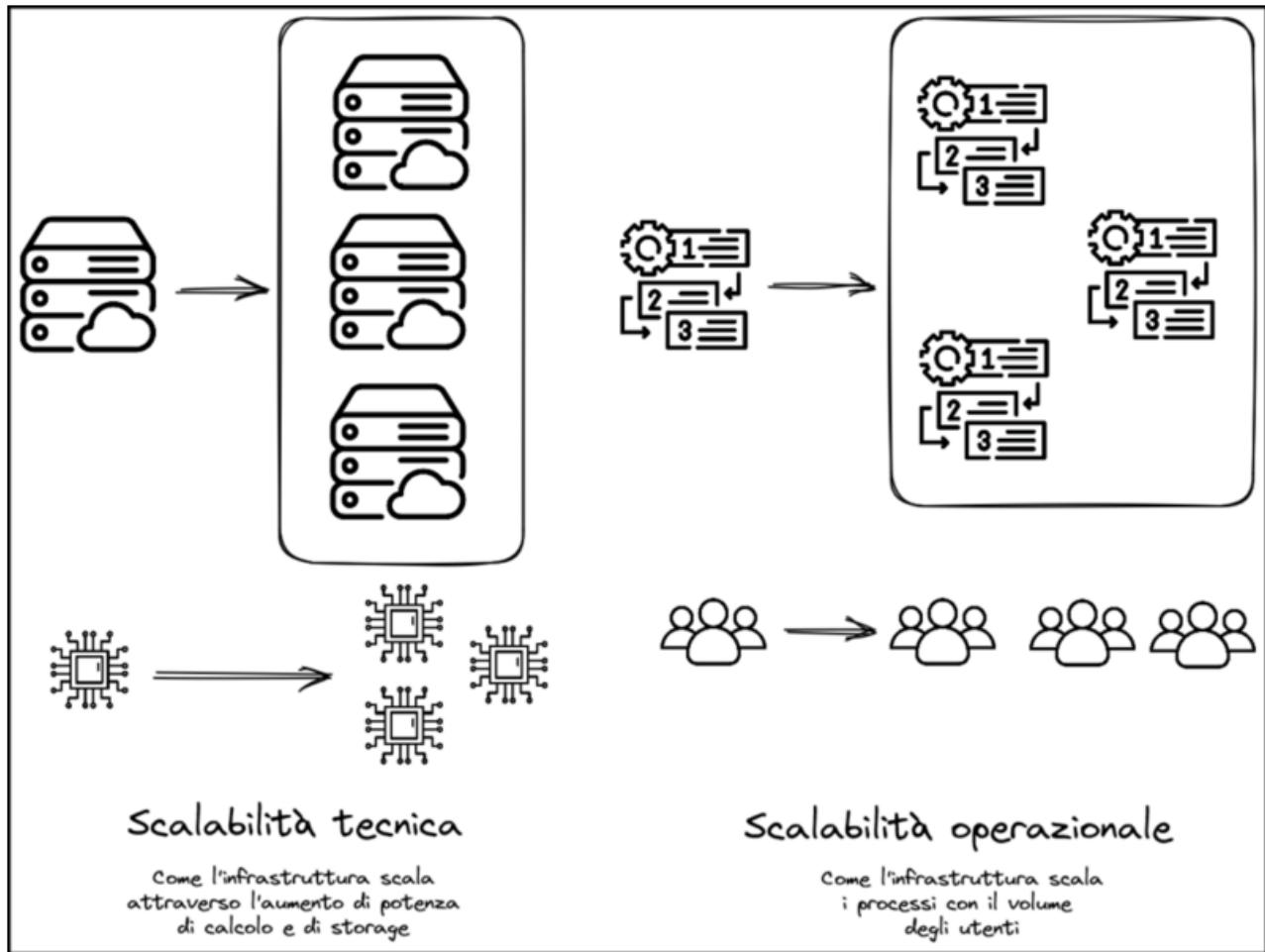


Figura 14.1 Come funzionano la scalabilità tecnica e quella operazionale, ad alto livello.

Fatte queste premesse, possiamo distinguere la scalabilità di una risorsa in due tipologie: quella *manuale*, dove il ridimensionamento avviene tramite dei comandi che permettono di aggiornare o ridurre il numero di Pod o di risorse disponibili, e quella *automatica*, dove si ha un processo di allocazione dinamica delle risorse in base ai requisiti di prestazioni definiti in precedenza nella gestione delle risorse. Al crescere del volume di lavoro, un'applicazione potrebbe richiedere risorse aggiuntive; quando la domanda diminuisce e le risorse aggiuntive non sono più necessarie, queste possono essere ridistribuite per ridurre i costi.

La scalabilità automatica sfrutta la flessibilità degli ambienti e delle applicazioni semplificando la gestione e riduce la necessità per un operatore di monitorare continuamente le prestazioni di un sistema e prendere decisioni sull'aggiunta o la rimozione di risorse.

È ovvio sottolineare che in un'azienda non sempre si hanno applicazioni che hanno bisogno di scalare, ma questo dipende dal tipo di applicazione e da come è stata progettata l'infrastruttura: se prendessimo, per esempio, un'applicazione che gestisce le prenotazioni di un ospedale con i diversi reparti e tutte le persone che vi lavorano, possiamo immaginare bene che un sistema di questo tipo è soggetto a dei picchi orari all'apertura degli uffici preposti alle prenotazioni. È in questo momento che si ha bisogno di avere a disposizione maggiori risorse e quindi di ricorrere a un meccanismo di scalabilità che ne aumenti il numero di repliche a disposizione; allo stesso modo, alla chiusura degli uffici, si può pensare di ridimensionare l'ambiente per evitare uno spreco di risorse e ridurre il numero di repliche attive.

Sebbene la scalabilità automatica possa offrire vantaggi straordinari, è importante riconoscere quando *non* dovresti utilizzare il ridimensionamento automatico. Questa modalità introduce complessità nella gestione del cluster e delle applicazioni installate al suo interno. Pertanto, se le richieste che arrivano a un'applicazione non cambiano notevolmente, può essere perfettamente accettabile gestire manualmente la crescita del volume di traffico. Se il caricamento dell'applicazione cambia invece in modo prevedibile, un intervento manuale per regolare la capacità in quei momenti può essere banale o non sufficientemente tempestivo, per cui una gestione automatizzata della scalabilità automatica potrebbe essere giustificato. Come avviene con la maggior parte di queste tecnologie, vanno sfruttate non per sovraccaricare il sistema, ma solo quando il vantaggio a lungo termine supera la relativa configurazione e manutenzione.

Quando scalare

Il tema della scalabilità automatica è particolarmente importante per i sistemi orientati ai servizi. Uno dei vantaggi della decomposizione delle applicazioni in entità distinte è la possibilità di ridimensionare parti diverse di un'applicazione in modo indipendente. Lo abbiamo fatto con architetture a più livelli ben prima che emergesse il *cloud*, ed è diventato un luogo comune separare le applicazioni Web dai loro database relazionali e ridimensionare l'app web in modo indipendente.

Con l'architettura dei microservizi questo tipo di separazione può essere estesa ulteriormente. Per esempio, un sito web aziendale può avere un servizio che gestisce il suo negozio online, che è diverso da un servizio che mostra i post del blog. Quando c'è un evento di marketing, il negozio online può necessitare di un ridimensionamento mentre il servizio relativo al blog non sarà interessato e potrà rimanere invariato.

Con questa opportunità di ridimensionare diversi servizi in modo indipendente, è possibile utilizzare in modo più efficiente l'infrastruttura utilizzata; tuttavia, l'automazione di questo processo di ridimensionamento diventa molto importante, se non essenziale.

La scalabilità automatica si presta bene a carichi di lavoro più piccoli e agili con immagini di dimensioni ridotte e tempi di avvio rapidi. Se il tempo necessario per eseguire il pull di un'immagine del container su un determinato nodo è breve, e se anche il tempo necessario all'avvio dell'applicazione dopo la creazione del container è breve, il carico di lavoro può rispondere rapidamente agli eventi che richiedono la scalabilità del sistema e la capacità può essere regolata molto più facilmente. Le applicazioni che hanno delle immagini di dimensioni superiori a un gigabyte e/o script di avvio che hanno tempo di esecuzione di minuti sono molto meno adatte a rispondere ai cambiamenti di carico e, dunque, carichi di lavoro come questo non sono buoni candidati per la scalabilità automatica, o comunque richiedono un'attenta configurazione che tenga conto di parametri aggiuntivi; tienilo a mente quando progetti e crei le tue app.

È anche importante riconoscere che il ridimensionamento automatico comporterà l'arresto delle istanze dell'app. Questo non si applica quando parliamo di scalabilità a livello applicativo, dove quindi varia il numero di Pod; tuttavia, nel caso in cui cambino le risorse del cluster, ciò che viene ridimensionato comporterà l'arresto delle istanze in esecuzione.

Quando non scalare

Prima di passare al lato più "tecnico" dell'argomento, vediamo anche perché dobbiamo valutare bene se la scalabilità fa al caso nostro. Abbiamo già detto che non tutti i carichi di lavoro possono scalare, soprattutto se orizzontalmente: per esempio, per le applicazioni che non possono condividere il carico tra istanze distinte, il ridimensionamento orizzontale è inutile. Questo è vero per alcune applicazioni

stateful per le applicazioni che richiedono un'elezione da parte dei leader. Per questi casi d'uso è possibile prendere in considerazione la scalabilità verticale del Pod.

Un altro fattore da considerare è la capacità del cluster: man mano che un'applicazione scala, potrebbe esaurire la capacità disponibile nei nodi di lavoro di un cluster. Questo può essere risolto fornendo una capacità sufficiente in anticipo, utilizzando degli avvisi per richiedere agli operatori della piattaforma di aggiungere ulteriore capacità, oppure utilizzando la scalabilità automatica del cluster, discussa in un'altra sezione di questo capitolo; in ogni caso, bisogna tener conto delle capacità complessive del cluster, di modo che la gestione automatica di queste risorse non ne comprometta la funzionalità.

Scalabilità manuale

Kubernetes offre diversi modi per scalare le applicazioni nel tuo cluster. È possibile ridimensionare un'applicazione modificando manualmente il numero di repliche all'interno di un controller come un Deployment, per esempio.

È anche possibile modificare il ReplicaSet agendo sul numero di repliche desiderate, ma non è consigliabile gestire le applicazioni con questa metodologia: ricordiamo infatti che la gestione dei macro attributi di un'applicazione è contenuta nei controller, ed è a queste risorse che andrebbe affidata l'informazione sul numero di repliche. Lo scaling manuale va benissimo per quelle applicazioni che non subiscono fluttuazioni nei carichi di lavoro o quando si conoscono gli orari in cui questi aumentano, ma nel caso in cui si debbano gestire anche picchi improvvisi o una domanda variabile, il ridimensionamento manuale non è l'ideale. Fortunatamente, Kubernetes fornisce dei meccanismi automatici per ridimensionare i carichi di lavoro al posto nostro.

Diamo prima un'occhiata a come ridimensionare manualmente un Deployment esaminando il seguente YAML.

Listato 14.1 Esempio di Deployment con 3 repliche

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
      ports:
        - containerPort: 80
```

In questo esempio viene creato un Deployment denominato `nginx-deployment`, il cui nome diventerà la base per definire i ReplicaSet e i Pod creati successivamente. Il Deployment creerà infatti un ReplicaSet con tre Pod replicati, come indicato nel campo `replicas`.

NOTA

In che modo avviene l'associazione tra ReplicaSet, Pod e Deployment? Il campo `.spec.selector` definisce come il ReplicaSet creato deve trovare i Pod da gestire. In questo caso, seleziona una label definita nel template del Pod (per esempio, `app: nginx`), anche se sono possibili regole di selezione più sofisticate.

Per scalare manualmente il Deployment a 4 repliche, è possibile utilizzare il seguente comando.

Listato 14.2 Esempio di scale up del Deployment a 4 repliche

```
kubectl scale deployment nginx-deployment --replicas 4
```

Abbiamo dunque specificato tramite `scale` l'operazione che vogliamo eseguire sul numero dei Pod, il tipo di controller da utilizzare (in questo caso, un Deployment) con il relativo nome (ossia `nginx-deployment`) e infine il numero di repliche che vogliamo ottenere.

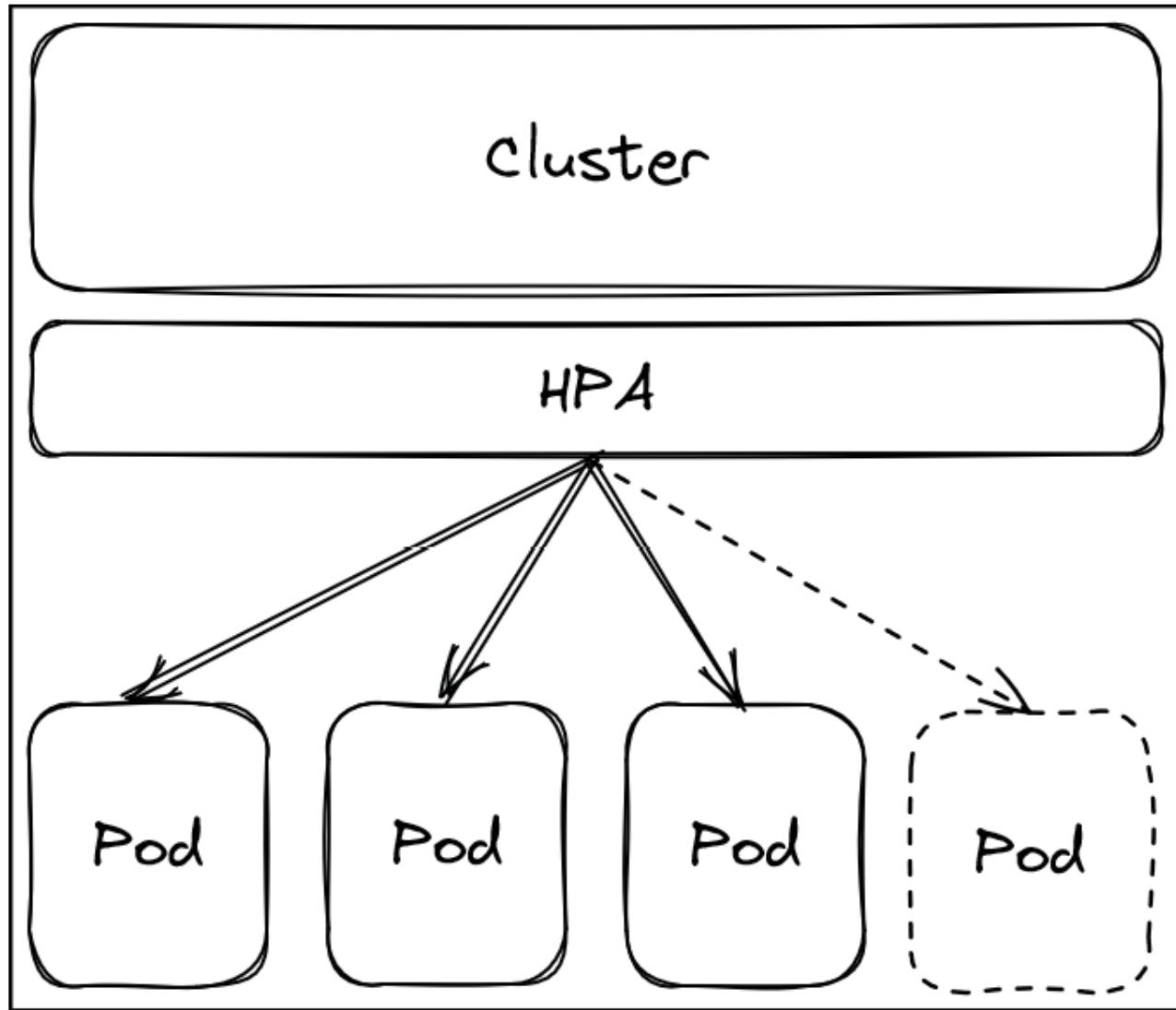


Figura 14.2 In che modo si interfaccia un HPA con dei Pod.

Ciò si traduce in quattro repliche del nostro servizio di server Nginx, il che è perfetto per gestire un carico di lavoro in graduale aumento. Lo stesso comando può essere utilizzato per scalare oggetti diversi, come uno StatefulSet: nell'esempio seguente, vediamo anche come specificare la tipologia di risorsa e il relativo nome con una sintassi leggermente diversa, ossia

`tipo_controller/nome_controller:`

Listato 14.3 Esempio di scale up di uno StatefulSet a 3 repliche

```
kubectl scale --replicas=3 sts/my-statefulset
```

Tra le altre cose, è possibile anche scalare il numero di Pod specificando come condizione che questa operazione avvenga se esiste già un certo numero di repliche: questo caso è mostrato nel seguente esempio.

Listato 14.4 Esempio di scale up del Deployment a 5 repliche se sono già presenti 3

```
kubectl scale --current-replicas=3 --replicas=5 deployment/my-deployment
```

Inoltre, è possibile scalare più risorse nello stesso comando, o anche tutti i controller presenti in un certo namespace:

Listato 14.5 Esempio di scale up di più risorse contemporaneamente

```
kubectl scale --replicas=5 deployment/my-deployment1 deployment/my-deployment2
```

Listato 14.6Esempio di scale up di tutte le risorse di tipo Deployment presenti nel namespace myproject a 5 repliche

```
kubectl scale --all --replicas=5 --namespace=myproject deployment
```

Ora diamo un'occhiata a come possiamo aggiungere un po' di "intelligenza" e ridimensionare automaticamente l'applicazione in base a diverse metriche.

Scalabilità automatica

Abbiamo introdotto le diverse modalità con cui si può scalare una risorsa all'interno del mondo Kubernetes: è importante però, prima di parlare di quali oggetti servono allo scopo, fare un ulteriore approfondimento su quelle che sono le diverse opzioni che abbiamo in termini di scalabilità automatica; questo perché, nel mondo enterprise, quando viene configurato un ambiente, difficilmente possiamo immaginarc di avere "qualcuno" che stia di fronte a uno schermo a monitorare il carico di lavoro di una certa applicazione in termini di risorse, per poter agire in tempo reale nel caso in cui il flusso cambi di intensità. Allo stesso modo, ci si augura sempre che le stime in termini di risorse iniziali siano sottostimate perché, se l'applicazione richiede più risorse nel corso del tempo, magari abbiamo avuto una crescita nell'utilizzo del sistema. Avere delle risorse "inutilizzate", comporta dei costi, soprattutto se il nostro cluster si trova all'interno di un ambiente cloud: dobbiamo quindi tenere fede a due motivazioni principali quando parliamo di scalabilità automatica, ossia la gestione del costo e la capacità applicativa, che ci permettono sia di calibrare in modo quanto più fedele possibile le risorse necessarie per tenere attivo il cluster, sia di adattarci a una capacità che potrebbe variare nel tempo, crescendo o stabilizzandosi su un set di risorse minimo.

La scalabilità automatica è interessante per le applicazioni che hanno delle fluttuazioni del carico di lavoro e del traffico. Senza ricorrere alla scalabilità automatica, ci sono due opzioni:

- provisioning in eccesso della capacità dell'applicazione, con costi aggiuntivi per l'azienda;
- avvisare ingegneri e ingegnere di tenersi pronti/e per le operazioni di ridimensionamento manuale, che comportano un effort notevole.

La scalabilità automatica, per tutte le componenti che sono scalabili, è quindi sicuramente una buona alternativa per rendere il nostro flusso di gestione del cluster più *responsivo* a picchi di lavoro imprevisti e per contenere i costi: per comprendere meglio le risorse Kubernetes di cui andremo a parlare, divideremo la scalabilità in diverse categorie:

- *scaling del carico di lavoro*, ossia gestire in maniera automatica la capacità delle singole componenti applicative;
- *scaling del cluster*, ossia gestire in maniera automatica la capacità di lavoro della piattaforma che ospita il nostro lavoro.

Ognuna di queste tipologie permette a sua volta una scalabilità che segue due approcci: scalabilità *orizzontale* e *verticale*, a seconda delle metriche utilizzate per cambiare la capacità delle risorse utilizzate.

HorizontalPodAutoscaler

Questa modalità comporta la modifica del numero di repliche di un carico di lavoro, per esempio il numero di Pod all'interno di un controller, piuttosto che il numero di nodi del cluster che ospita le nostre applicazioni. Viene solitamente abbreviato in HPA ed è perfetto quando il livello di utilizzo di un'applicazione cambia e c'è la necessità di aggiungere o rimuovere componenti che siano in grado di gestire il carico di lavoro che fluttua.

Questo componente determina il numero di Pod necessari in base alle metriche impostate dall'utente e applica la creazione o l'eliminazione di Pod in base a un set di soglie. Nella maggior parte dei casi, queste metriche sono rappresentate dall'utilizzo della CPU e della RAM, ma è anche possibile specificare le proprie metriche a seconda di alcune informazioni personalizzate. L'HPA controlla continuamente le metriche di CPU e memoria generate dal server di monitoraggio delle metriche presente nel cluster Kubernetes.

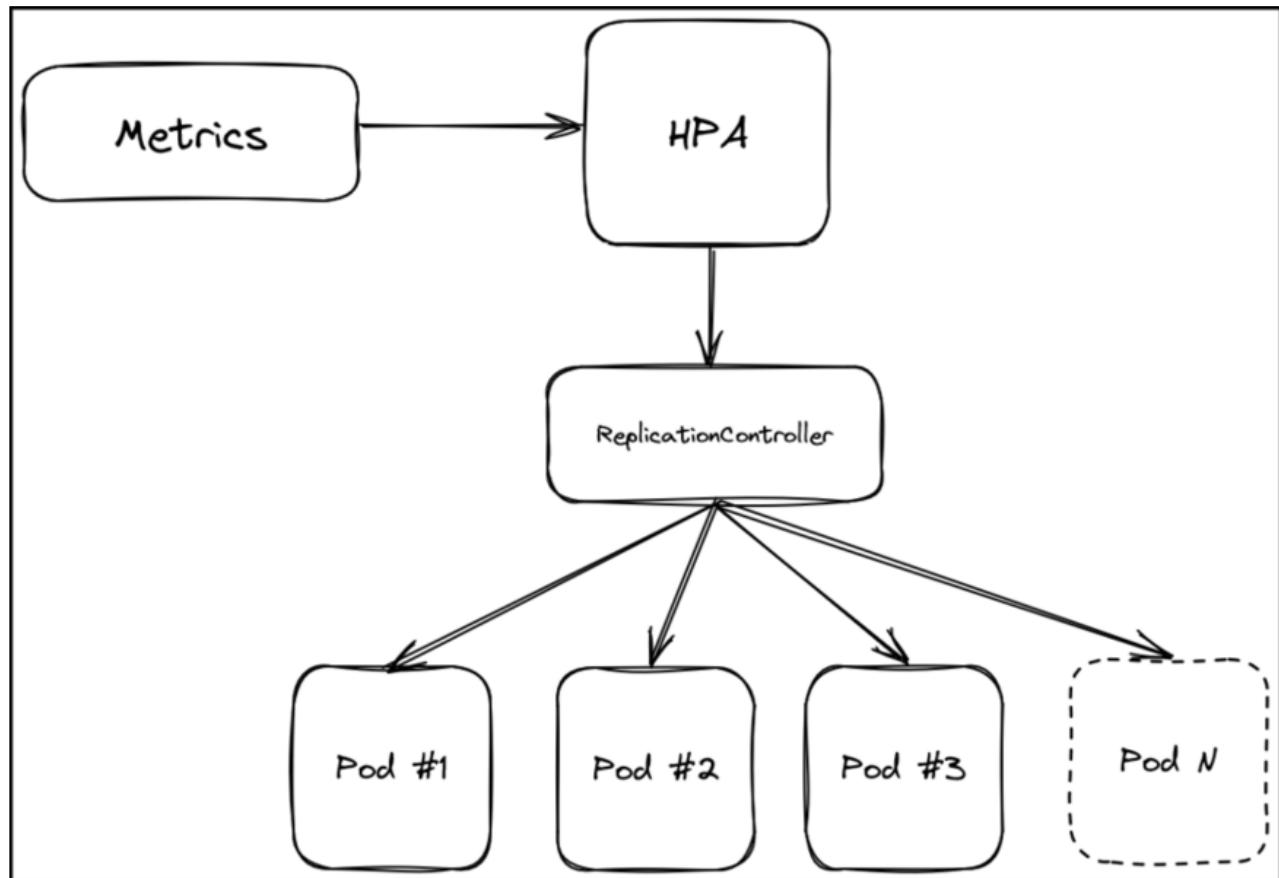


Figura 14.3 Come le metriche aiutano l'HPA per prendere delle decisioni su come gestire la numerosità dei Pod.

Se una delle soglie specificate viene soddisfatta, aggiorna il numero di repliche di Pod all'interno del controller di distribuzione (come può essere un Deployment). Questa risorsa non può essere utilizzata su oggetti che non possono scalare, come i DaemonSet. HPA può essere utile sia per le applicazioni statelessche per i carichi di lavoro stateful. Questa risorsa è gestita dal controllerKubernetes e viene eseguita in modo da controllare ciclicamente lo stato delle risorse replicate. Il controllerdi Kubernetes fornisce un flag che specifica la durata del controllo, che per impostazione predefinita è di 15 secondi.

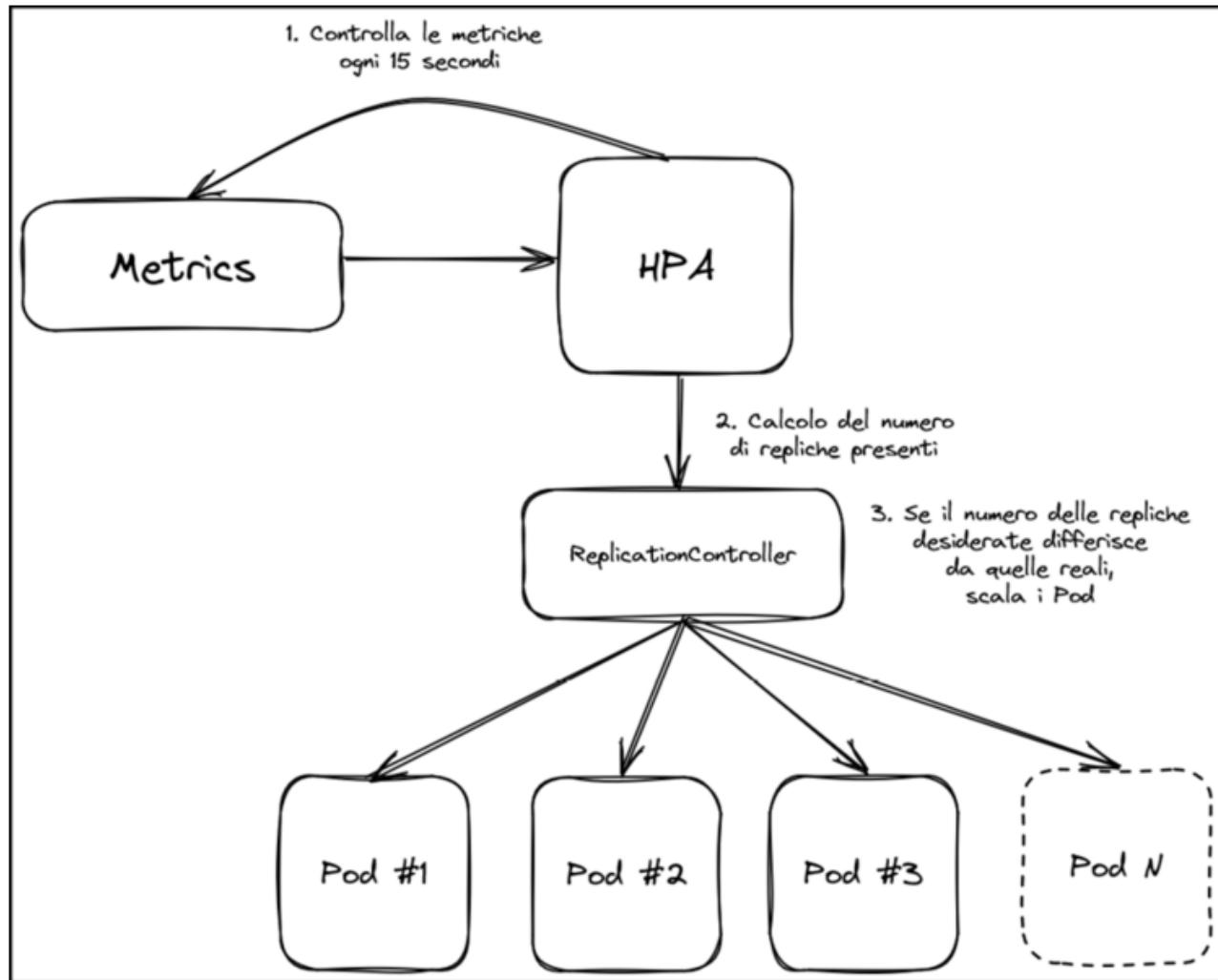


Figura 14.4 Funzionamento dell'HPA ad alto livello.

Dopo ogni periodo, il controller confronta l'effettivo utilizzo delle risorse con le metriche definite per ogni HPA, ottenendo le informazioni di cui ha bisogno dalle API delle metriche personalizzate o, se specificato che il ridimensionamento automatico deve essere basato sulle risorse per Pod (come l'utilizzo della CPU), dalle API delle metriche delle risorse.

Un esempio di HPA basato su delle metriche di base è riportato di seguito.

Listato 14.7 HPA basato sulla CPU percentuale

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample
spec:
  selector:
    matchLabels:
  
```

```

        app: sample
template:
  metadata:
    labels:
      app: sample
spec:
  containers:
    - name: sample
      image: sample-image:1.0
      resources:
        requests:
          cpu: "100m"
---
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: sample
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: sample
minReplicas: 1
maxReplicas: 3
targetCPUUtilizationPercentage: 75

```

Vediamo alcuni dei parametri di questo YAML: come prima cosa, è necessario impostare un valore `resources.requests` per la metrica, tra CPU e RAM, che sceglieremo di utilizzare. La voce `minReplicas` rappresenta il numero di repliche sotto il quale non dovrà mai essere ridimensionato il Deployment, così come `maxReplicas` rappresenta il valore massimo.

Per tarare il numero di Pod presenti nel Deployment, l'oggetto HPA andrà a basarsi sulla proprietà `targetCPUUtilizationPercentage`, ossia l'utilizzo della CPU in percentuale basato sulla request specificata. Come detto in precedenza, se l'utilizzo effettivo della CPU va significativamente oltre questo valore, il numero di repliche verrà aumentato; se si trova al di sotto, sarà diminuito.

Tra le opzioni, c'è anche la possibilità di specificare un valore numerico per la CPU o memoria, piuttosto che il valore medio; ovviamente, è necessario anche indicare qual è l'oggetto che l'Autoscaler deve monitorare, specificando tipologia e nome

Listato 14.8 HPA basato sulla CPU media

```

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: sample-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: sample
  minReplicas: 1
  maxReplicas: 3
  metrics:
    - type: Resource
      resource:
        name: cpu
      target:
        type: Utilization
        averageUtilization: 50

```

Targets Unknown

Quando viene creato l'HPA, potremmo voler vedere il suo stato tramite il comando `kubectl get hpa`; il valore della colonna `TARGETS` visualizzerà `<unknown>/50%` per circa 15 secondi. Si tratta di un comportamento normale, perché HPA deve raccogliere dei valori medi in un certo intervallo di tempo e non avrà dati sufficienti prima di 15 secondi, che sono configurati come impostazione predefinita.

Se volessimo monitorare lo stato attuale dell'Autoscaler, potremmo visualizzarne una istantanea con il comando `kubectl describe`, come mostrato di seguito.

Listato 14.9 Stato dell'HPA configurato

```
kubectl describe hpa sample-hpa
---
Name:                      sample-hpa
Namespace:                  default
Labels:                     <none>
Annotations:                <none>
...
Reference:                 Deployment/sample
Metrics:                    ( current / target )
  resource cpu on pods  (as a percentage of request): 240% (48m) / 50%
Min replicas:               1
Max replicas:               3
Deployment pods:            3 current / 3 desired
...
Events:
  Type   Reason          Age     From           Message
  ----  -----          ----   ----
  Normal SuccessfulRescale 17s   horizontal-pod-autoscaler  New size: 2; reason: cpu
resource utilization (percentage of request) above target
  Normal SuccessfulRescale 37s   horizontal-pod-autoscaler  New size: 3; reason: cpu
resource utilization (percentage of request) above target
```

Nella sezione relativa agli eventi, possiamo leggere i cambiamenti di stato del ciclo di vita di questo oggetto: per esempio, il Deployment che l'Autoscaler sta monitorando sembra aver utilizzato più del 50% di CPU, motivo per cui l'HPA si è attivato e ha incrementato il numero di Pod replicati presenti al suo interno.

L'HPA prende una decisione in base ai valori osservati di utilizzo della CPU o della memoria dei Pod in un cluster e abbiamo visto nell'esempio precedente che i valori di utilizzo vengono calcolati come percentuale della richiesta di risorse di ciascun Pod.

Quel che succede molto spesso è che potrebbe verificarsi un cambiamento piuttosto rapido e discontinuo del carico di lavoro dell'applicazione, che porterebbe l'Autoscaler a dover incrementare e decrementare il numero di Pod continuamente. Configurare un periodo di osservazione del carico di lavoro può essere una buona soluzione per valutare se effettivamente l'Autoscaler debba entrare in azione: diciamo che, per esempio, se si verifica un picco per più di 60 secondi, allora vogliamo che il numero di repliche del controller che ospita i nostri Pod aumenti, altrimenti continuiamo a monitorare la situazione in attesa di cambiamenti. Questo tipo di controllo può essere gestito grazie al campo `stabilizationWindowSeconds`: viene infatti utilizzato per limitare la fluttuazione del numero delle repliche quando le metriche subiscono dei cambiamenti rapidi e variabili.

Listato 14.10 HPA con stabilizationWindowSeconds

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: sample-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
```

```

name: sample
behavior:
  scaleUp:
    stabilizationWindowSeconds: 60
minReplicas: 1
maxReplicas: 3
metrics:
- type: Resource
  resource:
    name: cpu
    target:
      type: Utilization
      averageUtilization: 50

```

In questo caso, l'Autoscaler si attiverà non appena il valore soglia specificato per la CPU verrà superato e attenderà 60 secondi prima di aumentare il numero di repliche, per limitare le decisioni sul ridimensionamento, osservando i dati storici per un certo periodo di tempo e mantiene costante il numero di repliche in caso di metriche fluttuanti.

Esiste poi un'altra possibilità: quella di personalizzare il tipo di comportamento, sia per l'aumento che per la diminuzione dei Pod, con delle *policy* che vadano a valutare diverse metriche dell'HPA. Vediamo qualche esempio.

Listato 14.11 HPA con stabilizationWindowSeconds

```

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: sample-app-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: sample-app
  minReplicas: 1
  maxReplicas: 100
  behavior:
    scaleDown:
      stabilizationWindowSeconds: 60
      selectPolicy: Max
    policies:
    - type: Pods
      value: 4
      periodSeconds: 60
      - type: Percent
        value: 10
        periodSeconds: 60
  metrics:
  - type: Resource
    resource:
      name: cpu
    target:
      type: Utilization
      averageUtilization: 50

```

Nell'esempio precedente, se sono in esecuzione 100 repliche, l'HPA esaminerà le due policy disponibili: la prima policy indica all'Autoscaler di rimuovere 4 Pod per volta in un periodo di 60 secondi, mentre la seconda policy indica di rimuovere il 10% del numero di repliche presenti, sempre in una finestra di 60 secondi. Il secondo criterio ha l'impatto maggiore, poiché rimuoverebbe 10 Pod nella prima finestra di 60 secondi, per cui, avendo selezionato come policy quella che rimuove il maggior numero di Pod (vedi campo `selectPolicy`), la seconda policy avrà la meglio.

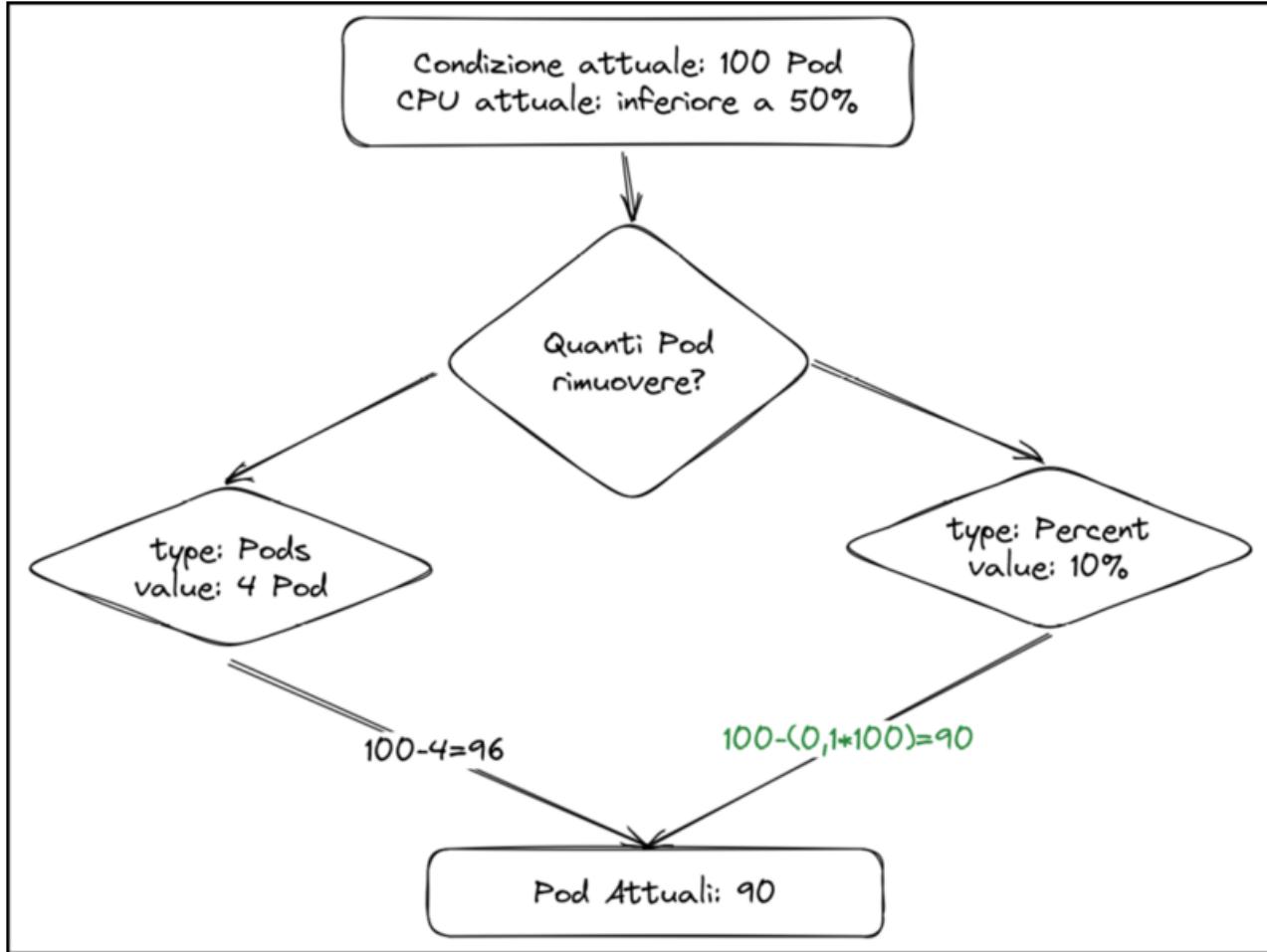


Figura 14.5 Come funzionano le diverse condizioni espresse nell'autoscaler.

Nella successiva finestra di 60 secondi, 9 Pod (ossia il 10%) verranno rimossi dai restanti 90. Nella terza iterazione, 9 Pod verranno nuovamente rimossi, poiché 9 è il risultato maggiore tra le due condizioni, essendo il 10% dei Pod attualmente presenti. Quando raggiunge 40 repliche, subentrerà la prima policy, poiché la seconda policy avrebbe un impatto inferiore rispetto a 4 Pod. Pertanto, da 40 Pod in poi, l'Autoscaler rimuove continuamente 4 Pod a ogni iterazione.

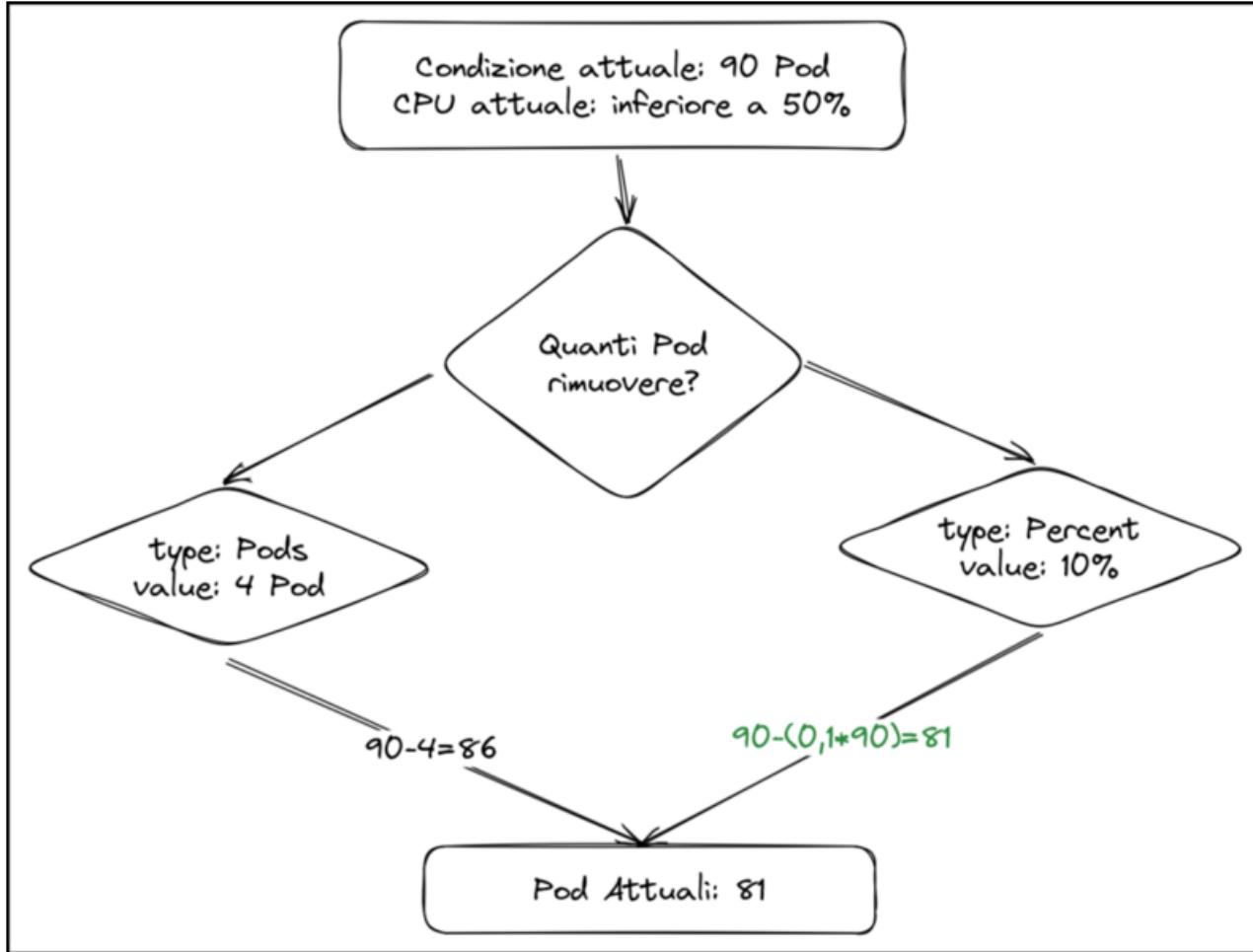


Figura 14.6 Aggiornamento delle condizioni dopo le prime iterazioni

Oltre a quanto visto finora, è inoltre possibile specificare le metriche delle risorse in termini di valori diretti, anziché come percentuali del valore richiesto, utilizzando un `target.type` di `AverageValue` invece di `Utilization` e impostando il campo `target.averageValue` corrispondente invece di `target.averageUtilization`.

Listato 14.12 HPA con un valore di utilizzo deterministico

```

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: sample-app-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: sample-app
  minReplicas: 1
  maxReplicas: 100
...
metrics:
...
- type: Resource
  resource:
    name: memory
    target:
  
```

```

type: AverageValue
averageValue: 500Mi

```

Custom metrics

Abbiamo detto che la scalabilità automatica è un approccio che ci permette di aumentare o ridurre automaticamente il numero di Pod in base al carico di lavoro; l'intero meccanismo di scalabilità automatica si basa sulle metriche che rappresentano il carico corrente di un'applicazione. Ma come fa un HPA a sapere come ottenere queste metriche? C'è un componente fondamentale in gioco: `Metrics API`. L'API Metrics e la pipeline che abilita offrono solo le metriche minime di CPU e memoria per abilitare il ridimensionamento automatico tramite HPA e/o VPA, per cui, se vuoi avere un set di metriche più completo, puoi integrare l'API delle metriche distribuendo una seconda pipeline che utilizza le *Custom Metrics API*. Abbiamo visto in precedenza il funzionamento dell'HPA e di come questa risorsa ricavi informazioni circa le metriche delle applicazioni interrogando ogni 15 secondi il server, e verificando che il numero di repliche presenti sia corrispondente a quello desiderato. In realtà, dietro alle quinte delle API che gestiscono le metriche, ci sono diversi componenti che concorrono a raccogliere queste informazioni:

- *cAdvisor*: demone per la raccolta, l'aggregazione e l'esposizione delle metriche dei container;
- *kubelet*: agente distribuito su tutti i nodi per la gestione delle risorse del container;
- *Summary API*: API fornita da *kubelet* per il recupero delle statistiche di riepilogo per i nodi;
- *metrics-server*: componente aggiuntivo del cluster che raccoglie e aggrega le metriche delle risorse estratte da ogni *kubelet*. Il server espone l'*API Metrics* per l'utilizzo da parte di HPA, VPA e del comando `kubectl top`, che permette la visualizzazione delle risorse applicative.
- *Metrics API*: API che supporta l'accesso alla CPU e alla memoria utilizzate per la scalabilità automatica.

Tutti questi componenti costituiscono il *Metrics Registry*, il quale elabora quindi le metriche di default, ma eventualmente anche metriche custom e/o esterne. L'API è stata infatti costruita proprio per adattarsi a questa flessibilità di intenti, e per poter aumentare queste proprietà tramite delle estensioni.

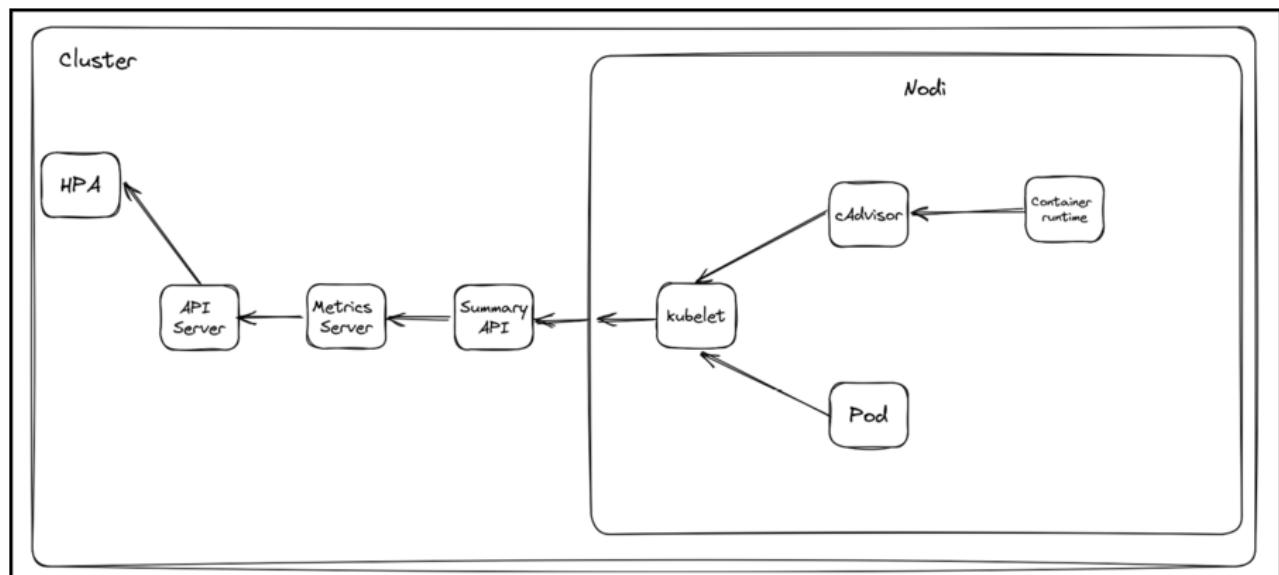


Figura 14.7 Come si interfacciano i diversi componenti.

Questo vuol dire che esistono delle estensioni per le API core di Kubernetes che possono essere rese accessibili attraverso il server che espone queste API. Pertanto, possiamo immaginare il registry come qualcosa di estensibile per aggiungere altre informazioni che possono essere utilizzate come sorgente per le risorse che devono leggerle e prendere delle decisioni sugli oggetti che gestiscono. Per raggiungere questo risultato, sarà ovviamente necessario che un server esponga queste informazioni, così come bisogna aggiungere un componente che possa leggere questi dati ed esporli correttamente. Una delle scelte più diffuse per la parte di raccolta dei dati, custom ed esterni, è Prometheus: questo software permette, attraverso un *adapter* che si integra con il componente principale, di raccogliere le metriche e di renderle fruibili. Per far sì che questo giro funzioni, sarà quindi necessario installare in primis un aggregatore di metriche, come Prometheus, o anche DataDog, configurarlo affinché raccolga i dati dai Pod o dalle risorse che li espongono, e poi installare un server per le API (come il Prometheus adapter) e definire in che modo esporre queste metriche. Un'ipotesi di architettura potrebbe essere rappresentata dalla Figura 14.8.

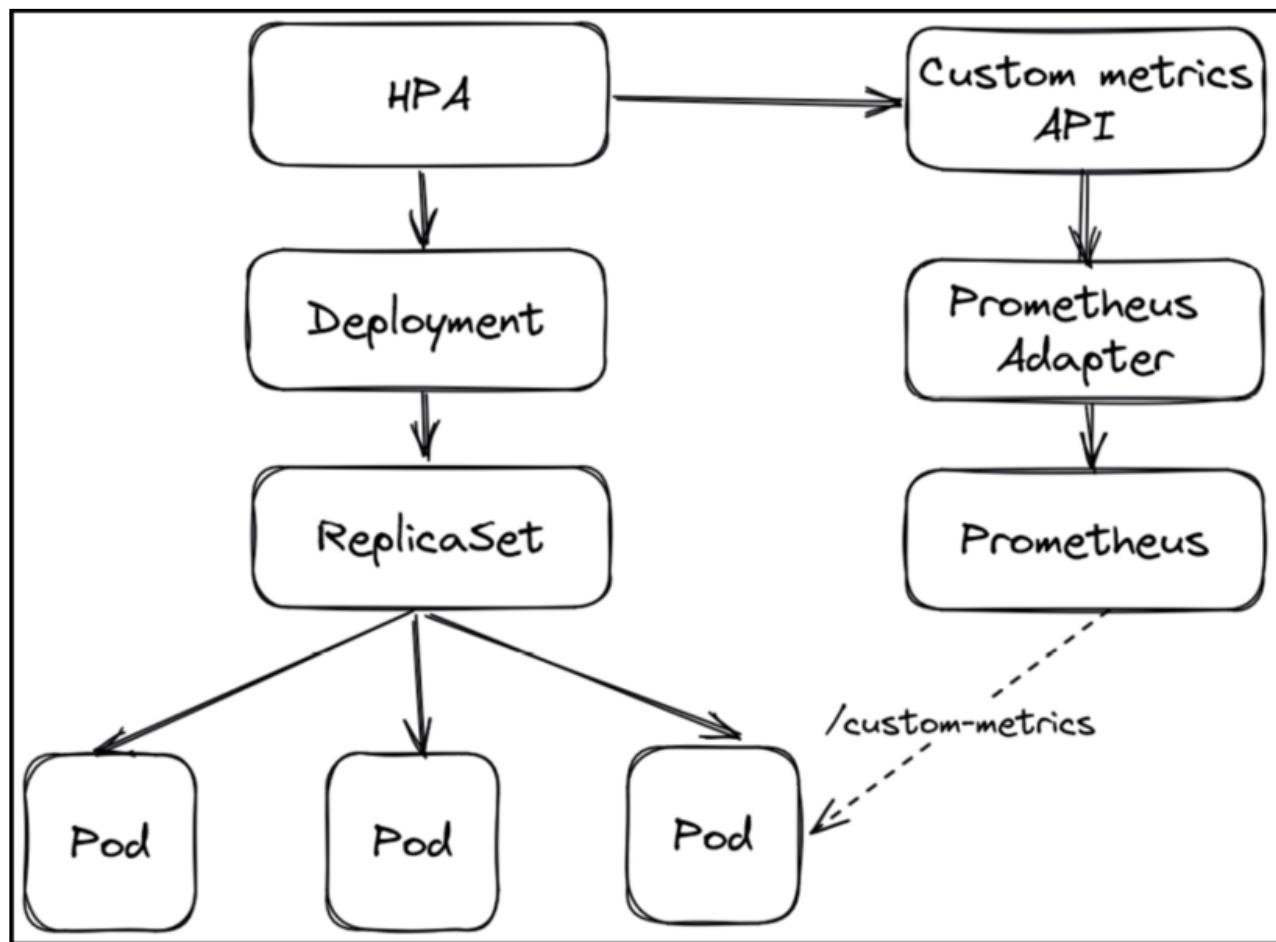


Figura 14.8 Come i diversi componenti applicativi si relazionano con quelli relativi alle metriche.

Inoltre, esistono altri due tipi di metriche, entrambe considerate metriche custom: quelle del Pod e quelle relative agli altri oggetti del namespace. Il primo tipo serve a descrivere i Pod e si ottiene calcolando la media tra i Pod presenti per poterli confrontare con un valore target e determinare il conteggio delle repliche. Funzionano in modo molto simile alle metriche delle risorse, tranne per il fatto che supportano solo il campo `AverageValue`. In altre parole, se volessimo specificare, per esempio, il numero medio di pacchetti per secondo ricevuti dai Pod come metrica per attivare l'Autoscaler, potremmo configurare un oggetto di questo tipo:

Listato 14.13 HPA con metriche custom basate sul Pod

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: sample-app-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: sample-app
  minReplicas: 1
  maxReplicas: 100
...
metrics:
...
- type: Pods
  pods:
    metric:
      name: packets-per-second
    target:
      type: AverageValue
      averageValue: 1k
```

Se invece volessimo utilizzare un oggetto diverso dai Pod per ottenere delle metriche, potremmo utilizzare le metriche il cui `type` è `Object`. Le metriche non vengono necessariamente recuperate dall'oggetto; lo descrivono solo. Le metriche degli oggetti supportano sia dei valori puntuali (tramite `.type: Value`), sia un valore medio (tramite `AverageValue`). L'esempio seguente rappresenta un HPA con delle metriche basate sul numero di request per secondo che vengono valutate tramite l'oggetto Ingress.

Listato 14.14 HPA con metriche custom basate su una risorsa

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: sample-app-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: sample-app
  minReplicas: 1
  maxReplicas: 100
...
metrics:
...
type: Object
object:
  metric:
    name: requests-per-second
    describedObject:
      apiVersion: networking.k8s.io/v1
      kind: Ingress
      name: main-route
    target:
      type: Value
      value: 2k
```

VerticalPodAutoscaler

Abbreviato in VPA, può allocare più (o meno) CPU e risorse di memoria ai Pod esistenti per modificare le risorse di calcolo disponibili per un'applicazione. Questa funzionalità può essere utile per gestire e regolare le risorse allocate di ciascun Pod nel corso della sua durata. Il VPA viene fornito con uno strumento chiamato *VPA Recommender*, che monitora il consumo di risorse attuale e passato e utilizza questi dati per fornire risorse di CPU e memoria consigliate da allocare per i container. Il Vertical Pod Autoscaler non aggiorna le configurazioni delle risorse per i Pod esistenti, ma controlla quali Pod hanno la corretta configurazione delle risorse e rimuove quelli che non hanno la configurazione consigliata in modo che i loro controller possano ricrearli con la configurazione aggiornata.

Listato 14.15 VPA basato sulla CPU

```
apiVersion: v1
kind: Pod
metadata:
  name: sample
spec:
  containers:
    - name: sample
      image: sample-image:1.0
  resources:
    requests:
      cpu: 100m
      memory: 50Mi
    limits:
      cpu: 100m
      memory: 50Mi
---
apiVersion: "autoscaling.k8s.io/v1beta2"
kind: VerticalPodAutoscaler
metadata:
  name: sample
spec:
  targetRef:
    apiVersion: "v1"
    kind: Pod
    name: sample
  resourcePolicy:
    containerPolicies:
      - containerName: '*'
        minAllowed:
          cpu: 100m
          memory: 50Mi
        maxAllowed:
          cpu: 1
          memory: 500Mi
        controlledResources: ["cpu", "memory"]
  updatePolicy:
    updateMode: Recreate
```

Vediamo alcune delle proprietà presenti in questo esempio: `containerName` rappresenta il nome del container a cui applicare il VPA, che secondo questo criterio di ridimensionamento applicherà quanto previsto a ogni container. Inoltre, grazie a `minAllowed` e `maxAllowed`, le `requeste` i `limit` delle risorse non saranno impostate al di sotto o al di sopra di questi valori, seguendo quanto indicato nelle risorse gestite (tramite `controlledResources`).

Ci sono invece tre opzioni per la proprietà `updateMode`. La modalità di ricreazione (ossia `Recreate`) attiverà il ridimensionamento automatico, con la creazione ex-novo delle risorse. La modalità

Initial applicherà il controllo di ammissione specificato nelle proprietà descritte in precedenza per impostare i valori delle risorse al momento della creazione, ma non eliminerà mai alcun Pod. La modalità `off` consigliera i valori delle risorse ma non li modificherà mai automaticamente e infine `Auto` procederà in autonomia. Usarlo in modalità `off` può essere estremamente utile: sebbene sia consigliabile e preferibile eseguire test di carico completi delle applicazioni prima che queste vadano in produzione, non sempre è possibile. Ciò porta comunemente a risorse sovradimensionate come misura di sicurezza, che spesso si traducono in uno scarso utilizzo dell'infrastruttura. In questi casi, il VPA può essere utilizzato per consigliare valori che vengono poi valutati e aggiornati manualmente dalle persone che ci lavorano una volta applicato il carico di produzione. Ciò garantisce loro la tranquillità che i carichi di lavoro non vengano eliminati nei periodi di utilizzo di punta, il che è particolarmente importante se un'app non termina correttamente. Tuttavia, poiché il VPA consiglia dei valori basati sui dati a sua disposizione, risparmia parte della fatica nella revisione delle metriche di utilizzo delle risorse e nella determinazione dei valori ottimali. In questo caso d'uso, non viene infatti considerato un Autoscaler, ma piuttosto un aiuto per l'ottimizzazione delle risorse.

Per ottenere consigli da un VPA in modalità `off`, è possibile eseguire il seguente comando; questo fornirà raccomandazioni per ogni container. Utilizzare il valore `Target` come raccomandazione di base per le richieste di CPU e memoria.

Listato 14.16 VPA in modalità Off

```
kubectl describe vpa NOME_VPA
>>>
...
Recommendation:
  Container Recommendations:
Container Name: mycontainer
Lower Bound:
Cpu: 25m
Memory: 262144k
Target:
Cpu: 25m
Memory: 262144k
Uncapped Target:
Cpu: 25m
Memory: 262144k
Upper Bound:
Cpu: 427m
Memory: 916943343
...
```

Quando si desidera utilizzare HPA e VPA contemporaneamente per gestire le risorse del container, è possibile che vadano in conflitto tra loro quando si utilizzano le stesse metriche (CPU e memoria). Entrambi cercheranno infatti di risolvere la situazione contemporaneamente, ottenendo come risultato un'errata allocazione delle risorse. Tuttavia, è possibile utilizzarli entrambi se si basano su metriche diverse, per esempio delle metriche custom per l'HPA basate sul numero di thread gestiti dal Pod. Il VPA utilizza in effetti il consumo di CPU e memoria come fonti univoche per raccogliere la perfetta allocazione delle risorse, ma l'HPA può essere utilizzato con metriche personalizzate in modo che entrambi gli strumenti possano essere utilizzati in parallelo.

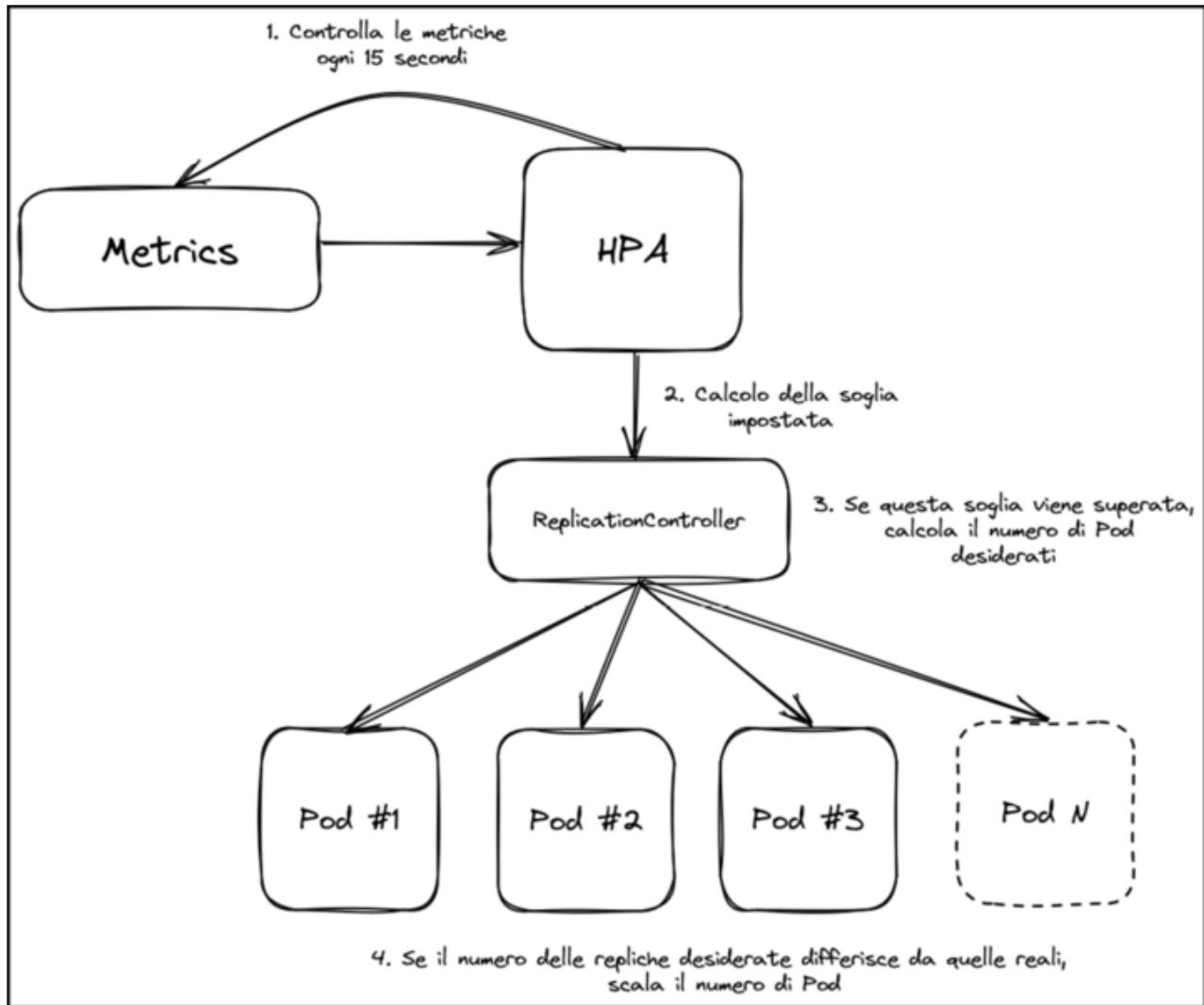


Figura 14.9 Funzionamento di un HPA.

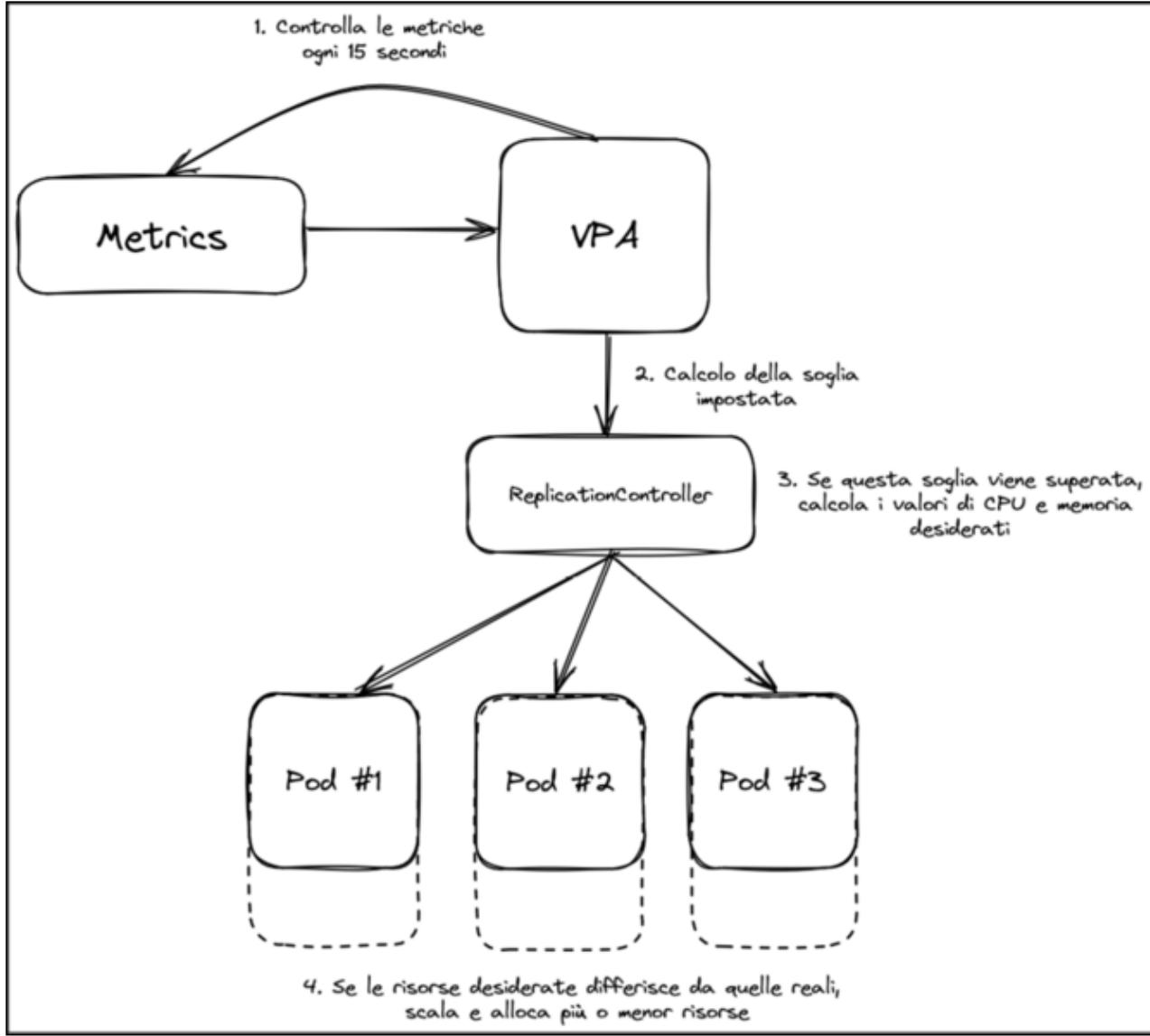


Figura 14.10 Funzionamento di un VPA.

Nei sistemi che necessitano di scalabilità, ovvero che presentano frequenti e significativi cambiamenti di carico, è bene preferire la scalabilità orizzontale ove possibile. La scalabilità verticale è limitata dalla macchina più grande che hai a disposizione e, inoltre, l'aumento della capacità con il ridimensionamento verticale comporta un riavvio delle applicazioni presenti.

Cluster Autoscaler

Una delle prime decisioni che devi prendere quando distribuisci un cluster è la dimensione delle istanze dei nodi che vorrai utilizzare al suo interno. Questa diventa più un'arte che una scienza, specialmente quando hai più carichi di lavoro in un singolo cluster. Dovrai prima identificare qual è un buon punto di partenza per il cluster; mirare a un buon equilibrio tra CPU e memoria è un'opzione. Dopo aver deciso una dimensione ragionevole per il cluster, è possibile utilizzare un paio di opzioni di Kubernetes per gestire il ridimensionamento del tuo cluster.

Il ridimensionamento *manuale* del cluster in genere consiste semplicemente nella scelta di un nuovo numero di nodi dove un servizio terzo aggiungerà i nuovi nodi al cluster in maniera trasparente. Questi

strumenti consentono inoltre di creare più nodi, per permettere l'aggiunta di nuovi tipi di istanza a un cluster già in esecuzione. Questo diventa molto utile quando si eseguono carichi di lavoro misti all'interno di un singolo cluster: per esempio, un carico di lavoro potrebbe essere più basato sulla CPU, mentre gli altri carichi di lavoro potrebbero essere applicazioni che tendono a sollecitare di più la memoria.

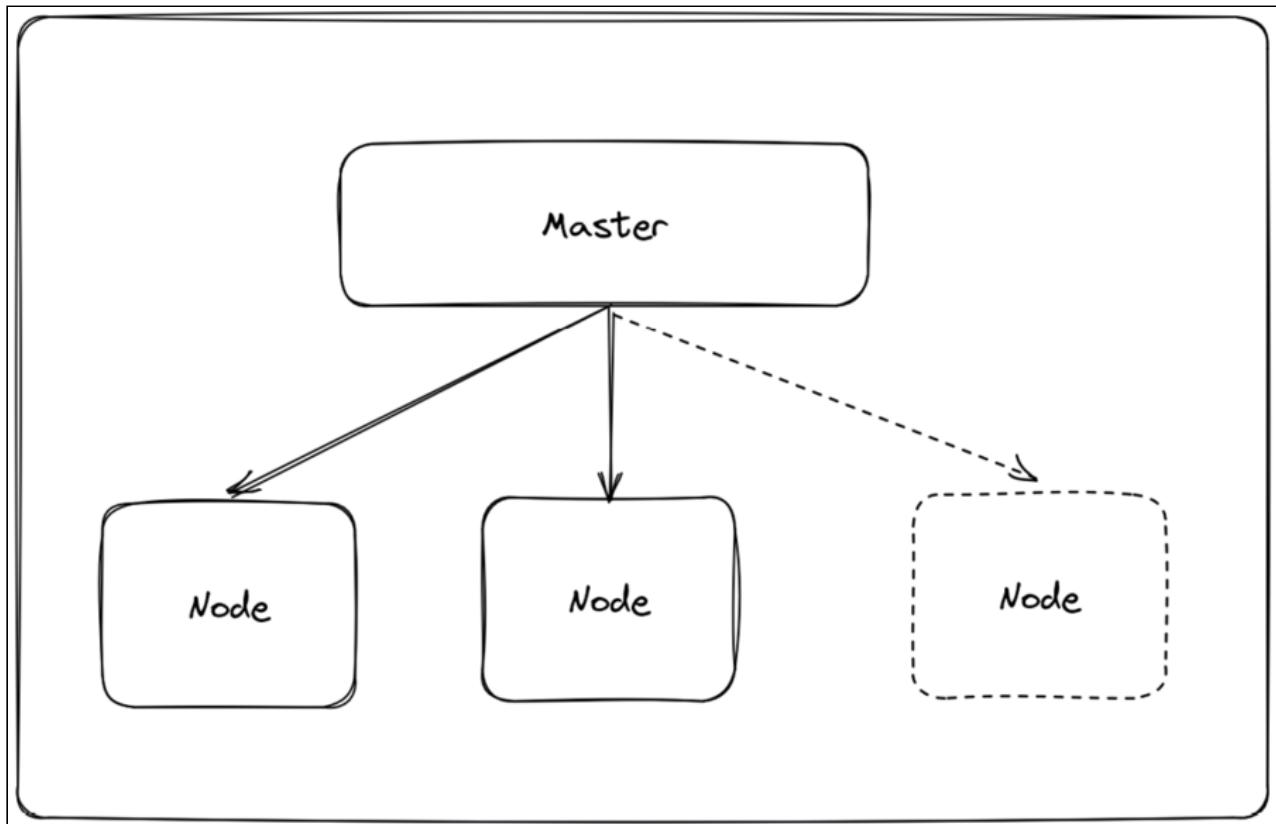


Figura 14.11 Rappresentazione astratta di un cluster Autoscaler.

La scalabilità manuale può sembrare, in questo caso, una cosa banale, ma ci sono considerazioni da fare prima di prendere in esame la scalabilità automatica del cluster. Kubernetes fornisce infatti un componente aggiuntivo per eseguire lo scaling automatico del cluster che ti consente di impostare il numero di nodi minimi disponibili per un cluster e anche il numero massimo a cui il tuo cluster può essere ridimensionato. In questo caso, l'Autoscaler basa la sua decisione su quando un Pod risulta in stato `Pending`. Per esempio, se lo scheduler Kubernetes tenta di istanziare un Pod che richiede 1 core di CPU, ma ne sono presenti 0.5, il Pod entrerà in stato `Pending` fintanto che non vi saranno risorse sufficienti.

A questo punto, entra in gioco il *Cluster Autoscaler*, che aggiungerà un nodo al cluster; non appena questo è funzionante; il Pod “sospeso” viene quindi istanziato. Lo svantaggio di Cluster Autoscaler è che un nuovo nodo viene aggiunto solo prima che un Pod “sospeso”, quindi il tuo carico di lavoro potrebbe finire per attendere che un nuovo nodo sia disponibile per poter procedere.

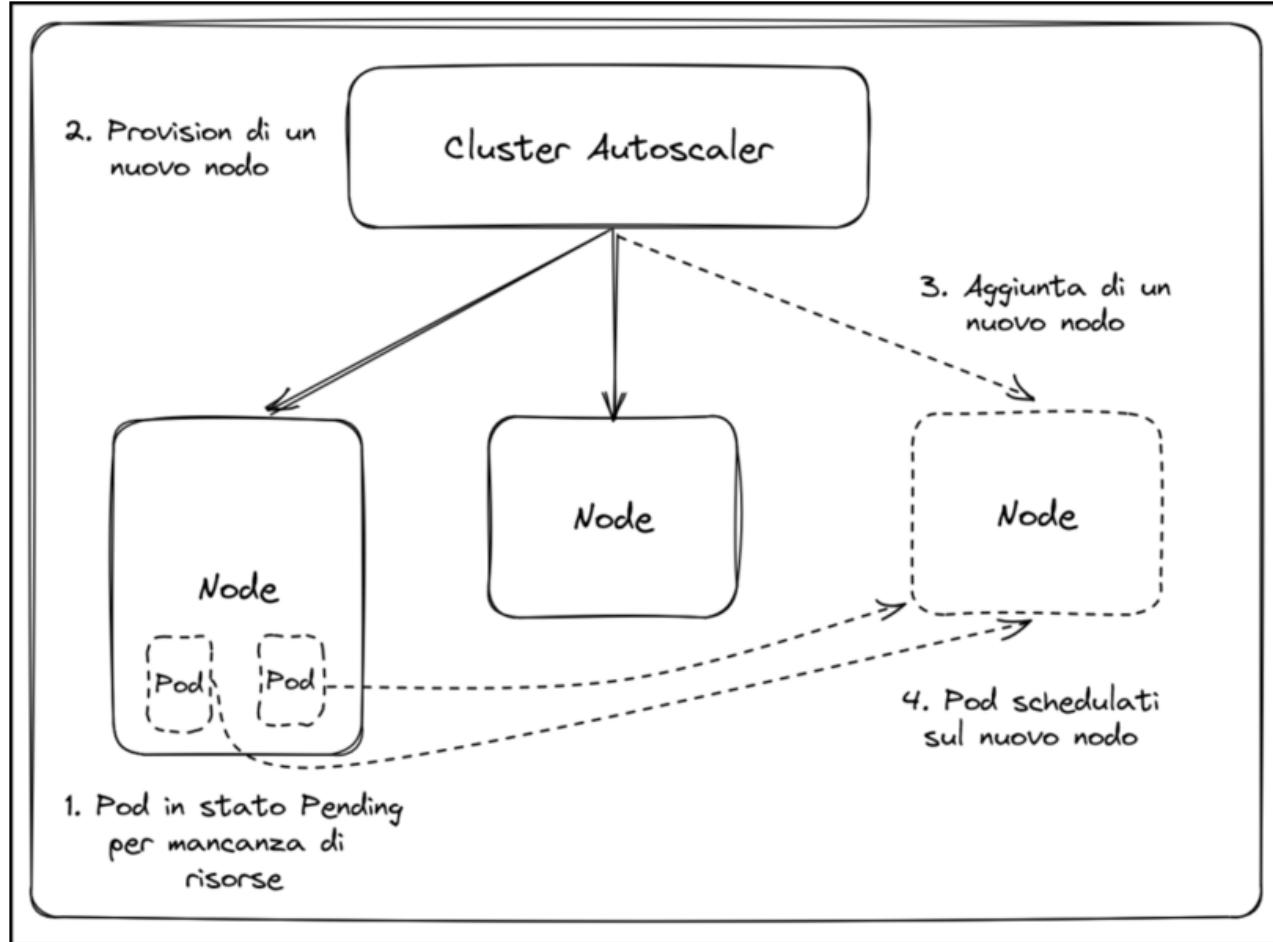


Figura 14.12 Funzionamento ad alto livello di come viene aggiunto un nodo al cluster nel caso quelli attuali non abbiano risorse sufficienti.

Il vantaggio è che il Cluster Autoscaler può anche ridurre la quantità dei nodi quando le risorse non sono più necessarie, “scaricando” il nodo (alias, *drain*) dei carichi di lavoro e pianificando la migrazione dei Pod in nuovi nodi nel cluster.

Un esempio di configurazione di un Autoscaler in AWS (che verrà trattato successivamente nel capitolo dedicato ai diversi cloud provider) è il seguente.

Listato 14.17 Configurazione di un'autoscaler in AWS

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: cluster-autoscaler
  namespace: kube-system
  labels:
    app: cluster-autoscaler
spec:
  replicas: 1
  selector:
    matchLabels:
      app: cluster-autoscaler
  template:
    metadata:
      labels:
        app: cluster-autoscaler
    annotations:

```

```

    prometheus.io/scrape: 'true'
    prometheus.io/port: '8085'
spec:
  priorityClassName: system-cluster-critical
  securityContext:
    runAsNonRoot: true
    runAsUser: 65534
    fsGroup: 65534
  serviceAccountName: cluster-autoscaler
  containers:
    - image: k8s.gcr.io/autoscaling/cluster-autoscaler:v1.22.2
      name: cluster-autoscaler
      resources:
        limits:
          cpu: 100m
          memory: 600Mi
        requests:
          cpu: 100m
          memory: 600Mi
      command:
        - ./cluster-autoscaler
        - --v=4
        - --stderrthreshold=info
        - --cloud-provider=aws
        - --skip-nodes-with-local-storage=false
        - --expander=least-waste
        - --nodes=1:10:worker-auto-scaling-group
  volumeMounts:
    - name: ssl-certs
      mountPath: /etc/ssl/certs/ca-certificates.crt
      readOnly: true
  imagePullPolicy: "Always"
volumes:
  - name: ssl-certs
    hostPath:
      path: "/etc/ssl/certs/ca-bundle.crt"

```

In questo esempio (non completo, attenzione) configureremo un Deployment che si occuperà di monitorare la situazione e scalare i nodi del cluster ove necessario; questo avverrà utilizzando un'immagine apposita (vedi `.containers.image[0]`), che ci permette di specificare alcuni parametri per lo script che si occuperà di gestire il numero di nodi come il provider da utilizzare, il minimo e massimo numero di nodi, e altri parametri.

NOTA

Questo oggetto non può, però, essere utilizzato così com'è: avrà infatti bisogno della configurazione degli opportuni permessi, dal momento che agisce sull'intero cluster: un esempio completo di tutte le risorse può essere trovato al seguente link: <https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler/cloudprovider>.

Una cosa da tenere a mente quando utilizzi questa modalità è che inevitabilmente andrà a distribuire i Pod tra i diversi nodi in maniera uniforme quando vengono creati per la prima volta; tuttavia, una volta che un Pod è in esecuzione, la decisione di istanziare un nuovo Pod in un secondo momento, potrebbe portare ad una situazione in cui potresti ritrovarti con Pod distribuiti in modo non uniforme tra i tuoi nodi di lavoro. In alcuni casi potresti ritrovarti con diverse repliche di una stessa applicazione su diversi nodi.

Un'altra considerazione relativa alla scalabilità automatica del cluster è la velocità con cui questo viene ridimensionato in caso di necessità: questo non avviene sempre in maniera trasparente, ed è qui che l'*overprovisioning* può aiutare. Ricordiamo infatti che il Cluster Autoscaler agisce nel momento in cui ci sono dei Pod in sospeso, che non è stato possibile istanziare a causa di risorse di calcolo insufficienti nel cluster. Pertanto, nel momento in cui questo interviene per ridimensionare i nodi del cluster, il tuo cluster è già pieno: ciò significa che, se non gestiti correttamente, l'applicazione potrebbe

subire un disservizio durante il tempo necessario affinché i nuovi nodi diventino disponibili per ospitare nuovi Pod.

Una soluzione potrebbe essere quella di combinare il Cluster Autoscaler con un HPA, impostandolo con dei valori sufficientemente bassi in modo che i tuoi carichi di lavoro vengano scalati ben prima che l'applicazione raggiunga la reale piena capacità. Ciò potrebbe fornire del margine, che consenta di eseguire il provisioning dei nodi in anticipo rispetto alla saturazione del cluster.

Un'altra soluzione consiste nell'utilizzare l'overprovisioning del cluster: con questo metodo, metti in standby dei nodi "vuoti" per fornire un margine per i carichi di lavoro che scalano orizzontalmente; ciò alleggerirà la necessità di gestire un HPA artificialmente basso in preparazione a eventuali picchi di lavoro. L'overprovisioning del cluster funziona distribuendo Pod che eseguono le seguenti operazioni:

- si richiedono le risorse sufficienti per riservare virtualmente tutte le risorse per un nodo;
- si utilizza una classe di priorità che porta alla messa a disposizione delle risorse non realmente utilizzate non appena un altro Pod ne ha bisogno.

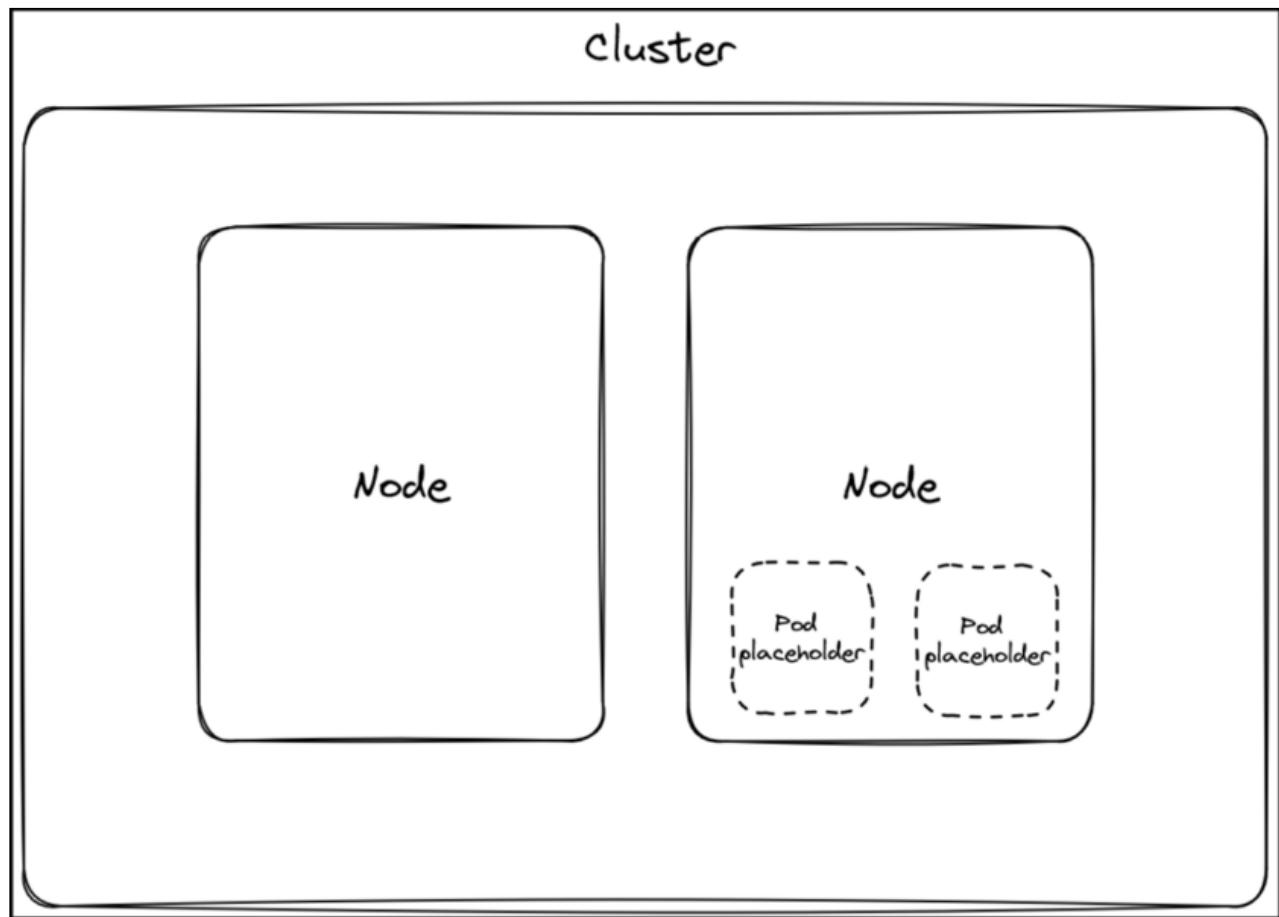


Figura 14.13 Rappresentazione dell'overprovisioning.

Gestione delle risorse

Nella maggior parte dei casi, Kubernetes fa un buon lavoro nel pianificare i Pod sui diversi nodi al posto tuo: prende infatti in considerazione i Pod posizionandoli solo su nodi che dispongono di risorse sufficienti e tenta inoltre di istanziare i Pod che appartengono allo stesso ReplicaSet tra diversi nodi per aumentarne la disponibilità e bilanciare l'utilizzo delle risorse. Quando questo non è sufficiente,

Kubernetes ti offre la flessibilità di influenzare il modo in cui le risorse vengono pianificate. Per esempio, potresti voler pianificare i Pod tra le diverse zone di disponibilità per mitigare la causa di un errore su una certa zona per evitare del tempo di inattività della tua applicazione. Potresti anche voler collocare i Pod in un host specifico per ottenere vantaggi in termini di prestazioni.

Vediamo quindi in questa e nelle seguenti sezioni alcune tecniche per gestire l'allocazione dei Pod nei diversi namespace o nodi, seguendo alcune tecniche.

Affinità

L'affinità (e l'anti-affinità) dei Pod ti consentono di impostare delle regole per posizionare i Pod in relazione ad altri Pod; queste consentono infatti di modificare il comportamento dello scheduler posizionando i Pod su dei nodi specifici. Per esempio, una regola di *anti-affinità* consentirebbe di distribuire i Pod da un ReplicaSet su più zone del data center, utilizzando le labelconfigurate sui Pod.

Zona di disponibilità: che cosa significa?

Una zona di disponibilità consiste in uno o più data center connessi con altre zone in una stessa regione. La maggior parte dei cloud provider (e non solo) ragionano in questi termini: avere dei data center *ridondanti* ci permette di resistere a eventuali guasti, fornendo più repliche della stessa applicazione grazie ad una sua distribuzione su diverse zone. Per esempio, se prendessimo AWS come provider e una regione quella di Milano, sappiamo che sono presenti sul territorio 3 zone di disponibilità (maggiori dettagli qui: https://aws.amazon.com/it/about-aws/global-infrastructure/regions_az/). Nelle seguenti immagini troverai degli schemi che ti saranno sicuramente utili per comprenderne meglio il significato.

Una configurazione che lavori con questa proprietà ti permette quindi di pianificare i Pod sullo stesso nodo (affinità) o impedire la pianificazione dei Pod sugli stessi nodi (anti-affinità). Di seguito è riportato un esempio di impostazione di una regola anti-affinità di un Deployment:

Listato 14.18 Esempio di applicazione di una policy di anti-affinità a un Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-cache
spec:
  selector:
    matchLabels:
      app: store
  replicas: 3
  template:
    metadata:
      labels:
        app: store
  spec:
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
            - key: app
              operator: In
              values:
            - store
              topologyKey: "kubernetes.io/hostname"
        containers:
        - name: redis-server
          image: redis:3.2-alpine
```

Questo esempio ha tre repliche configurate con una label pari a `app:store`. La configurazione di anti-affinità prevede che lo scheduler non collochi le repliche su un singolo nodo (grazie a `.topologyKey`),

basandosi sull'uguaglianza tra Pod che presentano una label con la stessa chiave e lo stesso valore. Ciò garantisce che, in caso di errore di un nodo, ci siano abbastanza repliche di Redis per servire i dati dalla sua cache.

Taint & toleration

Una *taint* consente a un nodo di rifiutare la pianificazione di un Pod a meno che tale Pod non abbia una tolleranza corrispondente.

Ma non è a questo che serve l'anti-affinità? Sì, ma le taintadottano un approccio diverso rispetto all'anti affinità del Pod e servono per dei casi d'uso diversi. Per esempio, potresti avere dei Pod che richiedono delle prestazioni specifiche e non desideri pianificare altri Pod su un nodo con determinate risorse. Le taintlavorano (quasi) sempre insieme alle tolleranze, che ti consentono di sovrascrivere le regole di "blocco" sui nodi. La combinazione delle due ti dà un controllo molto più granulare rispetto alle regole di anti-affinità. In generale, puoi utilizzare le tainte le tolleranze per i seguenti casi d'uso:

- hai uno o più nodi con dell'hardware specializzato (con GPU dedicata, IOPS ottimizzato ecc.);
- nodo dedicato per un singolo progetto/use case.

Puoi applicare le contaminazioni a un nodo tramite la specifica del nodo (`NodeSpec`) e applicare le tolleranze a un Pod (o ai controller) tramite la specifica del Pod (`PodSpec`). Se per esempio volessimo applicare una taint a un nodo, potremmo farlo nel modo che segue.

Listato 14.19 Esempio di taint

```
oc taint nodes [NODE_NAME] [KEY]=[VALUE]:[EFFECT]
# Esempio
oc taint nodes worker1 example-key=example-value:NoSchedule
```

Quando applichi una taint a un nodo, lo scheduler non può istanziare un Pod su quel nodo a meno che il Pod non sia in grado di tollerare la regola specificata.

Ogni nodo, come visto nell'esempio, può avere effetti diversi:

- `NoSchedule`: solo i nuovi Pod che corrispondono alla regola specificata nella taint vengono pianificati su quel nodo;
- `PreferNoSchedule`: istanzia i nuovi Pod solo non possono essere pianificati su altri nodi;
- `NoExecute`: simile a `NoSchedule`, ma la regola vale anche per i Pod già presenti; se questi non rispettano la taint, verranno rimossi.

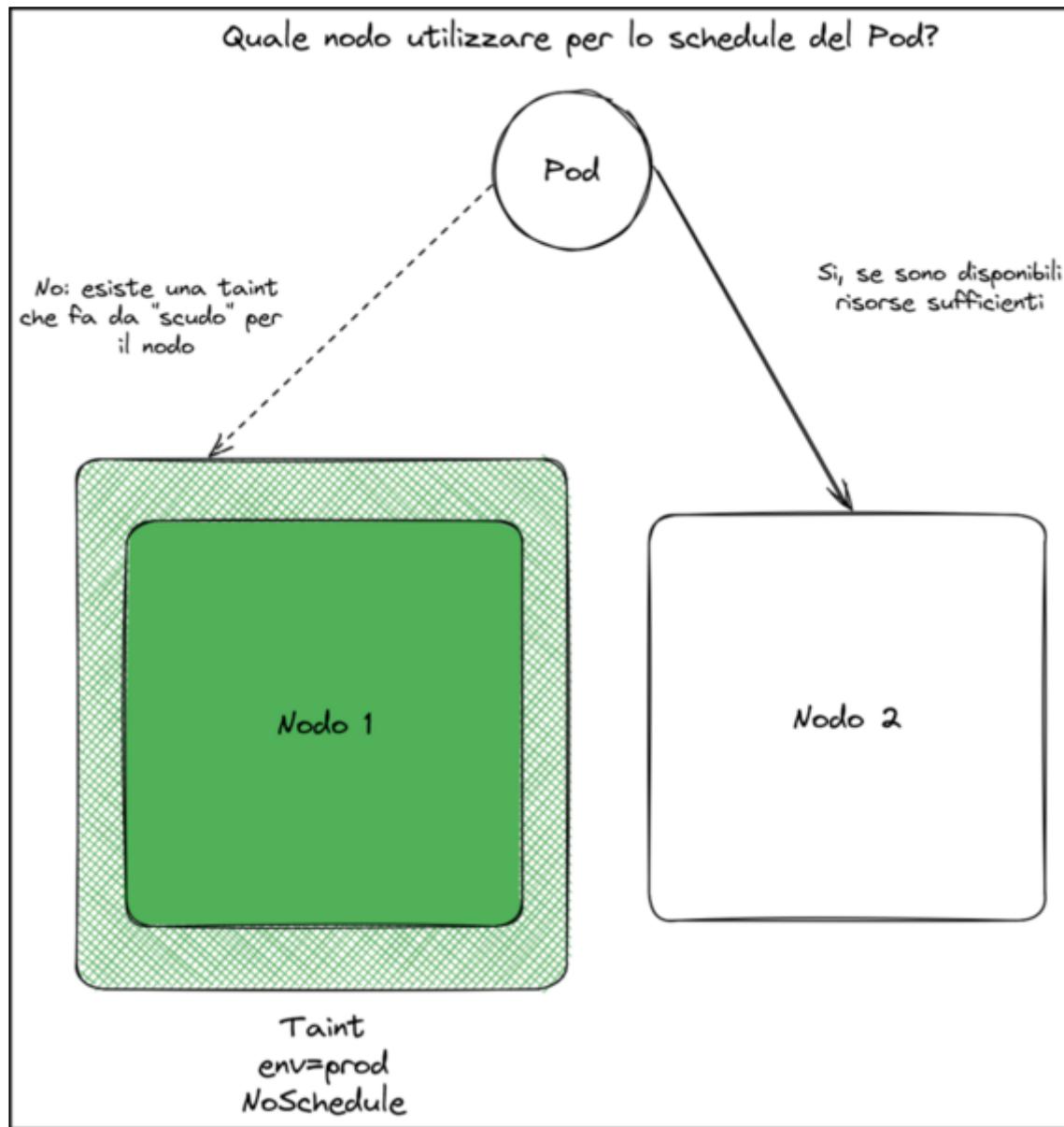


Figura 14.14 Come funziona lo schedule di un Pod con una taint.

Vediamo quindi un esempio completo: annotiamo uno dei nodi con una label specifica che indica che il nodo è dedicato per una demo, e lo faremo applicando una taint: tutti i Pod che fanno parte di questo ecosistema, dovranno essere istanziati su quel nodo specifico, mentre gli altri dovranno essere rimossi, e lo faremo grazie a una toleration.

Listato 14.20 Esempio di taint sul nodo worker

```
oc taint nodes worker1 app=demo:NoExecute
```

Listato 14.21 Esempio di toleration del Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
```

```

spec:
  containers:
    - name: nginx
      image: nginx
      imagePullPolicy: IfNotPresent
  tolerations:
    - key: "app"
      operator: "Exists"
      effect: "NoExecute"
      value: "demo"

```

Namespace

I *namespace* in Kubernetes offrono una buona separazione logica delle risorse distribuite in un cluster. L'importanza dei namespace è spesso sottovalutata: poter isolare progetti diversi, piuttosto che ambienti o stage di sviluppo, è fondamentale per rendere i processi e i carichi di lavoro indipendenti. Questo tipo di separazione consente di impostare, per esempio, delle quote di risorse come CPU o memoria allocata, gestire gli accessi in base al ruolo (applicato delle tecniche di RBAC) per namespacee anche configurazioni di rete custom. Tutto questo consente di ottenere il massimo dal cluster mantenendo al tempo stesso una forma logica di divisione tra le risorse.

Quando si progetta il modo in cui si desidera configurare dei namespace, è necessario pensare a come si desidera controllare l'accesso a un insieme specifico di applicazioni. Se disponi di più team che utilizzeranno un singolo cluster, in genere è meglio allocare un namespacea ciascun team. Se il cluster è dedicato a un solo team, potrebbe avere senso allocare un namespaceper ogni gruppo di prodotti che compongono una soluzione nel cluster, piuttosto che per ogni tipo di fase in cui si trova il progetto: `demo` per un ambiente utilizzato principalmente per le presentazioni di prodotto, `dev` per un ambiente dove eseguire dei test di sviluppo e così via. Non esiste un'unica soluzione a questo; l'organizzazione e le responsabilità del tuo team guideranno il design.

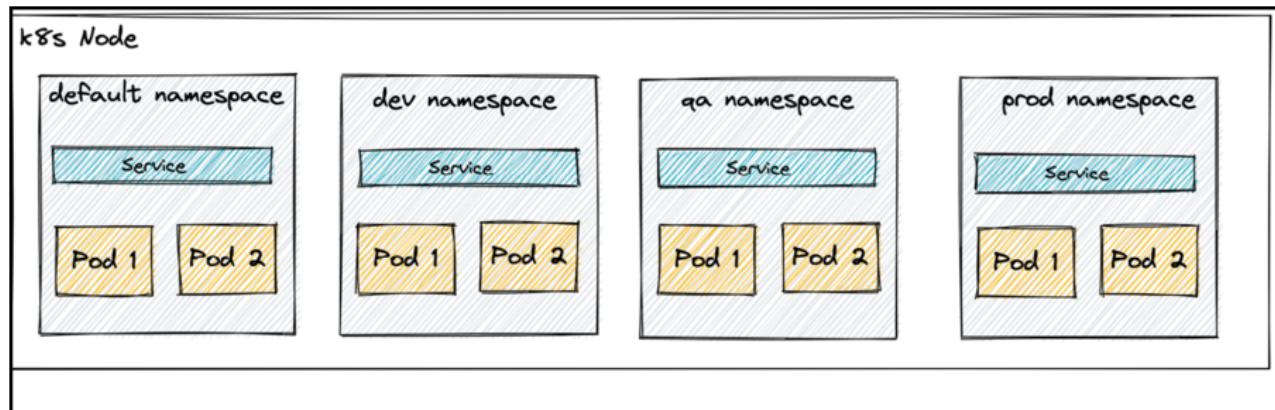


Figura 14.15 Esempio di come organizzare i namespace per stage di prodotto.

Esistono però dei progetti predefiniti quando crei il tuo cluster Kubernetes:

- `kube-system`: i componenti interni di Kubernetes vengono distribuiti qui, come `coredns`, `kubeproxy`, e `metrics-server`;
- `default`: questo è lo spazio utilizzato quando non si specifica un namespace per istanziare degli oggetti;
- `kube-public`: utilizzato per contenuti anonimi e non autenticati e riservato all'utilizzo di sistema.

Il consiglio è di evitare di utilizzare il *namespace* chiamato `default` perché, essendo un ambiente centralizzato, è molto facile commettere errori durante la gestione delle risorse all'interno del cluster ed è piuttosto consigliato crearne di nuovi.

Per esempio, immaginiamo di voler predisporre un cluster per il nostro team di sviluppo, dove le persone avranno bisogno di tre ambienti: uno per le demo con il cliente, uno per eseguire dei test sugli sviluppi e uno dove eseguire dei test di qualità del software. Creiamo tre namespaces diversi, e li chiamiamo rispettivamente `demo`, `dev` e `qa`:

Listato 14.22 Creazione dei namespace

```
kubectl create namespace demo
>>>
namespace/demo created
kubectl create namespace dev
>>>
namespace/dev created
kubectl create namespace qa
>>>
namespace/qa created
```

Quando si lavora con uno specifico namespace, per poter istanziare delle risorse al suo interno, è necessario utilizzare il flag `--namespace` (o `-n` in breve) durante l'esecuzione di un comando:

Listato 14.23 Esecuzione di un comando nel namespace dev

```
kubectl get pods --namespace dev
```

Questo va però ripetuto per ogni comando che riguarda uno specifico namespace. Per evitare questo passaggio, possiamo affidarci al concetto di *contesto*: è possibile infatti specificare l'ambiente che vogliamo utilizzare per tutti i comandi a seguire, sfruttando l'opzione `set-context` nel comando `config`, così che non sia più necessario utilizzare il parametro `--namespace`.

Listato 14.24 Configurazione del contesto

```
kubectl config set-context my-context --namespace=dev
>>>
Context "my-context" created.

kubectl get pods
>>>
No resources found in default namespace.
```

ResourceQuota

A questo punto, potremmo chiederci qual è l'utilità di isolare questi ambienti, oltre al fatto di astrarre delle fasi diverse di un'applicazione durante il suo sviluppo, piuttosto che team diversi. In realtà, durante la configurazione di un cluster, abbiamo allocato delle risorse che magari sono state concepite con l'idea di assegnare a ogni gruppo una fetta del cluster. Per poter perimetrire le risorse allocate a un ambiente, per esempio, di sviluppo, piuttosto che di produzione, potremmo creare delle *quote*: grazie all'utilizzo dell'oggetto `ResourceQuota`, possiamo infatti limitare le risorse computazionali, piuttosto che di spazio di memoria o di oggetti, che ogni namespace può avere a disposizione.

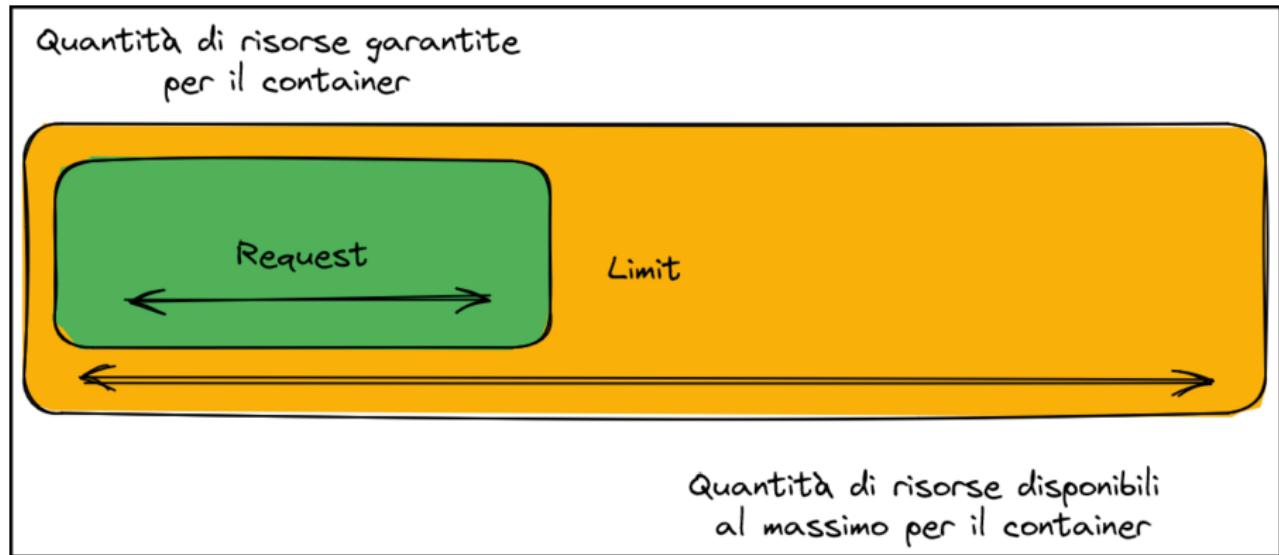


Figura 14.16 Rappresentazione di request e limit all'interno del contesto di una ResourceQuota.

Riprendiamo l'esempio precedente, dove il team ha richiesto la creazione di tre namespace per i diversi stadi del progetto: il progetto di sviluppo ha bisogno come requisito minimo di 500 millicores e 2 GB di memoria e al massimo di un coredi CPU e 4 GB di memoria. Andiamo a creare delle quote *ad hoc* che vadano a perimetrare le risorse che questi progetti possono utilizzare:

Listato 14.25 Configurazione della ResourceQuota per dev

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: dev-quota
  namespace: dev
spec:
hard:
  requests.cpu: «500m»
  limits.cpu: «1000m»
  requests.memory: "2G"
  limits.memory: "4G"
```

Con questo oggetto, abbiamo quindi creato una separazione tra la totalità delle risorse messe a disposizione del cluster e quelle dedicate al progetto di sviluppo. Che cosa succede, quindi, quando andiamo a creare un Pod? Prima della sua creazione, verranno valutate le risorse necessarie affinché il Pod possa essere istanziato: se queste superano la quota allocata al namespace, il Pod non verrà creato e presenterà un errore.

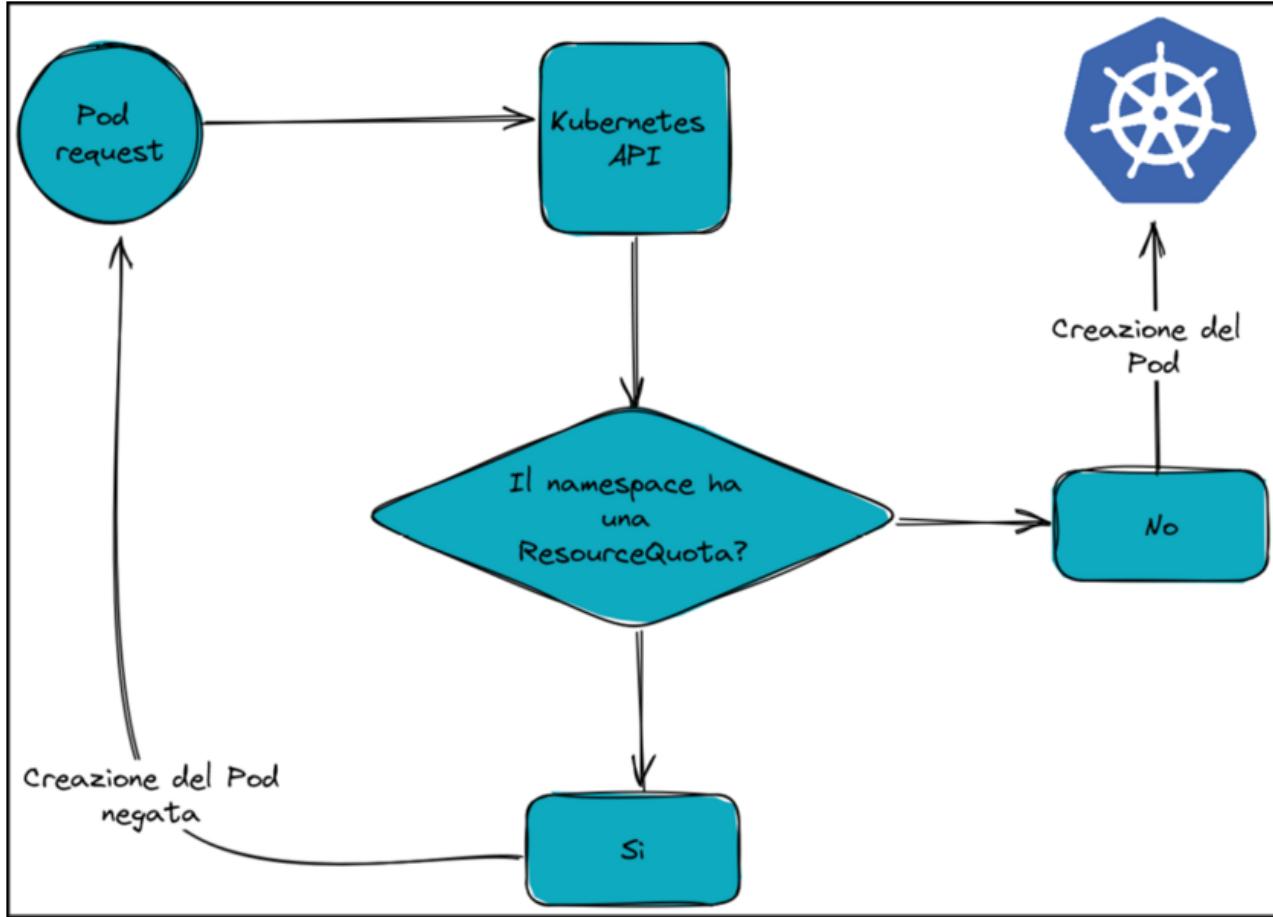


Figura 14.17 Meccanismo di decisione per pianificare un Pod tramite una ResourceQuota.

Listato 14.26 Creazione di un Pod che eccede i limiti del namespace dev

```
kubectl run nginx-test --image=nginx:latest --restart=Never --replicas=1 --port=80 --
requests='cpu=1000m, memory=4Gi' --limits='cpu=2000m, memory=8Gi' -n dev
---
```

Error from server (Forbidden): pods "nginx-test" is forbidden: exceeded quota: dev-quota

In questo caso, abbiamo creato un Pod con delle risorse in termini di `request` e `limit` specifici: cosa succede se il team di sviluppo si dimentica di definire queste informazioni nel template dell'oggetto?

Possiamo pensare di lavorare con i `LimitRange`: Kubernetes fornisce un controller di ammissione che ti consente di impostarli automaticamente quando non sono indicati nelle specifiche della risorsa.

Listato 14.27 Creazione di un LimitRange per la CPU

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-resource-constraint
spec:
  limits:
  - default:
      cpu: 500m
  defaultRequest:
      cpu: 500m
  max:
      cpu: «1»
```

```
min:  
cpu: 100m
```

Attenzione, però: un `LimitRange` non verifica la coerenza dei valori predefiniti che applica: ciò significa che un valore impostato da `LimitRange` può essere inferiore al valore reale della `request` per il container. In tal caso, il Pod finale non riuscirà a partire.

Linux Namespace

I namespace vengono rilasciati come funzionalità del kernel Linux versione 2.6.24 per gestire i processi e renderli isolati e indipendenti, avendo a disposizione una loro “fetta” di sistema. Esistono diverse tipologie di namespace, a seconda del tipo di sezione che vogliamo creare: quelle più comuni isolano i processi sulla base dell’identificativo del processo (alias *PID namespace*), oppure in base all’hostname (*UTS namespace*) o ancora sulla base delle interfacce di rete (*namespace di rete*).

Il meccanismo con cui funziona è il seguente: si creano degli *spazi isolati*, dove risorse e processi comuni risiedono: per esempio, un namespace che si basa sugli ID dei processi andrà a *isolare* le risorse in base a questa informazione. Ciò significa che due processi in esecuzione sullo stesso host potranno avere il medesimo PID.

Questo livello di isolamento è stato fondamentale per il mondo dei container: senza il concetto di namespace, un processo in esecuzione nel container X potrebbe, per esempio, modificare lo stato del filesystem nel container Y, modificare il nome host del container Z o rimuovere un’interfaccia di rete dal container W. Essendo isolato all’interno di un namespace, il processo nel container X non è nemmeno a conoscenza dell’esistenza dei processi nei container Y, Z, e W. Non è possibile “interferire” con qualcosa, se non puoi vederlo: questo è perfetto per limitare l’accesso alle risorse, per garantire una maggiore virtualizzazione e soprattutto una maggiore sicurezza.

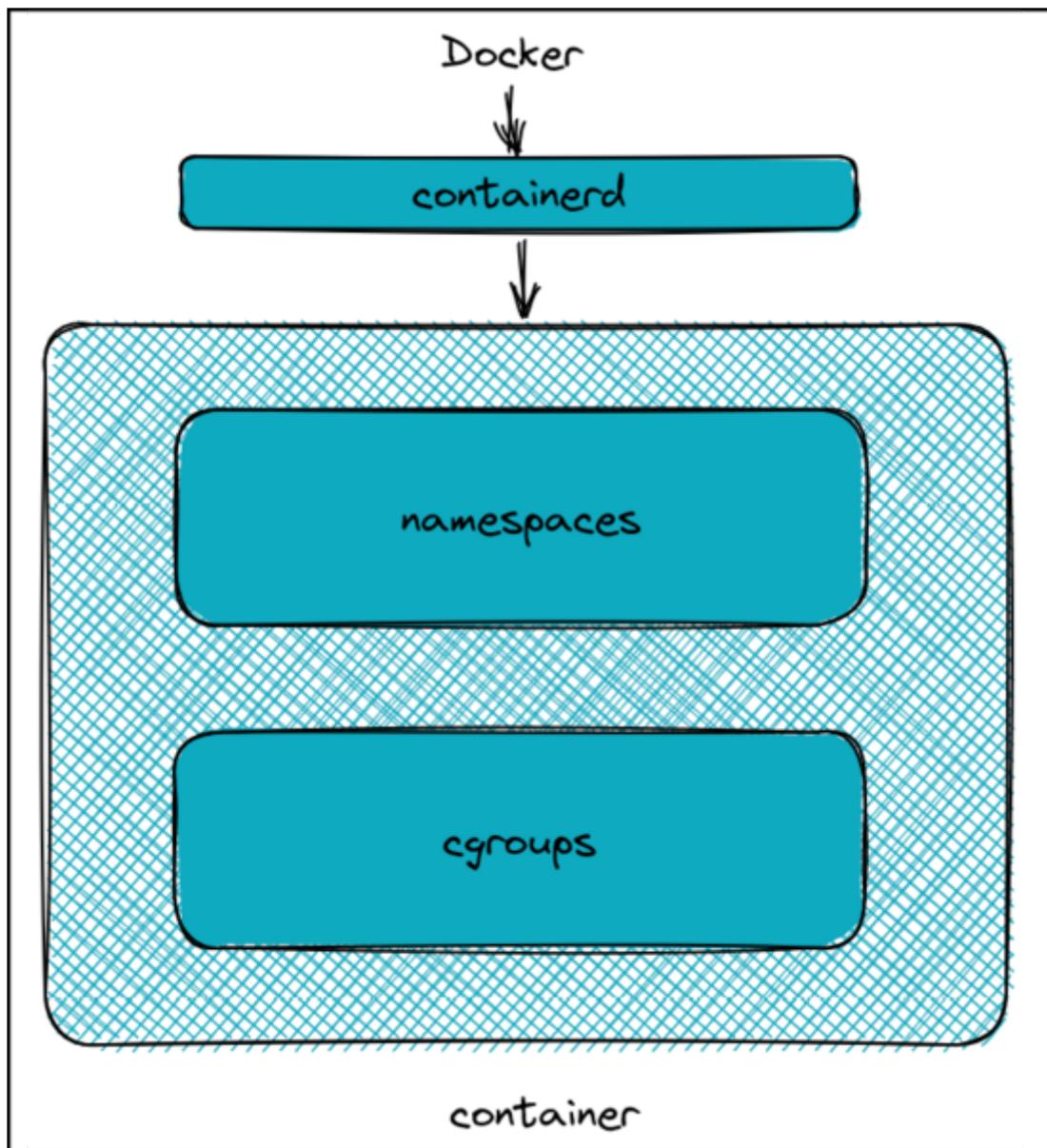


Figura 14.18 Schema di come Docker gestisce namespace e cgroup.

Già con i containerLinux si aveva questa funzionalità (quindi ben prima di Docker), di modo che fosse limitata la visibilità del namespace nel vedere solo la directory da cui erano stati avviati, i propri processi, i propri ID utente e qualsiasi interfaccia di rete a cui è stato consentito l'accesso. Allo stesso modo, i container Linux sono limitati tramite i *cgroup* per controllare l'utilizzo di CPU, memoria, rete e I/O.

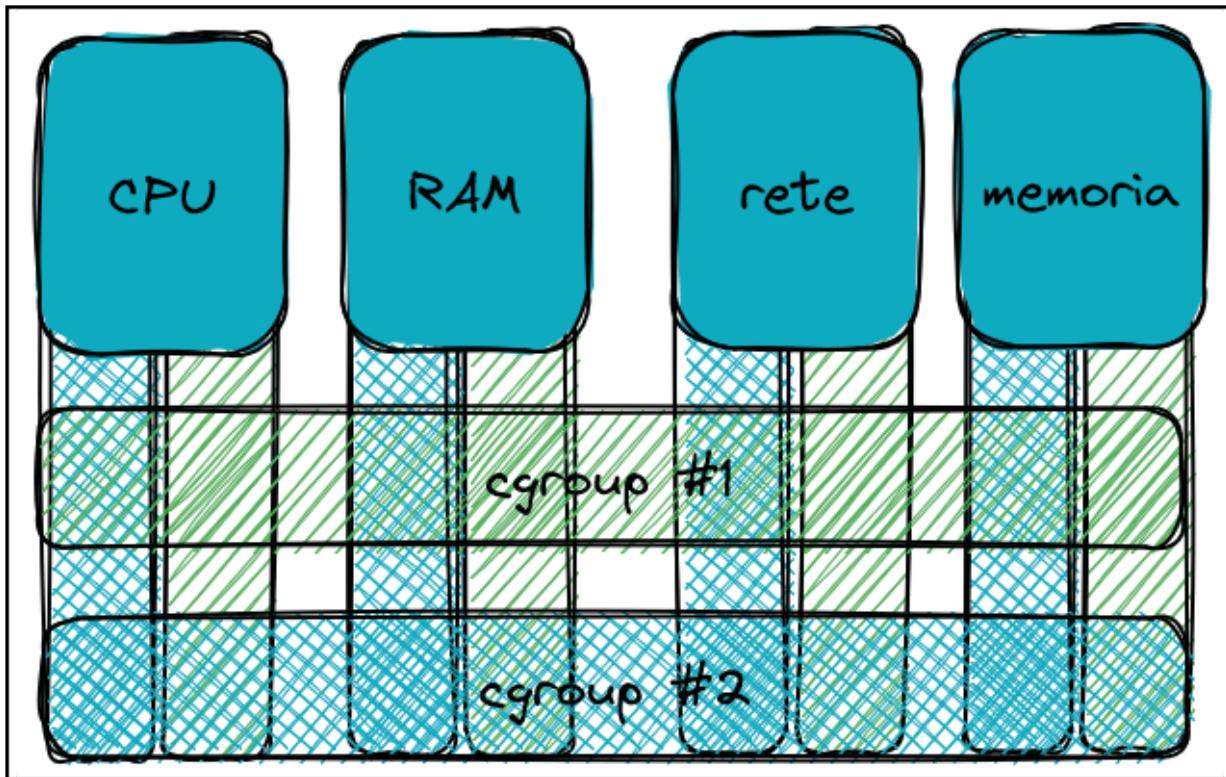


Figura 14.19 Come i cgroup gestiscono le risorse a disposizione.

Un cgroup è una funzionalità del kernel Linux che limita un'applicazione a un insieme specifico di risorse (CPU, memoria, I/O su disco, rete e così via). I cgroup consentono al Docker Engine di condividere le risorse hardware disponibili con i container e, facoltativamente, di impostare limiti e vincoli. Questa funzionalità è perfetta per allocare un set specifico di risorse ad un container: se, per esempio, volessi limitare la memoria disponibile tramite Docker a un container specifico, potresti farlo con il seguente comando.

Listato 14.28 Limitare le risorse utilizzate dal container

```
docker run -it --memory="1g" --cpus="1.0" ubuntu
```

L'ultima versione di cgroup, la versione 2, supporta anche la gestione della "contesa" delle risorse tra più oggetti, chiamata in inglese *Pressure Stall Information*: quando vengono condivisi CPU, memoria o storage con I/O, i carichi di lavoro possono portare a della latenza, perdite di throughput e corrano il rischio di arresto di un processo a causa di un OOM (acronimo di *Out-of-Memory*). Questa funzione identifica e quantifica le interruzioni causate da tali fluttuazioni in termini di risorse e l'impatto temporale che ha su carichi di lavoro complessi o addirittura su interi sistemi.

Listato 14.29 Quale versione di cgroup ho a disposizione?

```
stat -fc %T /sys/fs/cgroup/
>>>
cgroup2fs # cgroup v2
tmpfs # cgroup v1
```

E in Kubernetes? Il motore di K8S gestisce l'utilizzo delle risorse dei nodi del cluster tramite la condivisione delle risorse con un oggetto chiamato CFS (*Completely Fair Scheduler*): questo serve anche a gestire l'over-commitment da un lato (ossia quando la somma di tutti i limiti di risorse è maggiore delle risorse disponibili nel cluster), e dall'altro serve ad allocare su ciascun nodo la quantità totale di request di CPU o memoria di modo che sia inferiore o uguale alla quantità di CPU allocabile dal nodo. In questo scenario, è quindi fondamentale utilizzare request e limit: queste permettono allo scheduler di decidere se, nel caso in cui il nodo sia sovraccarico, deve garantire la sua esecuzione o meno.

Disaster Recovery

Quando parliamo di Disaster Recovery, intendiamo la capacità di ripristinare e riprendere quanto prima l'erogazione dei servizi legati alle applicazioni *business-critical* a seguito di disastri naturali (o umani), proteggendo l'infrastruttura o le applicazioni per ridurre il più possibile l'impatto sul business. Si tratta dunque di una strategia globale di continuità aziendale necessaria per qualsiasi grande organizzazione, progettata per preservare la continuità delle operazioni aziendali durante i principali eventi avversi. L'obiettivo è abilitare il ripristino automatizzato o automatico ed estenderlo a un cluster diverso, mentre il primario è in *manutenzione*. Ridurre il tempo necessario per il ripristino degli incidenti è fondamentale per il successo della tua organizzazione. Spesso, in questo contesto, si parla di backup e ripristino: questi termini si riferiscono a tecnologie e pratiche volte a eseguire copie periodiche di dati e applicazioni su un sistema secondario *separato* e quindi utilizzare tali copie per ripristinare i dati e le applicazioni, e soprattutto i flussi di lavoro aziendali da cui dipendono, nel caso in cui i dati originali e le applicazioni vengono perse o danneggiate a causa di un'interruzione di corrente, un attacco informatico, un errore umano, un disastro o altri eventi imprevisti. Queste tecniche sono una componente essenziale della strategia di ripristino di emergenza di qualsiasi azienda: il costante aggiornamento del backup dei dati è fondamentale per poter definire completo un piano di Disaster Recovery, ma un processo di backup e ripristino da solo non costituisce una soluzione, né un *Disaster Recovery plan*. Esiste infatti un'importante distinzione tra backup e piano di Disaster Recovery: mentre il backup è il processo di creazione di una copia aggiuntiva (o più copie) dei dati, che viene fatta per proteggerli o per ripristinarli se si verifica un'eliminazione accidentale, un danneggiamento del database o un problema con un aggiornamento del software, il DR, d'altra parte, si riferisce a un flusso di lavoro accurato e ai processi per ristabilire rapidamente l'accesso ad applicazioni, dati e risorse IT dopo un'interruzione. Tale piano potrebbe comportare il passaggio a un insieme secondario di server che replica quello primario e i relativi sistemi di archiviazione fino a quando il data center principale non sarà nuovamente funzionante. Questa distinzione è importante, perché spesso si tende a confondere il backup con il ripristino dopo un'emergenza, il quale ha delle regole da seguire, così come delle conseguenze; come è facile immaginare, si potrebbe scoprire solo dopo una grave interruzione che il semplice fatto di avere copie dei dati non significa poter mantenere il sistema in funzione. Per garantire la continuità aziendale, è necessario un piano di ripristino di emergenza solido e, soprattutto, testato.

In questo contesto, comprendere alcuni termini essenziali può aiutarti a prendere decisioni strategiche e consentirti di valutare meglio le soluzioni di backup e di applicazione di un DR, e cominciamo da RTO: *Recovery Time Objective*, o obiettivo del tempo di ripristino, è la quantità di tempo necessaria per ripristinare le normali operazioni aziendali dopo un'interruzione. Questo rappresenta quindi quanto tempo siamo disposti a perdere e l'impatto che il tempo avrà sui profitti un eventuale disservizio del sistema, motivo per cui questa metrica può variare notevolmente da un tipo di attività all'altro. Per esempio, se una biblioteca pubblica perde il proprio sistema di catalogazione, è probabile che possa continuare a operare manualmente per alcuni giorni mentre i sistemi vengono ripristinati; se un e-commerce molto noto perdesse invece il proprio sistema di inventario, anche 10 minuti di inattività porterebbero alla perdita di entrate di portata inaccettabile. Il *Recovery Point Objective* (abbreviato in RPO) si riferisce alla quantità di dati che puoi permetterti di perdere in caso di emergenza. Questo ha un valore molto diverso, perché potrebbe essere necessario copiare continuamente i dati in un data center remoto in modo che un'interruzione non comporti alcuna perdita di dati, oppure potresti decidere che perdere due ore di lavoro è comunque accettabile. Il *failover* è invece il processo di ripristino di emergenza che sposta automaticamente le attività sui sistemi di backup in modo trasparente per gli utenti: in altre parole, si parla di failover quando si passa il carico di lavoro dal data center principale a un sito secondario, dove i sistemi sono replicati per essere pronti a subentrare immediatamente. Al contrario, il *fallback* è il processo di ripristino che permette di tornare al sistema primario originale: una volta che l'incidente è passato e il data center principale è di nuovo

attivo e funzionante, dovremmo essere in grado di eseguire anche il failback senza interruzioni. Infine, con il termine *ripristino* si intende il processo di trasferimento dei dati di backup al sistema principale o al data center; questo è generalmente considerato parte del backup piuttosto che del DR, perché avviene solo grazie alla creazione e gestione dei backup.

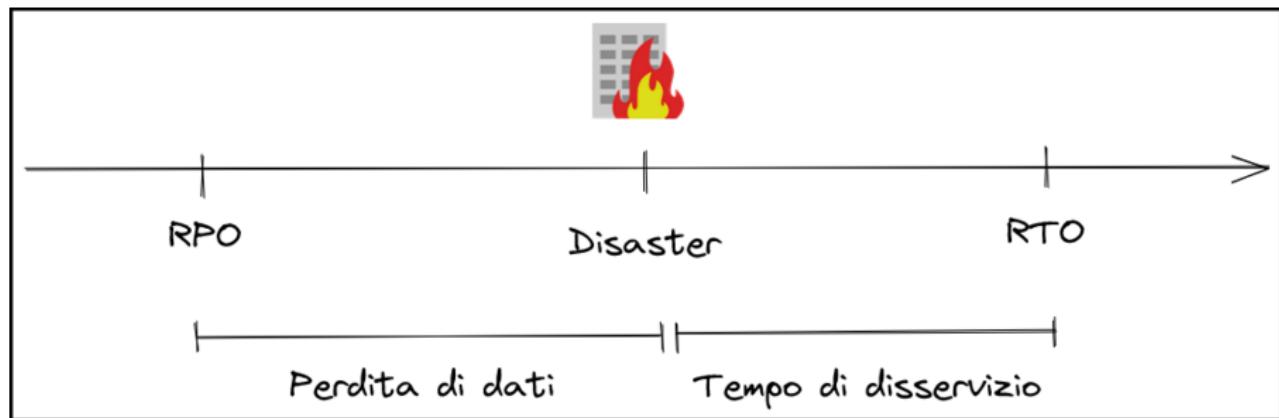


Figura 14.20 Rappresentazione di RPO e RTO in un piano di Disaster Recovery.

Ora che abbiamo preso un po' di familiarità con alcuni termini, dedichiamo un po' di attenzione a queste considerazioni: c'è una crescente pressione sulle applicazioni che vengono distribuite in Kubernetes e, soprattutto in un contesto di emergenza, come può essere il ripristino del sistema dopo che c'è stato un problema, i backup possono essere visti come una forma rudimentale di replicazione dei volumi in maniera asincrona. Una premessa è quindi doverosa: i backup sono uno strumento fondamentale per avere una forma di sicurezza aggiuntiva, ma comunque è necessario fare delle riflessioni se questa tecnica viene utilizzata come unica strategia per reagire alle conseguenze di un incidente: i backup degli ambienti di produzione vengono eseguiti molto raramente (o mai) e il rischio principale è quello di non riuscire a recuperare i dati nel punto esatto di quando ne hanno effettivamente bisogno. Di base, non esiste alcun livello di astrazione in Kubernetes per eseguire o pianificare un backup o un ripristino che valga per il cluster, o anche per i nodi *control plane*, ma è necessario utilizzare tecnologie (come Velero o Fossul) o procedere con delle strategie *ad hoc*. Per questa ragione, illustriamo in questa sezione alcune tecniche che possono tornare utili per eseguire un backup & restore che permetta alle nostre infrastrutture di resistere anche ai colpi più duri.

Backup di etcd

Nell'architettura Kubernetes, etcd è il cuore pulsante del cluster: tutti gli oggetti del cluster e il loro stato sono memorizzati in etcd, il quale funziona grazie a un sistema basato sul formato chiave-valore volto alla coerenza e alla sicurezza dei dati. Quello che non tutti sanno è che etcd ha un meccanismo di snapshot integrato: si chiama `etcdctl` ed è uno strumento che è possibile usare tramite riga di comando che interagisce con etcd per eseguire degli snapshot. Per farlo in maniera programmatica, puoi seguire questi passaggi: intanto, è necessario accedere a uno dei nodi *control plane* e verificare che `etcdctl` sia installato. Se non lo è, è possibile installarlo tramite il proprio package manager, come mostrato di seguito:

Listato 14.30 Installazione di etcdctl

```
sudo apt install etcd-client # per Ubuntu-like
# oppure
ETCD_RELEASE=$(curl -s https://api.github.com/repos/etcd-io/etcd/releases/latest | grep
tag_name | cut -d '"' -f 4)
```

```

ARCH=amd64 # o sostituire con quella attuale

wget https://github.com/etcd-
io/etcd/releases/download/${ETCD_RELEASE}/etcd-${ETCD_RELEASE}-linux-${ARCH}.tar.gz
cd etcd-${ETCD_RELEASE}-linux-amd64
sudo mv etcd* /usr/local/bin

```

Una volta installato, potremo procedere con la creazione di uno snapshot: per farlo, dovremo passare delle informazioni a `etcdctl`, tra cui l'endpoint con cui raggiungere le API di etcd e i certificati della CA (alias *Certificate Authority*), del server e la chiave, il tutto per configurare una comunicazione sicura. Tutte queste informazioni possono essere trovate all'interno del file YAML che descrive il Pod di etcd, oppure utilizzando i comandi `kubectl get Pods -n kube-system` e `kubectl describe Pod etcd-master-node -n kube-system`:

Listato 14.31 Esempio di definizione del Pod di etcd

```

apiVersion: V1
kind: Pod
metadata :
  annotations:
    kubeadm.kubernetes.io/etcd.advertise-client-urls: xxx
  creationTimestamp: null
  labels :
    component: etcd
    tier: control-plane
  name: etcd
  namespace: kube-system
spec:
  containers:
  - command:
    - etcd
    - --advertise-client-urls=https://10.0.0.10:2379
    - --cert-file=/etc/kubernetes/pki/etcd/server.crt
    - --client-cert-auth=true
    - --data-dir=/var/lib/etcd
    - --experimental-initial-corrupt-check=true
    - --initial-advertise-peer-urls=https://10.0.0.1
    - --initial-cluster=master-node=https://10.0.0.1
    - --key-file=/etc/kubernetes/pki/etcd/server.key
    ...
    - --trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
...

```

Lo snapshot può essere eseguito con il seguente comando: presi come parametri quelli evidenziati nell'output precedente, andiamo a sostituire le informazioni relative all'endpoint del Pod, così come i certificati, e poi specifichiamo dove dev'essere salvato il backup di etcd.

Listato 14.32 Esecuzione dello snapshot

```

ETCDCTL_API=3 etcdctl \
--endpoints=https://127.0.0.1:2379 \
--cacert=/etc/kubernetes/pki/etcd/ca.crt \
--cert=/etc/kubernetes/pki/etcd/server.crt \
--key=/etc/kubernetes/pki/etcd/server.key \
snapshot save /opt/backup/etcd.db
>>>
Snapshot saved at /opt/backup/etcd.db

```

Il messaggio di output è chiaro, e lo snapshot è stato eseguito con successo. Tuttavia, possiamo utilizzare `etcdctl` per riportare le informazioni salienti del backup sotto forma tabellare tramite un comando come il seguente, che ne definisce la dimensione, il numero di coppie chiave-valore presenti e anche un numero di revisione:

Listato 14.33 Descrizione dello snapshot effettuato

```
ETCDCTL_API=3 etcdctl --write-out=table snapshot status /opt/backup/etcd.db
>>>
+-----+-----+-----+-----+
| HASH | REVISION | TOTAL KEYS | TOTAL SIZE |
+-----+-----+-----+-----+
| c7137893| 12345 | 2023 | 5.7 MB |
+-----+-----+-----+-----+
```

ETCDCTL_API=3: che cosa vuol dire?

La versione dell'API utilizzata da `etcdctl` per comunicare con `etcd` può essere impostata sulla versione 2 o 3, per cui viene specificata tramite la variabile d'ambiente `ETCDCTL_API`. Per impostazione predefinita, `etcdctl` su master (3.4) utilizza l'API v3 e le versioni precedenti (3.3 e precedenti) utilizzano per impostazione predefinita l'API v2.

Ora che abbiamo eseguito uno snapshot, vediamo anche come fare il restore: il comando è molto simile a quanto visto per il backup, solo che stavolta indichiamo la directory dove recuperare il file che contiene i dati di `etcd` per procedere con il ripristino:

Listato 14.34 Esecuzione del restore di `etcd`

```
ETCDCTL_API=3 etcdctl snapshot restore /opt/backup/etcd.db
# oppure
export ETCDCTL_API=3
etcdctl snapshot restore /opt/backup/etcd.db
```

Abbiamo visto come eseguire il backup e restore, ma non quali strategieabbiamo a disposizione: se lavoriamo con un cluster non gestito (da un cloud provider, per esempio), eseguire il backup internamente all'infrastruttura del cluster o su un server esterno può avere diverse conseguenze. Se per esempio volessimo utilizzare una risorsa che si occupi di eseguire il backup di `etcd` frequentemente, potremmo pensare di delegare questo compito a un CronJob: questo tipo di oggetto è perfetto per ripetere l'attività frequentemente, di modo che in caso di eventi avversi (come la corruzione dei dati di `etcd`), sia possibile eseguire il restore. Per eseguire un backup dall'interno tramite `etcd`, possiamo utilizzare quindi un CronJob che non richiede l'installazione di alcun client `etcdctl` sull'host, ma è sufficiente sfruttare l'immagine di `etcd` riportare il comando visto in precedenza che sfrutta `etcdctl` per eseguire lo snapshot. Di seguito è riportata una definizione di esempio di un CronJob che eseguirà il backup `etcd` ogni minuto su una directory del nodo:

Listato 14.35 Definizione del CronJob di backup di `etcd`

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: backup
  namespace: kube-system
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: backup
              # Use the same image as specified in the etcd pod
              image: k8s.gcr.io/etcd:3.2.24
              env:
                - name: ETCDCTL_API
                  value: "3"
                command: ["/bin/sh"]
              args: ["-c", "etcdctl --endpoints=https://127.0.0.1:2379 --"]
```

```

cacert=/etc/kubernetes/pki/etcd/ca.crt
cert=/etc/kubernetes/pki/etcd/server.crt --key=/etc/kubernetes/pki/etcd/server.key snapshot
save /backup/etcd-snapshot-$(date +%Y-%m-%d_%H:%M:%S_%Z).db"]
    volumeMounts:
        - mountPath: /etc/kubernetes/pki/etcd
            name: etcd-certs
            readOnly: true
        - mountPath: /backup
            name: backup
            restartPolicy: OnFailure
hostNetwork: true
volumes:
    - name: etcd-certs
        hostPath:
            path: /etc/kubernetes/pki/etcd
            type: DirectoryOrCreate
    - name: backup
        hostPath:
            path: /data/backup
            type: DirectoryOrCreate

```

Lo stesso tipo di lavoro può essere eseguito da un server esterno, configurando con la stessa logica un cron Linux per eseguire il backup del cluster in maniera periodica.

Backup del cluster

Probabilmente, solo per questa sezione, ci vorrebbe un libro a parte: quando parliamo di backup del cluster, è possibile utilizzare tantissime tecniche e tecnologie, che non è possibile sintetizzare in queste poche righe. Molte delle soluzioni sul mercato sono peraltro proprietarie e disponibili solo dietro licenza a pagamento, ma proviamo comunque a fornirne una che possa dare l'idea di come eseguire il backup di un cluster (o parte di esso) tramite un progetto open source, ossia Velero.

Velero (precedentemente chiamato Heptio Ark) è un progetto che mette a disposizione diversi strumenti per eseguire il backup e il ripristino delle risorse di un cluster Kubernetes e dei suoi volumi persistenti. Può essere eseguito sia su infrastrutture cloud che on-premise, e consente l'esecuzione di backup del cluster e ripristino in caso di perdita, ma anche la migrazione delle sue risorse su altri cluster per, per esempio, replicare il tuo cluster di produzione su un'altro di sviluppo e test. Velero è costituito da due componenti principali: un server che viene eseguito sul cluster e un client della riga di comando che viene eseguito localmente per gestire Velero. Ogni operazione Velero (backup su richiesta, pianificato o ripristino) è una risorsa personalizzata, definita tramite una *Custom Resource Definition* (abbreviata in CRD) e memorizzata in etcd. Velero include anche controller che elaborano le richieste di backup, ripristini e tutte le operazioni correlate. Il backup è possibile a partire da tutti gli oggetti nel cluster oppure filtrando gli oggetti per tipo, namespace o label.

Vediamo qualche semplice esempio: immaginiamo di avere un namespace chiamato `dev` e di volerne eseguire il backup, non tenendo conto (per il momento) dei volumi persistenti: per eseguire il backup e il ripristino, è sufficiente installare Velero per usarlo da riga di comando e configurare i componenti server all'interno del cluster, tramite Helm o riga di comando. Il funzionamento di Velero è infatti molto semplice: Ogni volta che eseguiamo il comando di backup tramite riga di comando, Velero effettua una chiamata al server API Kubernetes per creare un oggetto di backup. Il controller di backup convalida quindi tale oggetto e verifica a quali risorse fare riferimento, se ci sono oggetti da escludere o namespace interi da non salvare. Infine, avvia il processo di backup: tramite il BackupController effettua quindi una chiamata allo storage predisposto per salvare il file di backup. Allo stesso modo, ogni volta che eseguiamo un comando di ripristino, Velero CLI effettua una chiamata al server API

Kubernetes per ripristinare le risorse a partire da un oggetto di backup. In base al comando di ripristino eseguito, il controller di Velero effettua una chiamata allo storage e avvia il ripristino (Figura 14.21).

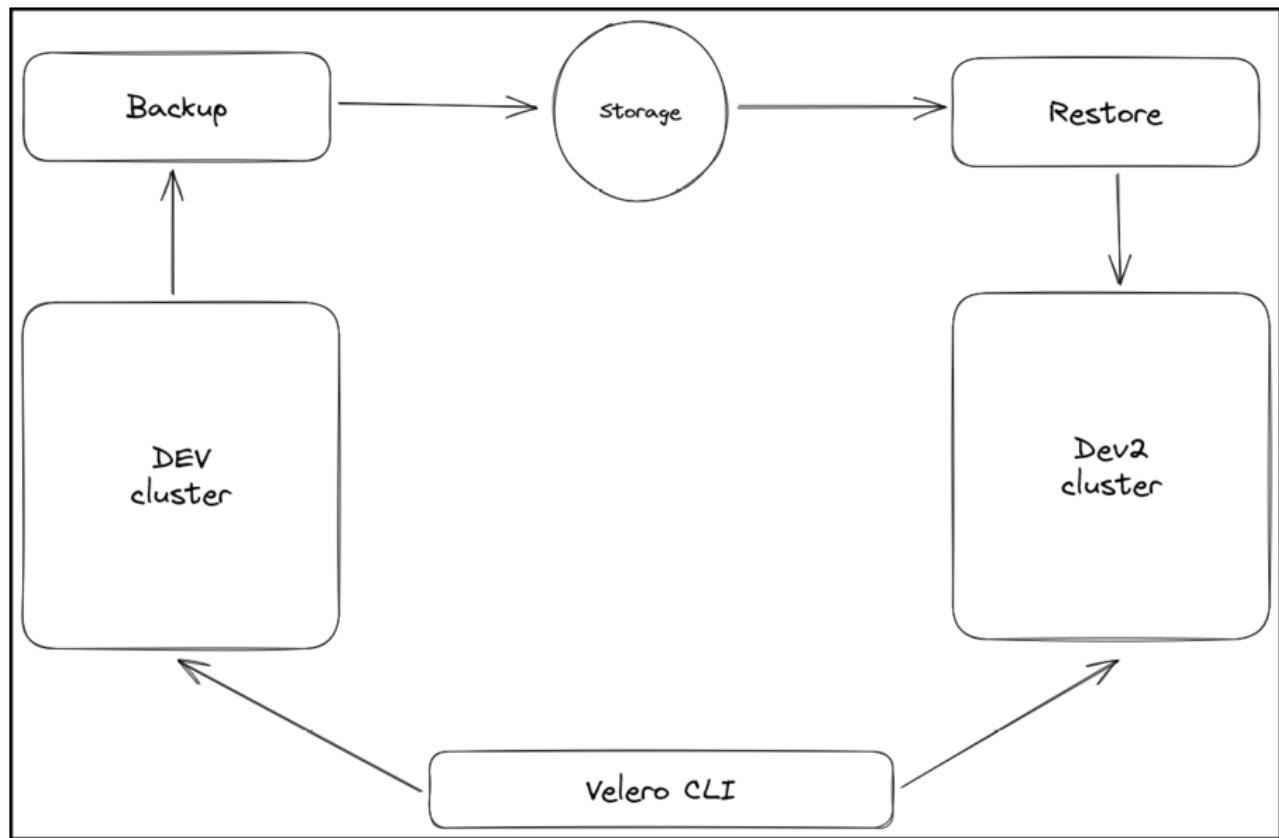


Figura 14.21 Schema generico del funzionamento di Velero per il backup e restore.

Quando Velero sarà utilizzabile da terminale e i controller di Velero saranno attivi, potremo procedere come segue: tramite il comando `velero backup`, specifichiamo la creazione della copia del namespace `dev` e lo chiamiamo `dev-backup`:

Listato 14.36 Backup di un namespace

```
velero backup create dev-backup --include-namespaces dev
```

Simulando un evento avverso, andiamo a distruggere il namespace `dev` quando il backup sarà completato ed eseguiamo il restore tramite il comando `velero restore`.

Listato 14.37 Restore di un namespace

```
velero restore create --from-backup dev-backup
```

Abbiamo anche detto che Velero consente la programmazione dei backup, di modo che vengano eseguiti in maniera ricorrente: è possibile utilizzare il comando `velero create schedule`, che funziona in maniera molto simile a quanto visto per il backup, tranne per il fatto che è necessario specificare l'orario o il giorno in cui eseguirlo:

Listato 14.38 Schedule del backup di un namespace

```
velero create schedule sch-backup-dev --schedule="15 10 * * *" --include-namespaces dev
```

Listato 14.39 Elenco schedule

```

velero get schedule
>>>
NAME          STATUS  CREATED           SCHEDULE      BACKUP TTL   LAST
BACKUP     SELECTOR
sch-backup-dev Enabled ...             5 10 * * *
<none>

```

Allo stesso modo, tramite Velero, è possibile ottenere la lista dei backup presenti e gestirne la persistenza con i comandi `velero get` e `velero delete`:

Listato 14.40 Elenco dei backup

```

velero get backup
>>>
NAME          STATUS           ERRORS
WARNINGs    CREATED
sch-backup-dev-20220905101515 Completed        0

```

Listato 14.41 Rimozione di un backup

```

velero delete backup sch-backup-dev-20220905101515
>>>
Are you sure you want to continue (Y/N)? y
Request to delete backup "backup sch-backup-dev-20220905101515" submitted successfully.
The backup will be fully deleted after all associated data (disk snapshots, backup files,
restores) are removed.

```

Quanto visto finora non vale solo per uno specifico namespace: Velero permette infatti di effettuare il backup dell'intero cluster, o anche di programmarlo, tramite gli stessi comandi; basta non specificare il namespace da includere, e la copia sarà completa di tutti i namespace presenti.

Listato 14.42 Schedule di un backup

```
velero schedule create <SCHEUDLE NAME> --schedule "0 8 * * *"
```

Questo crea un oggetto di backup con il nome <SCHEUDLE NAME>-<TIMESTAMP>. Il periodo di conservazione del backup, espresso tramite il campo `TTL`(alias *time to live*), è di 30 giorni, ma è possibile specificarne uno differente usando `--ttl`: in questo modo, se Velero si accorge che esiste un backup “scaduto”, ne rimuove tutte le risorse.

Listato 14.43 Schedule di un backup con scadenza due giorni

```
velero schedule create sch-bck-dev --schedule "0 8 * * *" --ttl 48h0m0s
```

Sostenibilità

Verrà naturale chiedersi che cosa c’entri Kubernetes con la sostenibilità: la realtà è che questa parola va ben oltre l’aspetto ambientale e riguarda moltissimi altri aspetti quotidiani, come quello economico: riservate delle risorse cloud senza utilizzarle realmente è uno spreco, soprattutto dal punto di vista economico. Pensare di tenere sotto controllo tutti i Pod o i controller inutilizzati è altrettanto impegnativo, motivo per cui nel 2022 la tech company italiana Mia Platform ha rilasciato un prodotto open source chiamato *kube-green* (repository ufficiale: <https://github.com/kube-green>), che mira a ottimizzare il consumo energetico delle infrastrutture server IT, riducendo il numero di Pod inattivi nei cluster, e di conseguenza ridurre le emissioni di CO₂ necessarie a mantenerli accesi.

Facciamo qualche calcolo: immagina di avere un cluster di sviluppo che permette di mettere a punto delle applicazioni e di fare dei test. Questo ambiente sarà utilizzato a regime durante le ore lavorative settimanali, che ammontano circa a 40, e non è detto che lo sia per la totalità di queste ore. Quasi sicuramente, di notte e il weekend questo ambiente non sarà utilizzato, e quindi l’energia necessaria a

mantenerlo andrà sprecata: tutte le infrastrutture che abbiamo a disposizione per i nostri cluster, andranno a consumare energia elettrica, e quindi emetteranno più CO₂.

Qual è il punto? Le risorse costano: diverse volte le aziende si sono pubblicamente lamentate dei costi delle infrastrutture per la gestione e la manutenzione, incollando il cloud di costi quasi superiori a quelli dei server “tradizionali”: la realtà è che anche le risorse in cloud hanno un costo, e una cattiva gestione può far aumentare notevolmente i costi. Riprendendo l'esempio di prima, solo nel 2022 ci sono stati 53 sabati e 53 domeniche; questo ammonta a 106 giorni di inattività, quindi quasi un terzo dell'anno lavorativo, escludendo le feste canoniche. Basti pensare che un server con 8 core di CPU e 16 GB di RAM consuma, in un solo giorno, l'equivalente di un auto che percorre 3 km: moltiplicato per quei famosi 106 giorni, parliamo di un viaggio Roma-Firenze pagato, o anche di un aereo Parigi-Londra di sola andata (fonte: <http://calculator.green-algorithms.org/>).

Senza perderci in altri conti, parliamo di kube-green: questo strumento nasce da un'idea di Davide Bianchi, Senior Tech Leader, come progetto open source e che funziona come un semplice plugin che andrà a spegnere alcune delle risorse del cluster, come Deployment e CronJob, quando non sono utilizzate. Per installarlo, è sufficiente disporre di un cluster Kubernetes con una versione compresa tra 1.19 e 1.24, o anche di OpenShift versione 4, e di disporre di un *cert-manager*. Possiamo quindi installarlo utilizzando kustomize, o anche un file YAML tramite il comando `kubectl apply`, come mostrato di seguito.

Listato 14.44 Comando kubectl apply

```
kubectl apply -f https://github.com/kube-green/kube-green/releases/latest/download/kube-green.yaml
```

Una volta installato, saremo pronti per configurare il “riposo” dei Pod tramite un oggetto chiamato `SleepInfo`: questa risorsa custom permette di definire non solo i giorni in cui vogliamo attivare kube-green, ma anche le fasce orarie o eventuali oggetti che devono essere esclusi da queste regole, come magari dei servizi fondamentali per l'attività di raccolta dati del cluster.

cert-manager: cos'è?

`cert-manager` è uno strumento per la gestione dei certificati nativo di Kubernetes. Può aiutare con l'emissione di certificati tramite diverse fonti, come *Let's Encrypt*, *HashiCorp Vault* e *Venafi*, ma anche supportare l'utilizzo di una semplice coppia di chiavi di firma o auto firmata. Si occupa di garantire che i certificati siano validi e aggiornati e di rinnovare i certificati poco prima della scadenza.

Un esempio di configurazione di `SleepInfo` è il seguente: immagina di voler disattivare le risorse dalle 19.00 alle 7.00 del mattino durante i giorni lavorativi. Puoi utilizzare un file come il seguente:

Listato 14.45 Configurazione SleepInfo

```
apiVersion: kube-green.com/v1alpha1
kind: SleepInfo
metadata:
  name: esempio
spec:
  weekdays: "1-5"
  sleepAt: "19:00"
  wakeUpAt: "07:00"
  timeZone: "Europe/Rome"
```

La risorsa `SleepInfo` può contenere le seguenti informazioni: `weekdays` sono i giorni della settimana espressi numericamente, dove * rappresenta tutti i giorni, 1 corrisponde a lunedì, e 1-5 è pari a un intervallo dal lunedì al venerdì. `sleepAt` rappresenta l'orario espresso in ore e minuti (HH:mm) in cui il namespace “andrà a dormire”, questo vuol dire che le risorse verranno spente e non saranno

nuovamente accese fintanto che non avverrà il deploy dell'applicazione. I valori validi sono, per esempio, `19:00` o `*:*` per impostare ogni minuto e ogni ora. Questi sono i due parametri obbligatori, ma esistono anche diversi parametri opzionali, come quelli mostrati nell'esempio: `wakeUpAt`, per specificare l'orario in cui il namespace dovrebbe essere ripristinato allo stato iniziale, o anche `timeZone` per definire il fuso orario nella specifica IANA. Come accennato prima, esiste anche la possibilità di escludere delle risorse dalle azioni di `kube-green`, tramite `excludeRef`: questo permette di descrivere tramite un array di oggetti le risorse da escludere dalla sospensione, tramite la tipologia e il nome.

Listato 14.46 Configurazione SleepInfo con esclusione del Deployment

```
apiVersion: kube-green.com/v1alpha1
kind: SleepInfo
metadata:
  name: working-hours
spec:
  weekdays: "1-5"
  sleepAt: "19:00"
  wakeUpAt: "07:00"
  timeZone: "Europe/Rome"
  suspendCronJobs: true
  excludeRef:
    - apiVersion: "apps/v1"
      kind: Deployment
      name: my-deployment
```

Il risultato ottenuto utilizzando `kube-green` in un piccolo cluster di sviluppo dove ci sono 5 namespace attivi per lo sviluppo è che si risparmiano circa 30 Kg di CO₂ a settimana, quindi poco più di 1500 kg di CO₂, ossia l'equivalente di 24 voli Parigi-Londra!

Che cosa abbiamo imparato

- Un ambiente di produzione differisce sotto diversi punti di vista da uno locale, e ci sono alcune considerazioni da fare: queste riguardano sia alcune risorse finora non trattate, che la gestione complessiva del cluster.
- Gli Autoscaler sono una risorsa preziosa per gestire la scalabilità del cluster e degli oggetti al suo interno.
- Gestire i namespace del cluster per isolare le applicazioni è una buona pratica sia in termini di sicurezza, che di gestione delle risorse.
- A proposito di gestione delle risorse, tanto e toleration insieme alle quote riservate a un namespace piuttosto che alle singole applicazioni, ci permettono di preservare l'integrità del cluster e tenere sotto controllo i consumi.
- E, a proposito di consumi, per essere più sostenibili, abbiamo parlato di `kube-green`, un progetto open source che permette di ridurre la quantità di risorse utilizzate all'interno del cluster negli orari in cui questo non è attivamente utilizzato.

Kubernetes on cloud

È giunto il momento di incoraggiare più donne a rendere ogni sogno possibile.
– Sheryl Sandberg, COO di Facebook

All'inizio del nostro viaggio abbiamo parlato di come Kubernetes sia permeato nel mondo della tecnologia su tanti fronti: della nascita del progetto nei laboratori di Google e della sua adozione nelle piccole, medie e grandi aziende. Ecco, è da qui che ripartiamo, per concludere questa esperienza con l'orchestrazione: analizzando come il contesto enterprise abbia scelto di sfruttare Kubernetes in diverse forme per renderlo accessibile agli utenti. Questo capitolo è quindi da intendersi come una panoramica più che ampia sulle possibilità che abbiamo, a oggi, per poterci lavorare, quali sfide incontreremo, quali considerazioni fare quando lavoriamo con un ambiente piuttosto che un altro. Daremo un'occhiata alle soluzioni offerte dal mercato dei servizi delle *Big Companies*, per poi tornare in Italia, con un'azienda che ha elevato le potenzialità di questa tecnologia all'ennesima potenza. Si comincia!

Piccolo disclaimer

Quanto affronteremo in questo capitolo è un accenno alle risorse che mettono a disposizione una serie di piattaforme per utilizzare Kubernetes; per ognuna di esse, sarebbe necessario scrivere un manuale separato. In ogni caso, quelli che vedremo saranno esempi pratici per iniziare a muovere i primi passi con queste tecnologie, tenendo in considerazione che ognuna di queste ha dei costi.

Managed vs self-hosted

Nel panorama delle soluzioni che offrono Kubernetes, occorre fare una piccola distinzione: molti di questi provider offrono delle soluzioni *managed*, ossia completamente (o parzialmente) gestite, dove la parte in carico al cliente riguarda perlopiù il lato applicativo, in contrapposizione alle opzioni *self-hosted*, ossia soluzioni dove sia l'infrastruttura e la sua configurazione, sia la relativa manutenzione, è in carico al cliente. La scelta tra strumenti di orchestrazione per container gestiti e self-hosted dipende da che cosa vogliamo ottenere e da quanta esperienza ed effort vogliamo porre nella manutenzione dell'infrastruttura. Per rendere questa decisione più semplice, proviamo a elencare alcuni vantaggi generici delle soluzioni gestite rispetto a quelle self-hosted. Proviamo a riassumere in queste righe quali sono i vantaggi e i contro di lavorare con l'una o l'altra soluzione:

Tabella 15.1 Soluzioni managed.

Vantaggi	Svantaggi
Controllo completo dell'infrastruttura	Automazione a carico del cliente
Flessibilità rispetto agli strumenti che si vuole utilizzare	Provision manuale
	Custom scaling
	Team dedicato per la progettazione, installazione e manutenzione del cluster

Tabella 15.2 Soluzioni self-hosted.

Vantaggi	Svantaggi
Provisioning dell'infrastruttura in pochi click	Nessun controllo sui nodi <i>control-plane</i>
Integrazione con altri servizi del provider	Poca (o nessuna) personalizzazione sull'infrastruttura sottostante
Minor gestione architetturale	
Alta disponibilità	
Alta scalabilità	
Supporto tecnico	
Sicurezza e compliance documentati	

Le soluzioni che vedremo in questo capitolo sono perlopiù managed: questo perché, come evidente da quanto descritto in precedenza, le soluzioni che prevedono un'alta personalizzazione e flessibilità richiedono anche una conoscenza a livello di reti e di amministrazione del cluster, competenze al di fuori dello scopo di questo manuale. Sperimentare e curiosare è fondamentale, per cui, se vi piace l'idea di provare a tirare su un'infrastruttura ex-novo, troverete in rete moltissima documentazione su come farlo.

AWS

Prima di addentrarci nel mondo di Amazon e dei suoi servizi, facciamo un passo indietro: nei primissimi capitoli abbiamo parlato di diverse tipologie di architetture, come SaaS, PaaS e IaaS. Esatto, diverse tipologie di infrastrutture dove la responsabilità su determinati aspetti del prodotto che utilizziamo sono assegnate al provider, così come al cliente. Amazon, così come altri cloud provider, mette a disposizione diversi strumenti che “inglobano” Kubernetes sotto forma di strumenti gestiti e non: facciamo quindi una prima distinzione tra le soluzioni che illustreremo. Amazon ECS è uno dei primi servizi che viene messo sul mercato con lo scopo di fornire un servizio di gestione dei container altamente scalabile e veloce che semplifica l'esecuzione, l'arresto e la gestione dei container Docker su un cluster. In altre parole, questo servizio è paragonabile, a livello di funzionalità, a Kubernetes o Docker Swarm. ECS esegue i tuoi container su un cluster composto da istanze di macchine virtuali Amazon EC2 (abbreviazione di *Elastic Compute Cloud*) preinstallate con Docker. Gestisce l'installazione dei container, il loro ridimensionamento e il monitoraggio, e tutto grazie a un'API e alla console di gestione di AWS. L'istanza specifica su cui viene eseguito un container e la manutenzione di tutte le altre istanze sono gestite dalla piattaforma e tu non dovrai minimamente pensarci. Questa soluzione, come evidente dalla descrizione appena fatta, completamente gestita e integrata con gli altri servizi di Amazon: infatti, tramite Elastic Container Registry possiamo avere un registry per le immagini, le quali possono essere utilizzate per creare dei container e poi distribuire e rese accessibili tramite Route 53, un Application Load Balancer, Amazon Gateway.

Figura 15.1 Pagina principale di Amazon ECS.

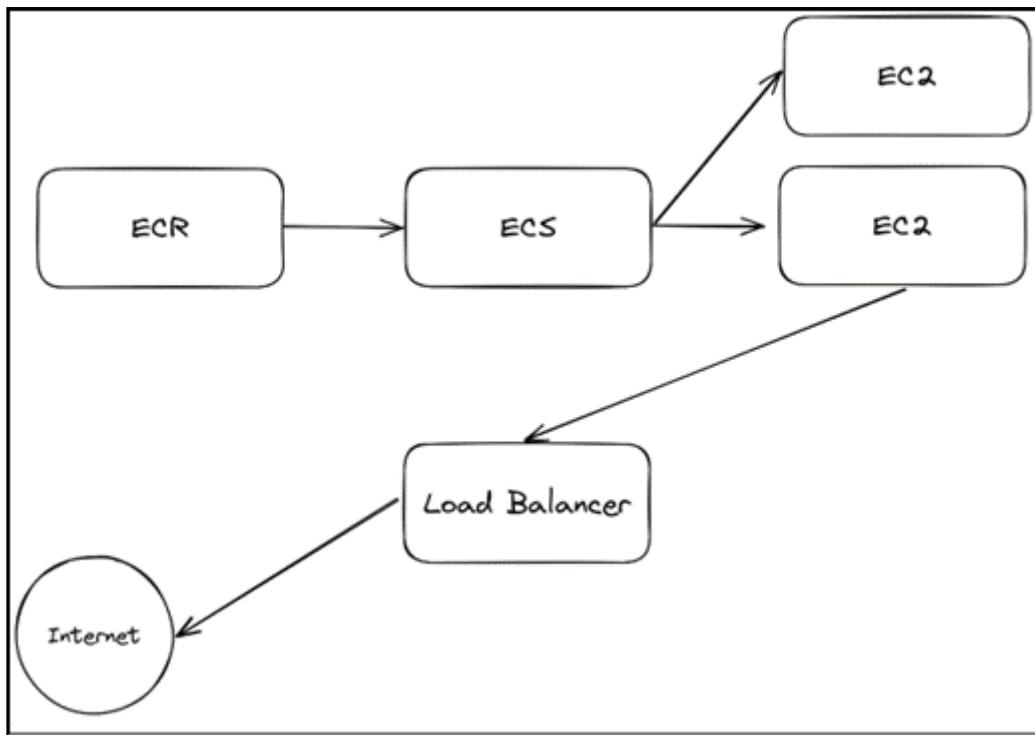


Figura 15.2 Esempio di utilizzo di ECR come registry per le immagini, ECS per la gestione dei container e EC2 come host per i container.

Chiaramente, al posto di alcuni di questi blocchi è possibile scegliere delle soluzioni che sono fuori dal mondo AWS, come GitHub o DockerHub al posto di ECR; questo tipo di scelta dipende da diversi fattori, tra cui c'è anche il costo. Questo servizio, essendo legato alle istanze EC2 (che altro non sono che macchine virtuali, ma on cloud), ha dei costi che dipendono dal tipo di istanza scelta per eseguire il container: infatti, nel mondo AWS esistono diverse famiglie di istanze, con diverse metriche in termini di CPU e memoria che ne variano il costo. Per usufruire di questi servizi, è necessario registrarsi: alcuni di questi consentono, tramite la creazione di un account, di accedere a delle prove gratuite con (o

senza) un limite di tempo, e tra questi c'è EC2: utilizzando delle istanze molto piccole, è possibile avere diverse ore da utilizzare durante l'arco di un mese in maniera totalmente gratuita, per il primo anno dalla registrazione.

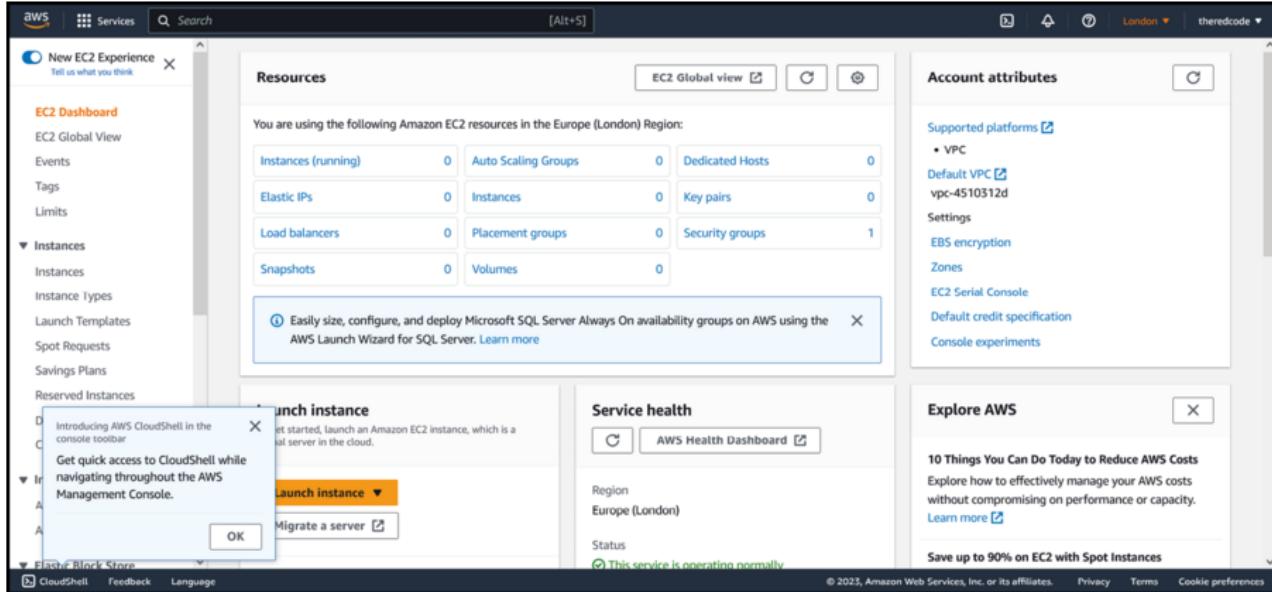


Figura 15.3 Pagina principale di EC2.

Torniamo però a ECS, e proviamo a introdurre un po' di terminologia: vedremo che, anche se cambiamo contesto, molta della logica vista finora è del tutto simile a quanto applicato nel mondo Amazon. Supponiamo che tu stia creando un'applicazione che viene eseguita su un container Docker, la cui base è Nginx e che serve per esporre delle semplici pagine HTML. Se prevedi un elevato traffico, potresti dover eseguire diverse istanze di questo container. Introduciamo quindi il concetto di *Task*, ossia il servizio da eseguire, e di *cluster*, dove si intende un'istanza di un container ECS e di un agente. Il *Task* è il progetto che descrive quali container Docker eseguire e rappresenta l'applicazione. Nel nostro esempio, sarebbe rappresentato dal container Nginx. La sua definizione descrive le immagini da utilizzare, la CPU e la memoria da allocare, le variabili di ambiente, le porte da esporre e il modo in cui interagiscono i container. Molto simile a quanto avveniva con Docker-Compose o con la definizione di un qualsiasi controller Kubernetes, non trovi? Ogni Task ha una sua *Task Definition*, che serve proprio a dettagliare i container; è possibile creare più Task a partire da una Task Definition, a seconda del carico di lavoro.

Un esempio di Task Definition è il seguente: attraverso un file JSON (o YAML), possiamo specificare nel campo `containerDefinitions` quelle che sono le proprietà inerenti all'immagine da creare. In questo caso, abbiamo fornito un nome, il riferimento all'immagine da utilizzare, le risorse in termini di CPU e memoria, quale porta esporre e come gestire logs ed eventuali volumi.

Listato 15.1 Esempio di Task Definition per Nginx tramite JSON

```
{
  "requiresCompatibilities": [
    "EC2"
  ],
  "containerDefinitions": [
    {
      "name": "nginx",
      "image": "nginx:latest",
      "memory": 256,
      "cpu": 256,
    }
  ]
}
```

```

    "essential": true,
    "portMappings": [
      {
        "containerPort": 80,
      "protocol": "tcp"
      }
    ],
    "logConfiguration": {
      "logDriver": "awslogs",
      "options": {
        "awslogs-group": "awslogs-nginx-ecs",
        "awslogs-region": "eu-west-1",
        "awslogs-stream-prefix": "nginx"
      }
    }
  ],
  "volumes": [],
  "networkMode": "bridge",
  "placementConstraints": [],
  "family": "nginx"
}

```

Se il JSON risultasse un po' ostico da leggere, questo è l'equivalente in YAML:

Listato 15.2 Esempio di Task Definition per Nginx tramite YAML

```

---
requiresCompatibilities:
- EC2
containerDefinitions:
- name: nginx
  image: nginx:latest
  memory: 256
  cpu: 256
  essential: true
  portMappings:
    - containerPort: 80
  protocol: tcp
  logConfiguration:
    logDriver: awslogs
  options:
    awslogs-group: awslogs-nginx-ecs
    awslogs-region: eu-west-1
    awslogs-stream-prefix: nginx
volumes: []
networkMode: bridge
placementConstraints: []
family: nginx

```

Questo tipo di operazione può essere eseguita anche tramite l'interfaccia di AWS: accedendo al servizio di ECS, avremo la possibilità di creare una nuova Task Definition facendo clic nel menu di sinistra e seguendo le istruzioni riportate nel wizard, tra cui le informazioni circa l'immagine, porte da esporre e via dicendo, il tutto in maniera visuale.

Amazon Elastic Container Service > Create new task definition

Step 1
Configure task definition and containers

Step 2
Configure environment, storage, monitoring, and tags

Step 3
Review and create

Configure task definition and containers

Task definition configuration

Task definition family [Info](#)
Specify a unique task definition family name.

Up to 255 letters (uppercase and lowercase), numbers, hyphens, and underscores are allowed.

Container - 1 [Info](#)

Container details
Specify a name, container image, and whether the container should be marked as essential. Each task definition must have at least one essential container.

Name	Image URI	Essential container
wordpress	repository-url/image:tag	<input checked="" type="checkbox"/> Yes

Private registry [Info](#)
Store credentials in Secrets Manager, and then use the credentials to reference images in private registries.
 Private registry authentication

Figura 15.4 Configurazione di un Task in ECS.

Oltre al Task, esiste il concetto di Service: questo definisce, a partire da una Task Definition, il numero di Task minimi e massimi da gestire per la scalabilità automatica e il bilanciamento del carico. Questo oggetto ci torna utile per poter avere il controllo delle risorse in esecuzione: più Task vuol dire più container, e più container significa avere più istanze ECS, che hanno un costo.

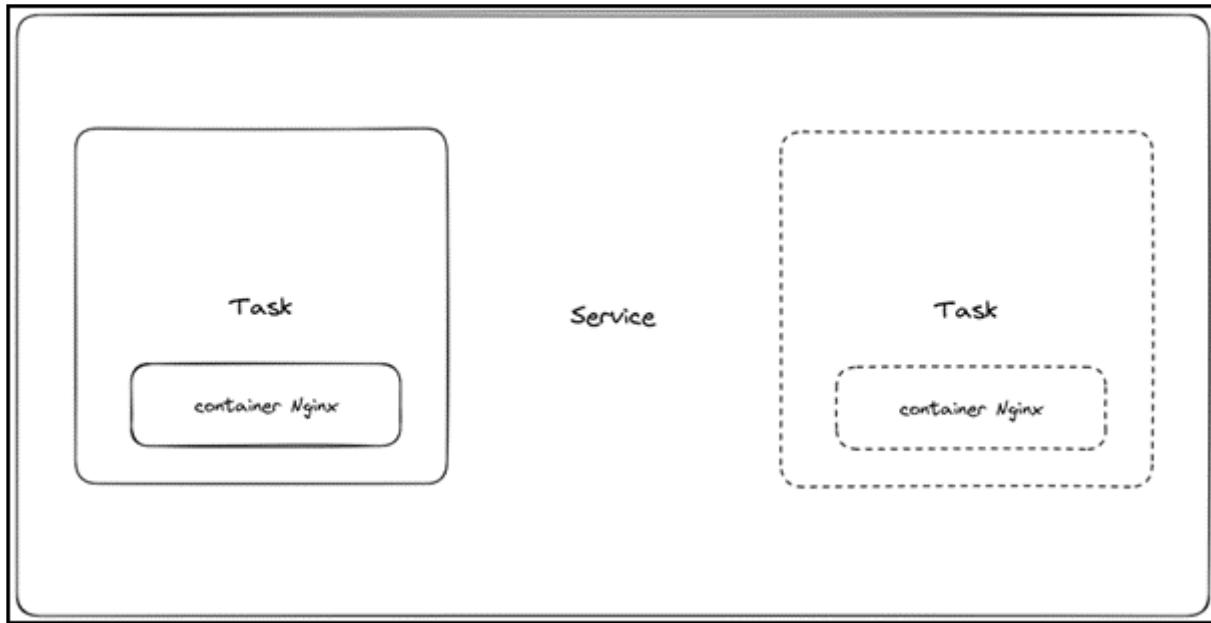


Figura 15.5 Rappresentazione del concetto di Task e del concetto di Service.

Ora che abbiamo un Service e abbiamo un Task, questo deve essere eseguito da qualche parte per essere accessibile: per questo si parla di *cluster* e la gestione del container sarà in esecuzione su una o più *ECS container instance*. Un ECS container instance non è altro che un'istanza EC2 che esegue

Docker e l'*ECS container agent*, ossia il componente che si occupa della comunicazione tra ECS e l'istanza, fornendo informazioni utili sullo stato dei container in esecuzione e di eventuali altri container che devono essere aggiunti.

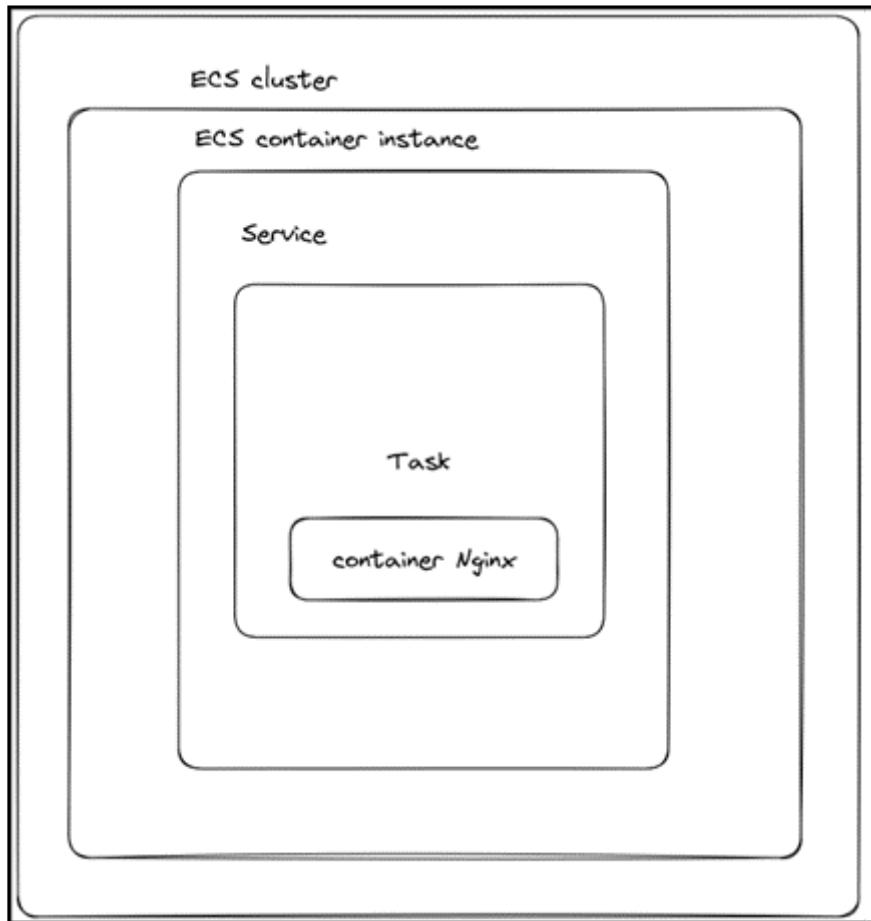


Figura 15.6 Come un cluster ECS e le diverse risorse (Task, Service) sono organizzate.

Anche i cluster possono essere creati attraverso la console Amazon: sempre tramite il menu di sinistra, possiamo definire i dettagli di creazione di un cluster, la rete VPC da utilizzare e quali sono le risorse che stiamo utilizzando e che devono essere gestite tramite questo oggetto:

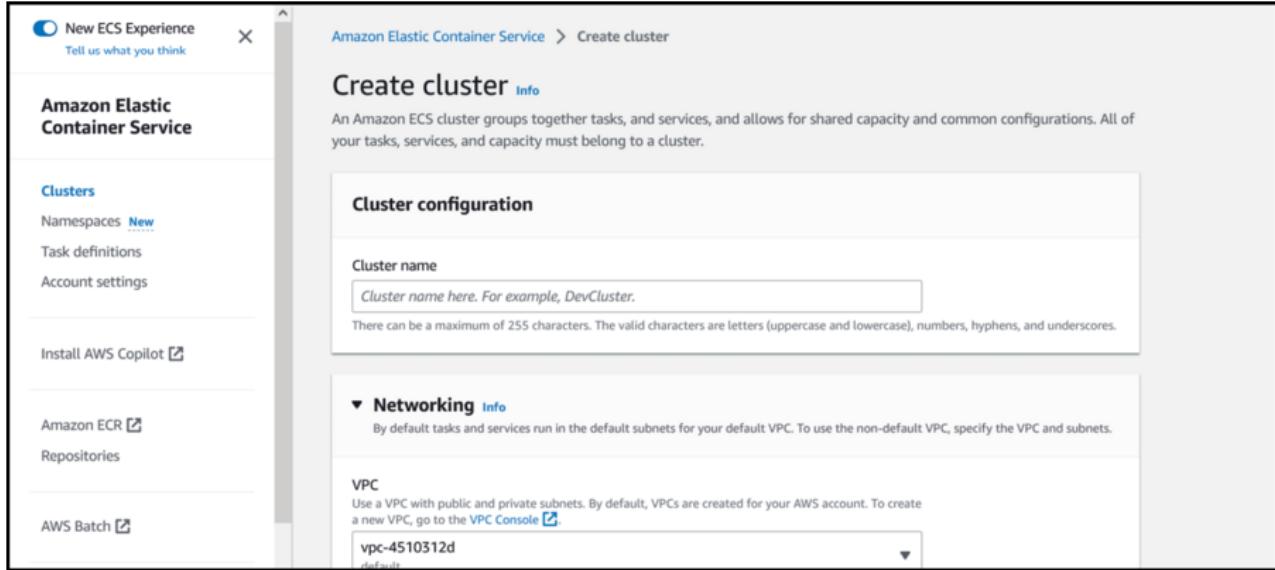


Figura 15.7 Creazione di un cluster ECS.

ECS può essere utilizzato con le istanze EC2 di Amazon, ma anche con uno strumento che si chiama *Fargate*: si tratta di un motore serverless per container che astrae l'infrastruttura sottostante e può essere utilizzato per avviare ed eseguire container senza doversi preoccupare di effettuare il provisioning delle istanze EC2. Come servizio gestito per l'orchestrazione dei container, ci sono molti aspetti di ECS che aiutano a semplificare la gestione dei container, tra cui la creazione, l'impostazione e la manutenzione dei cluster. Tuttavia, ECS non automatizza tutto ciò che riguarda la gestione dei cluster. Lascia comunque in carico al cliente il livello di elaborazione, richiedendo agli utenti di eseguire il provisioning, ridimensionare, monitorare, proteggere e gestire autonomamente le istanze EC2 sottostanti. Sebbene coloro che usano ECS abbiano un controllo più granulare sulle operazioni del cluster, questo ha come controparte la gestione infrastrutturale: quando i container ECS vengono distribuiti alle istanze EC2, spetta all'utente determinare quale tipo di istanza utilizzare e quando ridimensionarle. Al contrario, l'esecuzione di ECS con Fargate, tuttavia, elimina la necessità di eseguire manualmente il provisioning, la scalabilità e la gestione delle istanze di calcolo. Gli utenti creano un cluster, vi aggiungono le proprie applicazioni e specificano i requisiti delle risorse (CPU e memoria) e, quando i container ECS vengono distribuiti, Fargate avvierà, eseguirà e gestirà i server preconfigurati che soddisfano i requisiti del container. Questi vantaggi in termini di risparmio di tempo eliminano l'onere operativo della gestione dell'elaborazione, ma il compromesso consiste in funzionalità limitate, minor controllo e costi potenzialmente più elevati.

Ma Amazon ECS non è l'unico servizio che lavora come o con Kubernetes: esiste infatti *EKS*, che sta per *Elastic Kubernetes Service*: si tratta di un servizio completamente gestito che semplifica agli utenti l'esecuzione di cluster Kubernetes su AWS senza la necessità di preoccuparsi dei nodi *control-plane*, né di attività come il provisioning, gli aggiornamenti e l'applicazione di patch ai propri sistemi. Definizione a parte, diamo un'occhiata ai vantaggi di Amazon EKS: esegue l'infrastruttura di gestione Kubernetes su più zone di disponibilità AWS, togliendo il pensiero di eventuali disservizi dovuti a problemi su una specifica zona di disponibilità (attività comunque quasi del tutto impossibile). L'infrastruttura in esecuzione su Amazon EKS adotta un approccio *security by design*, impostando un canale di comunicazione protetto e critografato tra i nodi applicativi e le API di Kubernetes. Non a caso, Amazon è tra le aziende che contribuiscono attivamente al progetto di Kubernetes e al lavoro

fatto dalla community. Chiaramente, le applicazioni gestite da Amazon EKS sono completamente compatibili con delle applicazioni eseguite su qualsiasi ambiente Kubernetes standard.

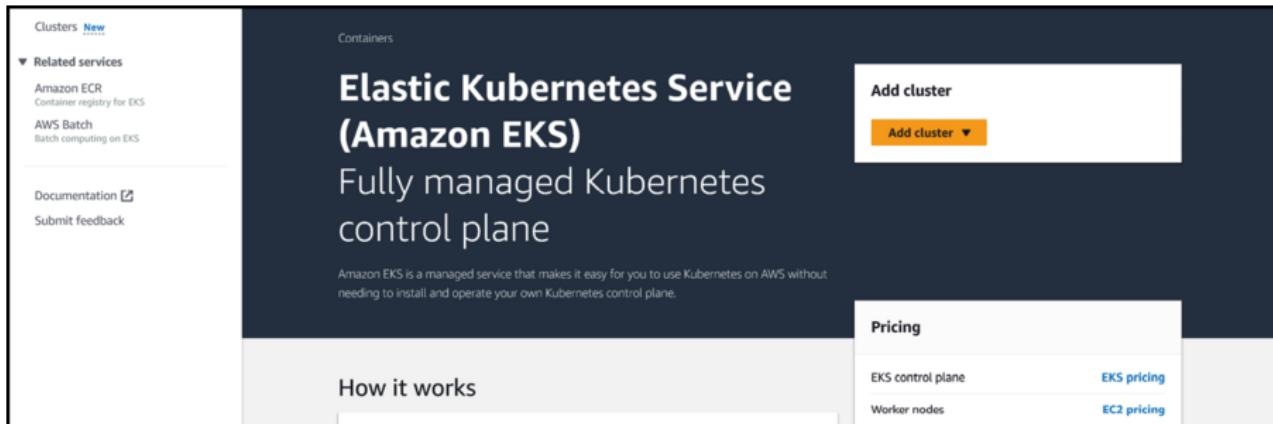


Figura 15.8 Pagina principale di Amazon EKS.

Ma come funziona? Innanzitutto, si crea un cluster Amazon EKS tramite la console di gestione AWS o con l'AWS CLI; una volta creato il cluster, si avviano i nodi applicativi che ospiteranno le applicazioni e si aggiorna la configurazione del cluster aggiungendoli al cluster Amazon EKS creato in precedenza. Quando il cluster è pronto, è possibile configurare gli strumenti Kubernetes per comunicare con il cluster (come `kubectl`) e quindi rilasciare e gestire le applicazioni così come si farebbe con qualsiasi altro cluster Kubernetes. Anche in questo caso, dietro alle quinte di EKS ci sono EC2 e Fargate, per cui i costi per ogni cluster variano molto.

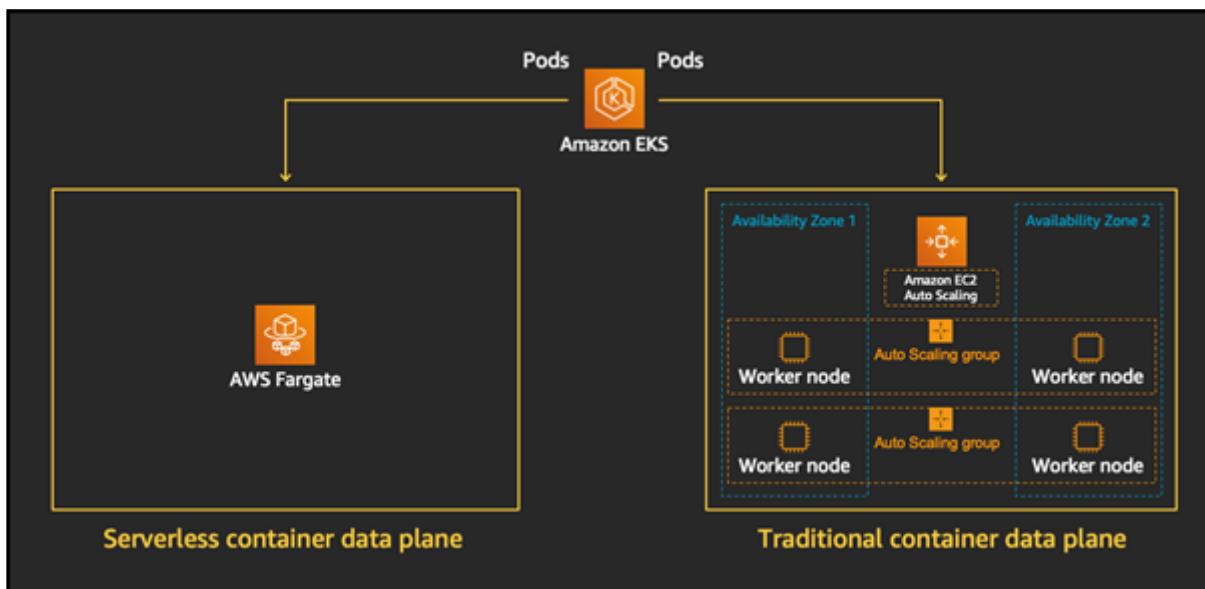


Figura 15.9 Architettura di Amazon EKS: è possibile prevedere l'utilizzo di Fargate per lavorare in modalità serverless, oppure di affidarsi a delle istanze EC2 e sfruttare l'Auto Scaling per gestire il flusso di lavoro che cambia nel tempo.

Deploy di un'applicazione tramite EKS

Creare un cluster ed eseguire un'applicazione è piuttosto semplice: provare per credere. Per prima cosa, creiamo un ruolo tramite IAM, di modo che Kubernetes possa utilizzarlo per creare delle risorse

AWS, come delle istanze EC2. Andiamo quindi sulla console di IAM, facciamo clic su *Roles* e creiamo un nuovo ruolo, assegnandogli le policy relative al cluster Amazon EKS:

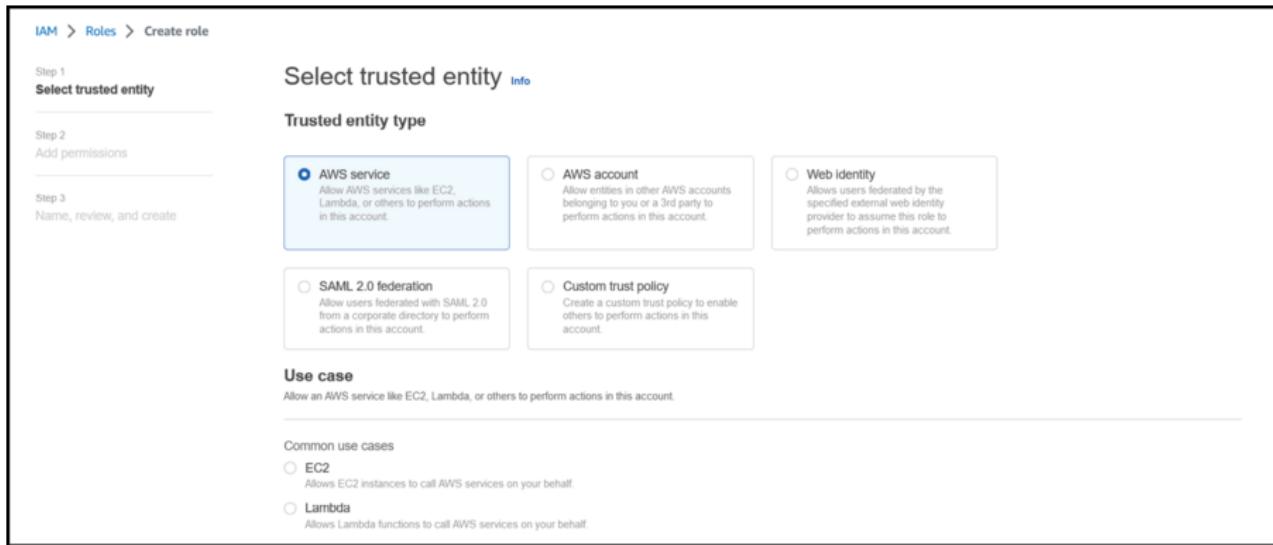


Figura 15.10 Creazione di un ruolo all'interno di IAM.

Amazon EKS richiede anche una *Virtual Private Cloud* (VPC) per eseguire il cluster. Per creare questa VPC, puoi procedere in autonomia oppure usare AWS CloudFormation: fai clic su *Crea stack* e, nella pagina *Select Template*, seleziona l'opzione *Specify an Amazon S3 template URL*; inserisci l'URL seguente e, dopo aver completato l'inserimento di tutti i dettagli e averli controllati, puoi fare clic su *Crea* per procedere.

Listato 15.3 URL per il template della creazione di una VPC con subnet private e pubbliche tramite CloudFormation

```
https://s3.us-west-2.amazonaws.com/amazon-eks/cloudformation/2020-10-29/amazon-eks-vpc-private-subnets.yaml
# Documentazione ufficiale
https://docs.aws.amazon.com/eks/latest/userguide/creating-a-vpc.html
```

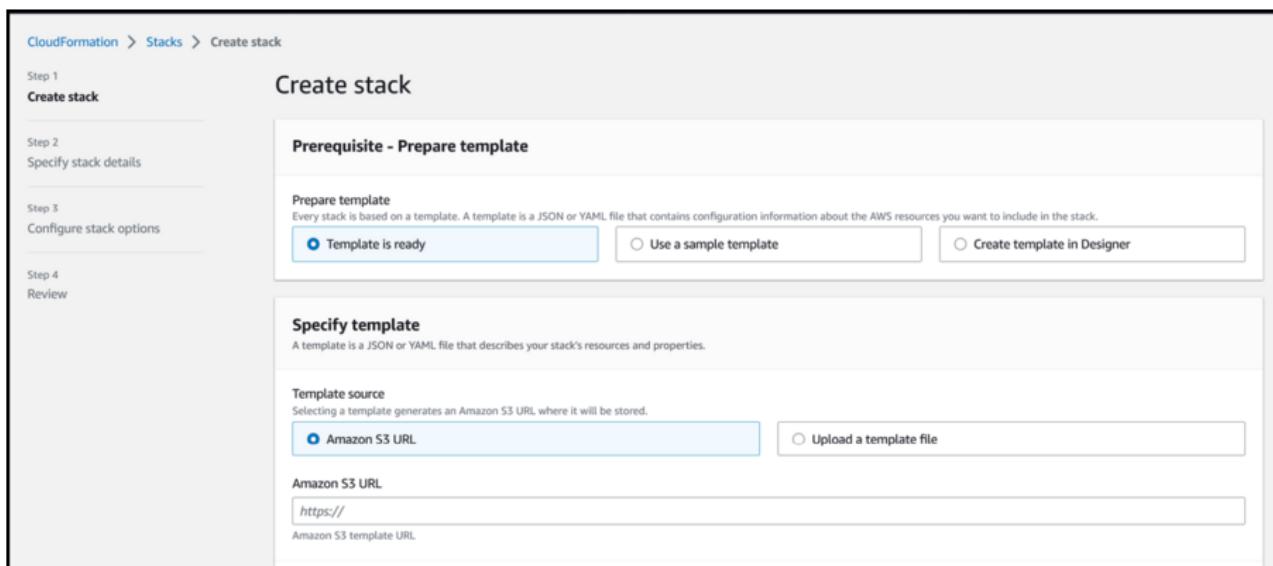


Figura 15.11 Creazione dello stack per l'avvio del cluster tramite CloudFormation e il file YAML riportato.

A questo punto, possiamo procedere con la creazione del cluster: andiamo sul servizio EKS e clicchiamo su *Crea*, inserendo nel primo step le informazioni relative al nome che il cluster assumerà, la versione di Kubernetes che vogliamo utilizzare e il ruolo creato in precedenza. Negli step successivi, selezioniamo la VPC creata, che livello di logging vogliamo mantenga CloudWatch Logs (il servizio di AWS che salva e permette di analizzare i log), eventuali altri strumenti come `kube-proxy` e `CoreDNS`, e facciamo clic su *Crea*.

Figura 15.12 Configurazione del cluster EKS.

Questo ci permetterà di creare i nodi *control-plane* del nostro cluster; attendi che lo stato del tuo cluster venga visualizzato come *ATTIVO*. Infatti, se avvii i tuoi nodi applicativi prima che il cluster sia in esecuzione, questi non riusciranno ad aggiungersi al cluster e dovrà riavviarlo. Per aggiungere poi i nodi su cui le applicazioni gireranno, avremo bisogno di creare le risorse: possiamo sempre procedere manualmente o sfruttare CloudFormation come fatto in precedenza per la VPC. Un esempio di template da utilizzare è quello che segue.

Listato 15.4 URL per il template della creazione dei nodi computazionali

```
https://amazon-eks.s3-us-west-2.amazonaws.com/cloudformation/2019-10-08/amazon-eks-nodegroup.yaml
# Documentazione ufficiale
https://docs.aws.amazon.com/eks/latest/userguide/managed-node-groups.html
```

In questo file possiamo andare a definire informazioni come il numero di nodi minimi, la gestione della sua scalabilità, il tipo di istanza da utilizzare per l'avvio di questi nodi, la rete e via dicendo; tutti dettagli che riguardano la costruzione dei nodi computazionali e del modo in cui ci aspettiamo che lavorino. A questo punto, potremo utilizzare `kubectl` per comunicare con il cluster Kubernetes, insieme a un sistema che ci permetta di autenticarci con il cluster: ci servirà quindi un utente creato tramite IAM che ci permetta di assumere un'identità precisa per collegarci al cluster. Creiamo un utente e, tramite delle chiavi di accesso, configuriamo il profilo locale con il quale collegarci.

Figura 15.13 Recupero delle credenziali per accedere al cluster.

Installiamo, quindi, anche la CLI di AWS, tramite la quale potremo eseguire i comandi che ci permetteranno di inserire all'interno del `kubeconfig` locale i dati del cluster EKS, così da poterci collegare. Dopo aver installato questi strumenti, è possibile utilizzare il comando `aws eks update-kubeconfig` per creare o aggiornare il `kubeconfig` relativo al cluster e testare la tua configurazione eseguendo un comando di esempio:

Listato 15.5 Connessione al cluster

```
aws eks update-kubeconfig --name mycluster

kubectl get svc
>>>
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)        AGE
kubernetes    ClusterIP   xxx           <none>         443/TCP       10m
```

Ora, così come fatto con il cluster utilizzato in locale, potremo eseguire le nostre applicazioni: creiamo quindi, per esempio, un'applicazione di test con Nginx composta dal solito Deployment e Service (di tipo `LoadBalancer`), e creiamo questi due oggetti all'interno del namespace di default. Al termine potremo utilizzare il comando `kubectl get svc` e successivamente `kubectl get describe` per ottenere l'indirizzo del Load Balancer a cui raggiungere l'applicazione. Sarà infatti ELB, acronimo di *Elastic Load Balancer*, a fornirci un punto di ingresso per accedere alla pagina principale di Nginx, e quindi al Service.

Listato 15.6 Output

```
kubectl get svc nginx
>>>
Name:            nginx
Namespace:       k8s-training
Labels:          <none>
Annotations:    <none>
Selector:        com.docker.project=tutorial
Type:            LoadBalancer
IP Family Policy: SingleStack
IP Families:    IPv4
IP:              10.100.232.115
IPs:             10.100.232.115
LoadBalancer Ingress:  xxx.eu-central-1.elb.amazonaws.com
```

```

Port: 80-tcp 80/TCP
TargetPort: 80/TCP
NodePort: 80-tcp 30127/TCP
Endpoints: xxx:80
Session Affinity: None
External Traffic Policy: Cluster
Events:
  Type Reason Age From Message
  ---- ---- - - -
Normal EnsuringLoadBalancer 8m47s service-controller Ensuring load balancer
Normal EnsuredLoadBalancer 8m44s service-controller Ensured load balancer

```

Azure

Azure Kubernetes Service (AKS) è il servizio di orchestrazione di container gestito, disponibile nel cloud pubblico di Microsoft Azure. Ogni azienda può usare AKS per distribuire, ridimensionare e gestire i container Docker e le applicazioni basate su container in un cluster che può essere creato con pochi semplici clic. Come visto per gli altri provider, è possibile registrarsi sulla console di Azure con un account che mette a disposizione alcune delle risorse in forma gratuita insieme a un credito iniziale di circa 200 dollari; il caso di esempio che vedremo per la creazione di un cluster può essere riprodotto sfruttando il *free tier* di Azure. Ci colleghiamo alla console di Azure tramite il sito <https://portal.azure.com/> e selezioniamo, all'interno dei servizi, *Kubernetes*: la pagina che ci verrà mostrata ci permette di creare un cluster da zero, semplicemente facendo clic sul pulsante al centro della schermata.

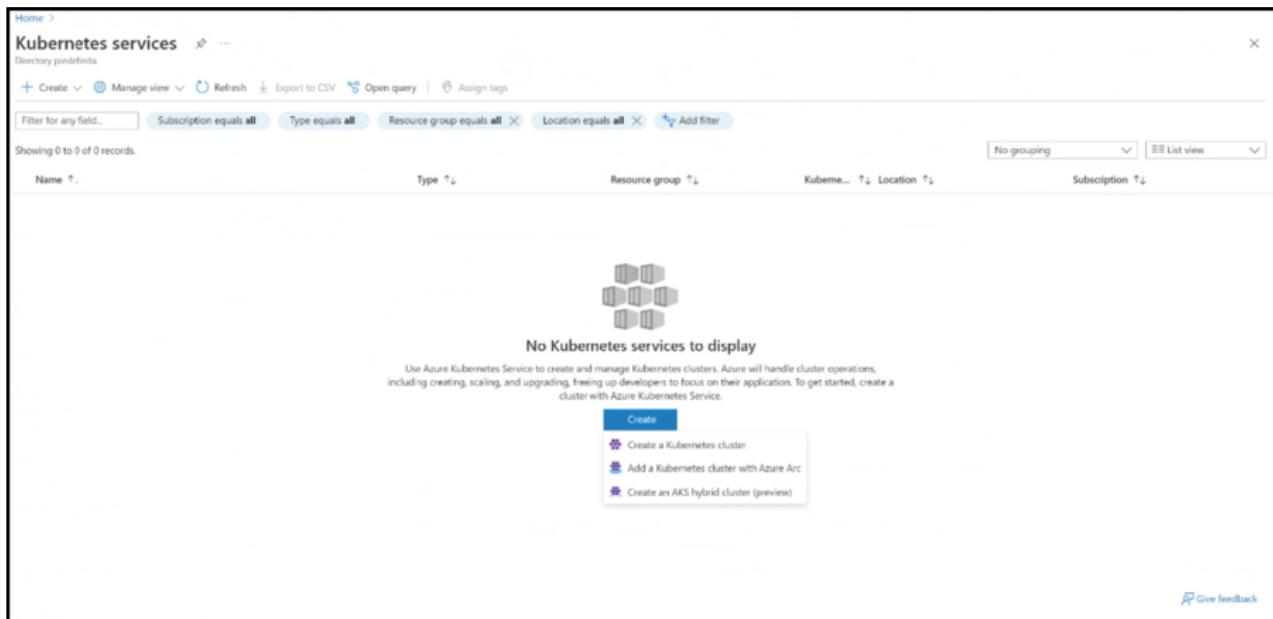


Figura 15.14 Pagina principale del servizio relativo a Kubernetes.

Seguendo quanto mostrato nella finestra successiva, inseriamo i dettagli relativi al cluster come il nome e la regione dove vogliamo che questo cluster risieda, e poi selezioniamo il piano: scegliendo l'opzione *Free*, possiamo utilizzare un cluster minimale che non comporta costi aggiuntivi per il nostro account.

Home > Kubernetes services >

Create Kubernetes cluster

Project details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ

DEVAzureSubscription

Resource group * ⓘ

(New) demo-aks-rg

Create new

Cluster details

Cluster preset configuration

Dev/Test

To quickly customize your Kubernetes cluster, choose one of the preset configurations above. You can modify these configurations at any time.
[Learn more and compare presets](#)

Kubernetes cluster name * ⓘ

demo-aks

Region * ⓘ

(Europe) West Europe

Availability zones ⓘ

None

AKS pricing tier ⓘ

Free

Kubernetes version * ⓘ

1.24.9 (default)

Automatic upgrade ⓘ

Enabled with patch (recommended)

Primary node pool

The number and size of nodes in the primary node pool in your cluster. For production workloads, at least 3 nodes are recommended for resiliency. For development or test workloads, only one node is required. If you would like to add additional node pools or to see additional configuration options for this node pool, go to the 'Node pools' tab above. You will be able to

[Review + create](#)

[< Previous](#)

[Next : Node pools >](#)

Figura 15.15 Creazione di un cluster Kubernetes.

Negli step seguenti, definiamo gli ultimi dettagli, tra cui il tipo di istanza da utilizzare per l'infrastruttura Kubernetes (ci atteniamo sempre a quelle disponibili per il *free tier*) così come le subnet da utilizzare (lasciamo quelle di default) e, dopo aver rivisto e controllato tutti i dettagli, confermiamo la creazione. Questa impiegherà qualche minuto (fino a 10 minuti) e il progresso verrà mostrato nella schermata successiva: quando l'installazione sarà ultimata, verrà mostrata una notifica di completamento dell'operazione.



Figura 15.16 Messaggio di stato riguardo l'avvio del cluster.

A questo punto, tornando nel menu principale del servizio principale, possiamo vedere nell'elenco dei cluster quello appena creato, e possiamo anche vedere i dettagli relativi alla sua infrastruttura, come il numero di nodi, la versione di Kubernetes utilizzata e i range di indirizzi IP utilizzati per Services e Pods.

Networking	
API server address	demo-aks-dns-ull1fdtd8t.hcp.westeurope.azurek8s.io
Network type (plugin)	Kubenet
Pod CIDR	10.244.0.0/16
Service CIDR	10.0.0.0/16
DNS service IP	10.0.0.10
Docker bridge CIDR	172.17.0.1/16
Network Policy	None
Load balancer	Standard
HTTP application routing	Not enabled
Private cluster	Not enabled
Authorized IP ranges	Not enabled
Application Gateway ingress controller	Not enabled

Figura 15.17 Informazioni circa il cluster creato e le sue proprietà.

Creazione di una pipeline con Automated deployments

Si tratta di uno strumento da poco introdotto su Azure (che sostituisce Deployment Center, deprecato da marzo 2023) e che permette di ottenere una pipeline per distribuire un'applicazione basata su container il cui codice sorgente è disponibile su GitHub nella cartella relativa al capitolo corrente; questo servizio automatizza e semplifica il processo di configurazione tramite le GitHub Actions che crea e modifica per avviare l'applicazione all'interno del cluster Azure Kubernetes Service. Vediamo un esempio pratico, utilizzando uno dei tanti esempi già visti all'interno del manuale: selezioniamo la voce relativa al servizio dal menu di sinistra e creiamo la prima pipeline.

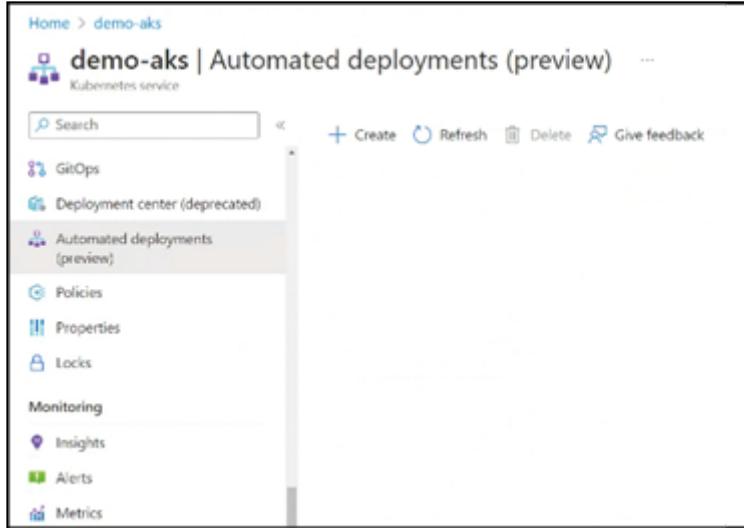


Figura 15.18 Selezione del servizio “Automated deployments” nella console di Azure.

Per iniziare, forniamo un nome al workflow: questo rappresenta la pipeline. Dopodiché, autorizziamo il cloud di Azure ad accedere a GitHub per poter leggere l’elenco di repository presenti nel nostro account e poi selezioniamo quello che vogliamo utilizzare, con il relativo branch. Nello step successivo, definiamo qual è il Dockerfile e il contesto per eseguire la build, con lo stesso approccio che già avevamo visto con Docker Compose: a partire da questa cartella, recupererà i file necessari per la creazione dell’immagine.

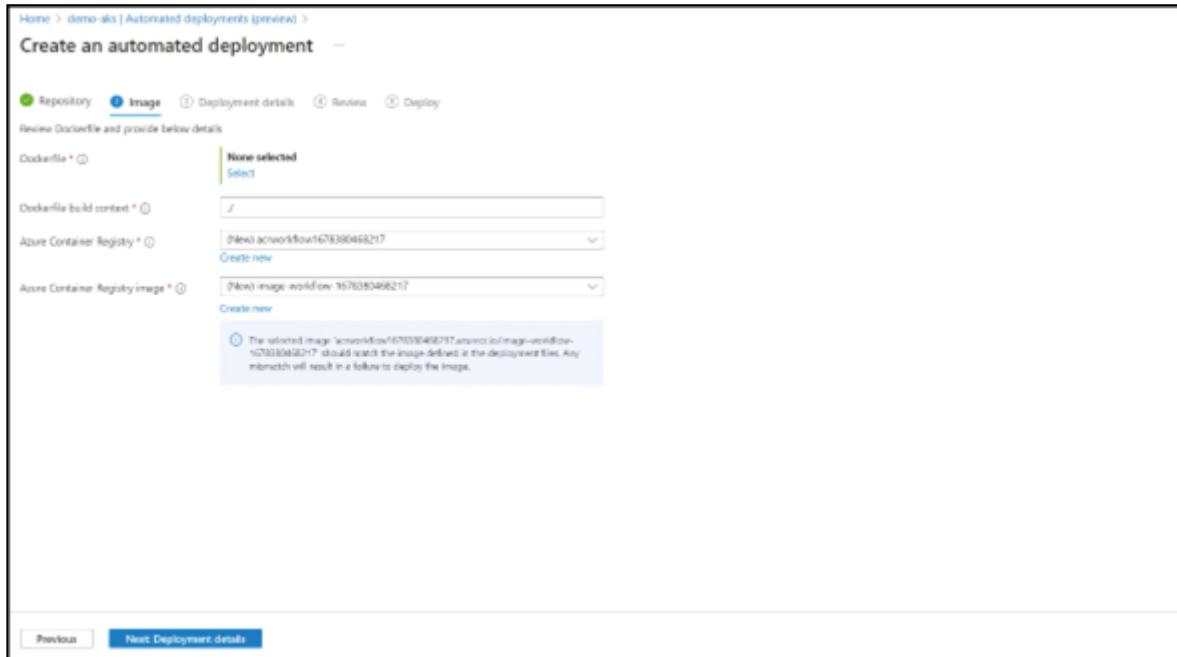


Figura 15.19 Creazione di una pipeline tramite il servizio “Automated Deployment”.

Facendo clic su **Select**, si aprirà infatti una finestra sulla destra che permetterà di selezionare il Dockerfile direttamente dal repository GitHub specificato.

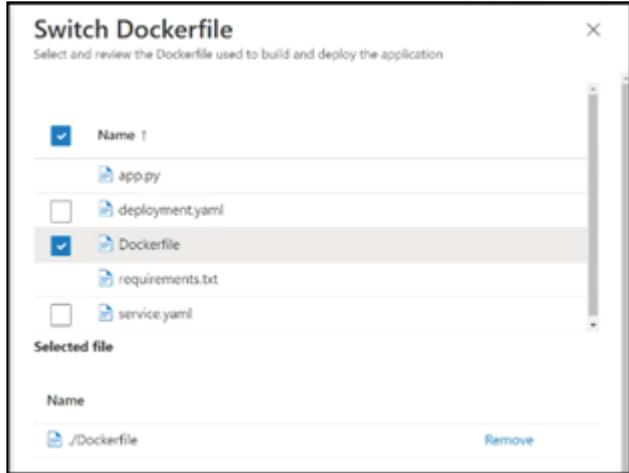


Figura 15.20 Selezione del Dockerfile.

Ora, attenzione: nello step successivo ci verrà chiesto di fornire anche i manifest relativi alle risorse Kubernetes da utilizzare. Questo vuol dire che, per esempio, se nel file relativo al Deployment avessimo il riferimento a un'immagine disponibile sul DockerHub che abbiamo compilato in precedenza, la build non avrebbe nessun effetto. Non a caso, in azzurro sotto al campo *Azure Container Registry Image* viene mostrato un messaggio come quello che segue.

Listato 15.7 Informazione relativa all'immagine

```
The selected image 'acrworkflow1678383134720.azurecr.io/image-workflow-1678383134720'
should match the image defined in the deployment files. Any mismatch will result in a
failure to deploy the image.
```

Questo vuol dire che, prima di procedere con il prossimo step, dovremo andare a sostituire il riferimento all'immagine nel file del controller presente nel repository utilizzato: nel file `deployment.yaml`, come valore del campo `image`, inseriamo quello riportato dalla console Azure. Aggiorniamo il file, salviamo e committiamo la modifica sul repository GitHub, e poi andiamo avanti con il passaggio successivo.

Listato 15.8 Informazione relativa all'immagine

```
...
spec:
  containers:
    - env:
        - name: APP_VERSION
          value: "1.0.0"
      image: acrworkflow1678383134720.azurecr.io/image-workflow-
1678383134720
      name: flask-app
      ports:
        - containerPort: 5000
      resources: {}
    restartPolicy: Always
```

A questo punto, possiamo scegliere che tipo di risorse utilizzare per avviare l'applicazione, tra l'opzione che prevede l'utilizzo di Helm, oppure le risorse “tradizionali”; per il caso di esempio, selezioniamo i file Kubernetes sempre dalla finestra che si apre sulla destra, e definiamo anche il nome del namespace in cui questa applicazione verrà avviata. Possiamo scegliere se utilizzare un nuovo namespace o uno già presente:

Dopo aver controllato che tutti i dettagli inseriti siano corretti, procediamo alla creazione. Questo processo comporta la build dell'immagine attraverso il Dockerfile, il push sul repository interno di Azure (chiamato *Azure Container Registry*, o ACR), la creazione del namespace, l'aggiunta delle credenziali nei Secretsdi GitHub per poter operare all'interno del repository, e infine una richiesta di pull nel repository per poter creare il file relativo alla pipeline. Infatti, quando tutti gli step mostrati nella schermata saranno completati, dovremo andare sul repository per poter confermare la pull request, attraverso la quale Azure avrà aggiunto un file per descrivere il flusso di lavoro della GitHub Action. Questa avrà un aspetto simile al seguente: il file YAML descrive tutti i passaggi che servono per eseguire la build dell'immagine e l'aggiornamento delle risorse a essa associate all'interno del cluster, attraverso i `job`. Questi lavorano sfruttando le credenziali presenti all'interno dei Secrets creati in precedenza, e grazie all'integrazione tra Azure e GitHub.

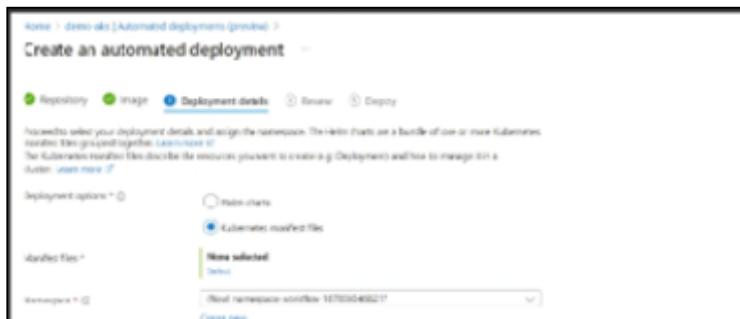


Figura 15.21 Selezione dei file YAML per le risorse Kubernetes.

Listato 15.9 Informazione relativa all'immagine

```

name: aks-workflow-name
"on":
  push:
    branches:
      - main
  workflow_dispatch: {}
env:
  ACR_RESOURCE_GROUP: acrworkflow1678380468217
  AZURE_CONTAINER_REGISTRY: demoaksacr834947
  CLUSTER_NAME: demo-aks
  CLUSTER_RESOURCE_GROUP: demo-aks-rg
  CONTAINER_NAME: demoaksacrimage
  DEPLOYMENT_MANIFEST_PATH: |
    ./deployment.yaml
    ./service.yaml
jobs:
  buildImage:
    ...
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: azure/login@v1.4.3
        name: Azure login
        with:
          client-id: ${{ secrets.AZURE_CLIENT_ID }}
          subscription-id: ${{ secrets.AZURE_SUBSCRIPTION_ID }}
          tenant-id: ${{ secrets.AZURE_TENANT_ID }}
      - name: Build and push image to ACR
        run: az acr build --image ${{ env.CONTAINER_NAME }}:${{ github.sha }} --
registry ${{ env.AZURE_CONTAINER_REGISTRY }} -g ${{ env.ACER_RESOURCE_GROUP }} -f
./Dockerfile ./
  deploy:

```

```

...
needs:
  - buildImage
steps:
  - uses: actions/checkout@v3
  - uses: azure/login@v1.4.3
    name: Azure login
    with:
    ...
  - uses: azure/use-kubelogin@v1
    name: Set up kubelogin for non-interactive login
    with:
      kubelogin-version: v0.0.25
  - uses: azure/aks-set-context@v3
    name: Get K8s context
    ...
  - uses: Azure/k8s-deploy@v4
    name: Deploys application
    with:
      action: deploy
      images: ${{ env.AZURE_CONTAINER_REGISTRY }}.azurecr.io/${{ env.CONTAINER_NAME }}:${{ github.sha }}
      manifests: ${{ env.DEPLOYMENT_MANIFEST_PATH }}
      namespace: namespace-workflow-1678380468217

```

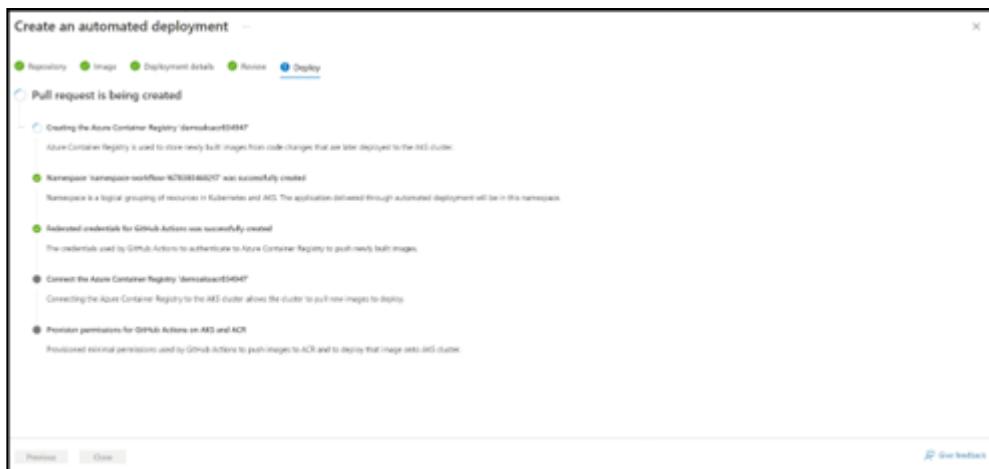


Figura 15.22 Avvio del deployment automatico in corso.

Una volta che la merge sarà stata confermata, tornando sulla console Azure, vedremo un risultato del genere: all'interno del servizio per il deploy automatico, sarà presente l'elenco delle pipeline, dove verrà anche riportato lo stato rispetto all'ultima esecuzione della GitHub Action che, in questo caso, ha avuto successo. Se invece questa fallisse, potremo comunque recuperare le informazioni relative agli step che non hanno avuto successo tramite la dashboard di GitHub, dove ci vengono mostrati i singoli step che corrispondono alla pipeline definita.

Workflow	Pull request	Last run status	Last run time	Workloads	Created on
aks-workflow-name t	Merged	Succeeded	2023-03-09T16:54:56Z	namespace-workflow-1...	2023-03-09T16:52:16.36...

Figura 15.23 Riepilogo delle pipeline presenti.

Questo vuol dire che anche le risorse Kubernetes saranno disponibili tramite le relative voci nel menu di sinistra nella sezione relativa ai *Workloads*, dove potremo vedere i Pod: questa interfaccia apparirà molto simile ad altre dashboard, dal momento che consente di mostrare il file YAML che definisce la risorsa, così come i log applicativi prodotti dal container in esecuzione, gli eventi relativi al suo ciclo di vita all'interno del cluster e anche un'analisi delle risorse utilizzate.

Figura 15.24 Sezione dedicata al dettaglio delle risorse applicative appena create; qui vediamo i log del Pod, ma è possibile selezionare anche gli eventi e le metriche tramite il menu a sinistra.

Ora che l'applicazione è stata avviata con successo, potremo fare clic sul menu di sinistra nella sezione relativa ai Services, dove ci verrà riportato l'indirizzo IP pubblico associato all'applicazione con la relativa porta, che sarà la 5000 per il caso di esempio. Se apriamo il browser e digitiamo queste informazioni, il risultato sarà un messaggio di *Hello world* direttamente dall'applicazione Flask di Python.

Google Cloud Platform

Google Cloud offre un proprio servizio Kubernetes gestito chiamato *Google Kubernetes Engine*, noto semplicemente anche come GKE. Diverse aziende utilizzano questo strumento per distribuire le proprie applicazioni in produzione; uno dei migliori esempi proviene proprio dal gioco Pokemon Go, che è interamente deployato su un cluster GKE e, di fatto, è il più grande rilasciato tramite GKE di sempre. Rispetto ad altri servizi gestiti come EKS (AWS) e AKS (Azure), GKE è relativamente facile da configurare e utilizzare. Utilizzando GKE, è possibile evitare di dover configurare manualmente tutto ciò che concerne l'infrastruttura Kubernetes, dal momento che questo è a carico di Google Cloud. GKE offre due tipi di cluster managed: *Autopilot*, un cluster in cui tutte le operazioni relativa all'installazione e alla manutenzione dell'infrastruttura sono gestite da Google Cloud, e la modalità *standard* dove, a eccezione dei nodi *control-plane*, si ha la responsabilità della gestione dell'infrastruttura sottostante (nodi, ridimensionamento ecc.). Chiaramente, nel primo caso attività come lo scaling dei nodi, piuttosto che la configurazione dei nodi è trasparente per l'utente, che deve preoccuparsi solamente di installare le proprie applicazioni e che, però, avrà un prezzo relativo a ogni Pod che viene eseguito; nel secondo

caso, i costi si riducono e sono relativi al numero di istanze che compongono il cluster, anche se di contro si ha tutta la gestione della scalabilità del sistema e del cluster.

Questo esempio presuppone che si abbia già un account Google Cloud Platform. Se non si dispone di un account, è sufficiente andare su <https://console.cloud.google.com/getting-started> e crearne uno. Se invece si ha un account, ma si sta provando Google Cloud Platform per la prima volta, è bene verificare i limiti di quota delle risorse gratis o inclusi nel pacchetto scelto, considerato che, per chi si iscrive per la prima volta, ci sono solitamente 300 dollari di credito da utilizzare per provare i loro servizi, più alcuni che sono gratuiti "a vita" (o quasi). Per creare un cluster con GKE, ci collegiamo alla console di Google Cloud Platform attraverso questo indirizzo: <https://console.cloud.google.com/>. Creiamo il nostro primo progetto e poi clicchiamo sul pulsante *Crea un cluster GKE*. Alternativamente, possiamo installare Google Cloud SDK (sito ufficiale: <https://cloud.google.com/sdk>) per lavorare da riga di comando attraverso il tool `gcloud`: questo ci consente di creare un cluster e definire le relative configurazioni direttamente tramite un qualsiasi terminale.

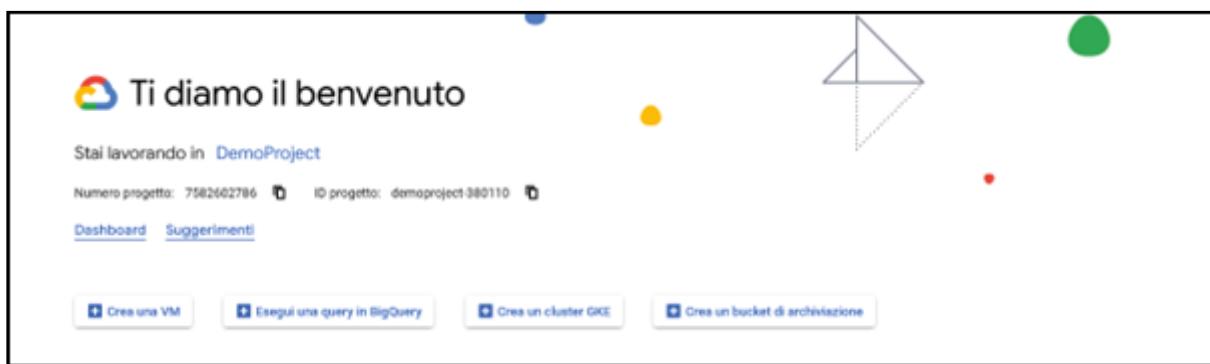


Figura 15.25 Pagina principale della Google Console.

Per lavorare con GKE, abbiamo quindi due modalità: mentre la *Autopilot* ha diversi automatismi e un costo superiore, quella standard è perfetta per fare un esperimento rapido: clicchiamo su *Configure* e procediamo.

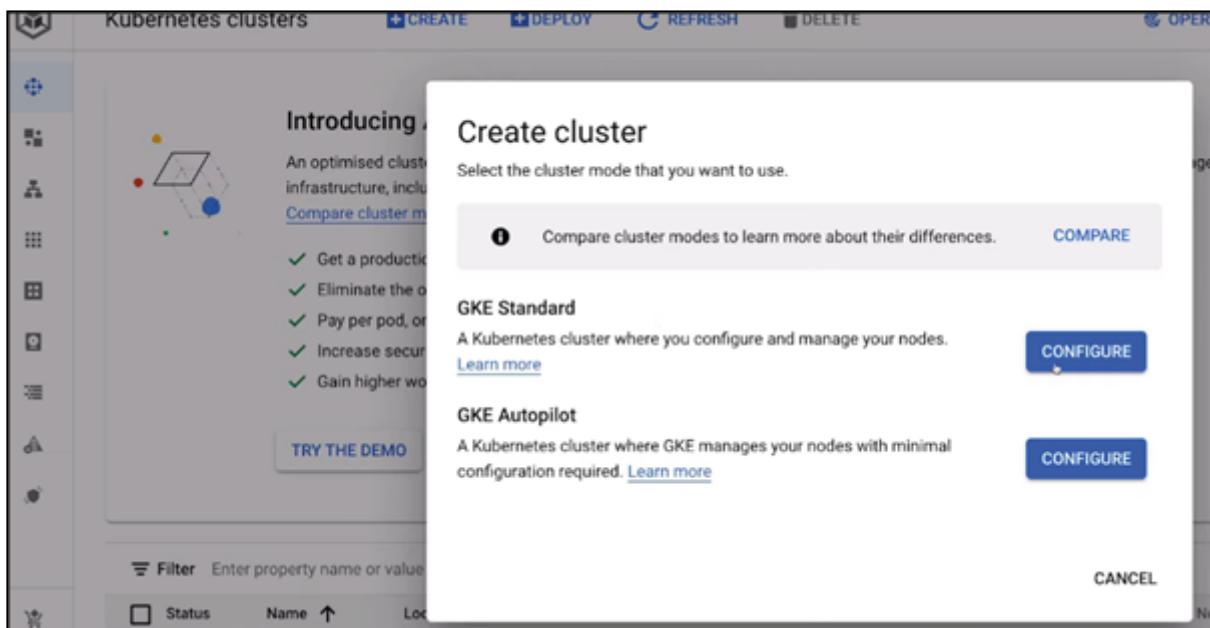


Figura 15.26 Creazione di un cluster GKE.

Da questa finestra, potremo inserire tutte le informazioni di base che servono per avviare un cluster: il suo nome, la regione in cui avviarlo e se renderlo *multi-zone* o *single-zone*. Al contrario della modalità Autopilot, qui abbiamo la possibilità di avviare un cluster su una singola zona di disponibilità, il che riduce i costi, ma ci pone anche di fronte a un'eventuale indisponibilità del sistema, qualora ci dovessero essere problemi. Sulla destra, sarà presente una finestra con la documentazione che riguarda tutte le proprietà del cluster che vengono elencate e il relativo dettaglio, come la scelta di un cluster privato piuttosto che pubblico, la VPC da utilizzare e i costi. Teniamo presente che esistono due tipi di cluster GKE standard. Il primo è quello pubblico, dove i *control-plane* sono accessibili pubblicamente e tutti i nodi worker hanno un'interfaccia pubblica collegata. Questo tipo di cluster è protetto utilizzando delle regole del firewall e inserendo nella *whitelist* solo gli intervalli IP approvati durante la definizione del cluster per consentire la comunicazione con le API, e questo per ridurre la superficie di attacco. La seconda opzione è quella di creare un cluster GKE privato: i *control-plane* e i nodi applicativi vengono distribuiti in un intervallo di rete VPC predefinito definito dall'utente. L'accesso ai componenti del cluster sarà completamente privato e, mentre i nodi *control-plane* saranno comunque gestiti da Google Cloud, quelli worker potranno essere controllati dall'utente. Seguendo entrambe le opzioni, sarebbe comunque bene creare prima una VPC dedicata piuttosto che utilizzare quella di default, dove dovremmo creare una sottorete per ospitare i nodi del cluster, e una separata per fare riferimento ai Pod e ai Services del cluster.

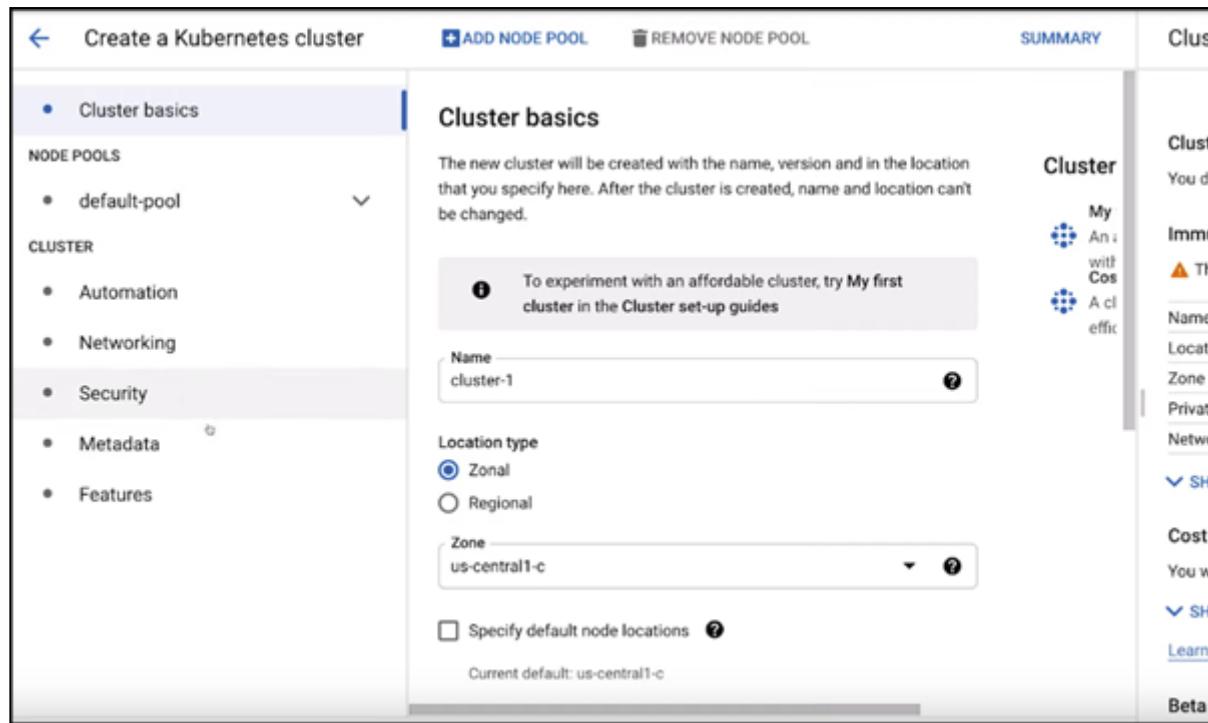


Figura 15.27 Informazioni di base del cluster.

Sul menu di sinistra ci sono diverse voci che permettono di definire il livello di dettaglio della configurazione del cluster: per esempio, cliccando sotto *Node pools*, possiamo definire il numero di nodi che faranno parte del *pool* (genericamente, dell'insieme) del cluster, così come l'opzione per abilitare la scalabilità, l'aggiornamento di sistema in maniera automatica o ancora il tipo di istanza utilizzata per creare i nodi.

The screenshot shows the 'Create a Kubernetes cluster' interface. On the left, there's a sidebar with sections like 'Cluster basics', 'NODE POOLS' (with 'default-pool' selected), and 'CLUSTER'. The main area is titled 'Node pool details' and contains fields for 'Name' (set to 'default-pool'), 'Node version' (set to '1.20.8-gke.900 (control plane version)'), 'Size' (set to 'Number of nodes * 3'), and an 'Enable auto-scaling' checkbox. A note at the bottom says 'Pod address range limits the maximum size of the cluster. [Learn more](#)'.

Figura 15.28 Dettaglio relativo al pool di nodi del cluster, come la versione di GKE da utilizzare, il numero, la gestione dello scaling e così via.

Nella sezione *Cluster* è possibile definire i dettagli generali sull'automazione di aspetti che riguardano l'interezza del cluster, come la manutenzione e gli aggiornamenti, il *provisioning* automatico dei nodi, la configurazione di uno scaling verticale automatico, e via dicendo; inoltre, è possibile definire dei parametri avanzati per la rete del cluster, così come l'*encryption* dei nodi e l'integrazione con i Google Groups per gestire l'RBAC dell'infrastruttura. Nel caso di esempio, è possibile lasciare questi parametri ai valori di default e procedere con la creazione. Saranno necessari alcuni minuti per completare l'installazione e per creare tutte le istanze con i dettagli specificati; quando il cluster sarà avviato, ci sarà un'icona verde con una spunta che indica la disponibilità del cluster nella pagina principale che mostra l'elenco completo:

<input type="checkbox"/>	<input checked="" type="checkbox"/>	standard-	us-central1-c	Standard	6	12	34.5 GB
--------------------------	-------------------------------------	-----------	---------------	----------	---	----	---------

Figura 15.29 Stato attuale del cluster GKE creato.

Quando l'installazione sarà completata, potremo scaricare il file `kubeconfig` tramite la console e aggiungerlo a quello locale, che ci permetterà di collegarci all'istanza del cluster e di utilizzare `kubectl` come di consueto:

Listato 15.10 Verifica dell'avvenuta connessione al cluster

```
kubectl cluster-info
>>>
Kubernetes control plane is running at https://xxx
GLBCDefaultBackend is running at https://xxx/api/v1/namespaces/kube-system/se rvi ce
s/default -http-backend : http/proxy
KubeDNS is running at https://xxx/api/v1/namespaces/kube-system/servi ces/kube - dns : dns/proxy
Metrics-server is running at https://xxx/api/v1/namespaces/kube- system/servi ces/https :
```

```
metri cs -server : /proxy  
...
```

A questo punto potremo installare le nostre applicazioni come di consueto, sfruttando le tante risorse che Kubernetes ci mette a disposizione: la differenza rispetto ad altre piattaforme è la possibilità di avere una dashboard integrata per gestire questi oggetti tramite la console di Google. Nel menu di sinistra della pagina principale del servizio *Kubernetes Engine* troviamo infatti la voce *Workloads*, attraverso la quale potremo gestire tutte le risorse applicative, come Deployment e Pod, insieme ai Services attraverso l'apposita sezione *Services & Ingress*, e anche i volumi, nella sezione *Storage*. Infatti, le dashboard “tradizionali” di Kubernetes installabili tramite un file YAML non fanno parte del setup di GKE, e sono anche deprecate.

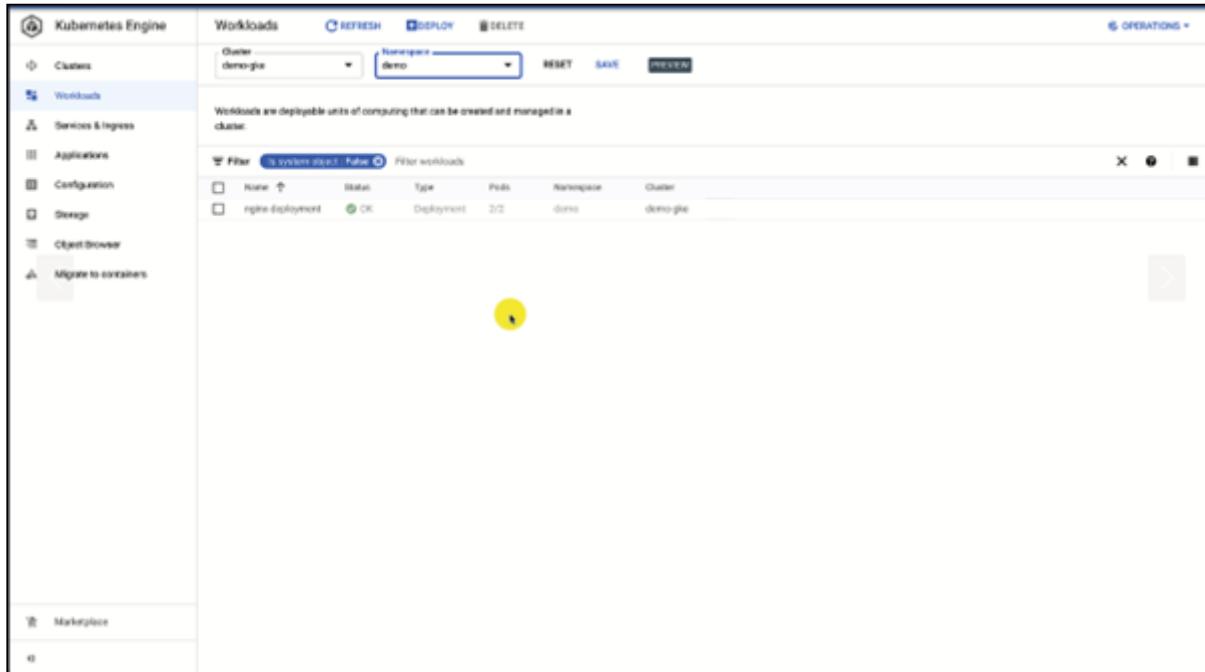


Figura 15.30 Riepilogo delle risorse applicative presenti nel cluster, tramite la dashboard di Google.

Mia-Platform

Apriamo una parentesi in questo capitolo dedicato al mondo dei servizi cloud in cui viene utilizzato Kubernetes per parlare di un prodotto tutto italiano: questa azienda informatica nata ufficialmente nel 2016 (lanciata nel 2013), negli ultimi anni si è distinta come una delle tech company più innovative del settore digitale, tanto da essersi guadagnata la fiducia di grandi company in vari ambiti, tra cui quello assicurativo, energetico ma anche healthcare. Perché parlare di Mia-Platform? Perché tra i prodotti principali vi è proprio il servizio omonimo, ossia una *platform builder cloud-native* che ti consente di creare e gestire la tua piattaforma digitale. In altre parole, si tratta di un insieme di strumenti che ti aiutano a creare, gestire e orchestrare dei microservizi, a governare delle API, a gestire i tuoi dati, a gestire i tuoi DevOps (*Platform Engineering*), a governare la tua piattaforma e a gestire il ciclo di vita del software. Il bello di questa piattaforma è che è stata creata da persone che sviluppano, per persone che sviluppano: anche l'interfaccia grafica è curata per essere sotto l'occhio della prospettiva di una persona che lavora in ambito Dev e Ops, ma anche di chi di tecnico ne sa un po' meno, e ha bisogno di avere sott'occhio tutti gli strumenti che gli servono per governare i propri servizi, migliorando la *Developer Experience*. Cosa c'entra con questo manuale? Ovviamente, lo strumento principale dietro a

questo lavoro è Kubernetes, ed è una piattaforma su cui hanno lavorato tante persone, soprattutto giovani, che hanno curato questo progetto in ogni dettaglio. Perché parlare di cloud vendor che vendono cari i propri servizi, e non di idee strepitose nate proprio su questo territorio?

Tramite la sua dashboard, Mia-Platform (repo ufficiale: <https://github.com/mia-platform>) consente di creare dei micro servizi modulari e riutilizzabili con un paio di click, partendo da template già pronti o utilizzando delle immagini Docker, e permette anche di gestire degli ambienti tramite variabili e configurazioni, astraendo la complessità dei microservizi. Questi potranno essere distribuiti in pochi clic su più provider cloud e in ambienti on-premise, mettendo a disposizione uno strumento utile anche per chi, come avviene all'interno dell'ecosistema di molto aziende, sul proprio portfolio di prodotti, deve fare *governance*: questo vuol dire poter gestire le proprie soluzioni e il loro rilascio all'interno di un'unica piattaforma, avendo a disposizione dashboard utili al loro monitoraggio. Mia-Platform semplifica la complessità del cloud, in modo che ci si possa concentrare sulla creazione dei prodotti, invece di perdere tempo a configurare i servizi. Così, tutti gli strumenti di cui possiamo aver bisogno sono racchiusi in un unico luogo, che ci dà una visione di insieme che va dall'infrastruttura, all'orchestrazione dei microservizi, fino ad arrivare alla gestione del ciclo di vita del software. Tutte le risorse che vengono create all'interno della piattaforma, come visto anche con alcuni degli approcci precedenti, sono basate su Kubernetes, di modo che queste risorse possano essere un giorno esportate e utilizzate su un ambiente esterno alla piattaforma stessa. L'integrazione con i principali strumenti adottati sul mercato è incredibile: tra questi, per citare alcuni nomi, c'è Node.js, MongoDB, PostgreSQL, Falco, e anche Kafka, Fluentd, Elasticsearch, OAuth e Let's Encrypt.

Per rendere però questa panoramica più efficace, procediamo con un caso d'uso pratico: vediamo il deploy di una semplice applicazione REST in Node.js, così da poter introdurre le tante funzionalità e i concetti dietro questa piattaforma. Intanto, dopo aver eseguito il login, la pagina principale mostra una panoramica dei progetti: questi corrispondono a una company specifica, o anche un sottogruppo di un'azienda che ha in esecuzione un set di progetti o di tecnologie all'interno dello stack di Mia-Platform. Selezionato il progetto, è possibile accedere alla console dedicata: l'aspetto interessante è che questa finestra è stata creata con lo scopo di far concentrare le persone che la utilizzano sul rendere esecutiva l'applicazione o il prodotto, senza dover perdere tempo a configurare l'infrastruttura; un approccio molto simile a quanto visto in precedenza con altri cloud provider.

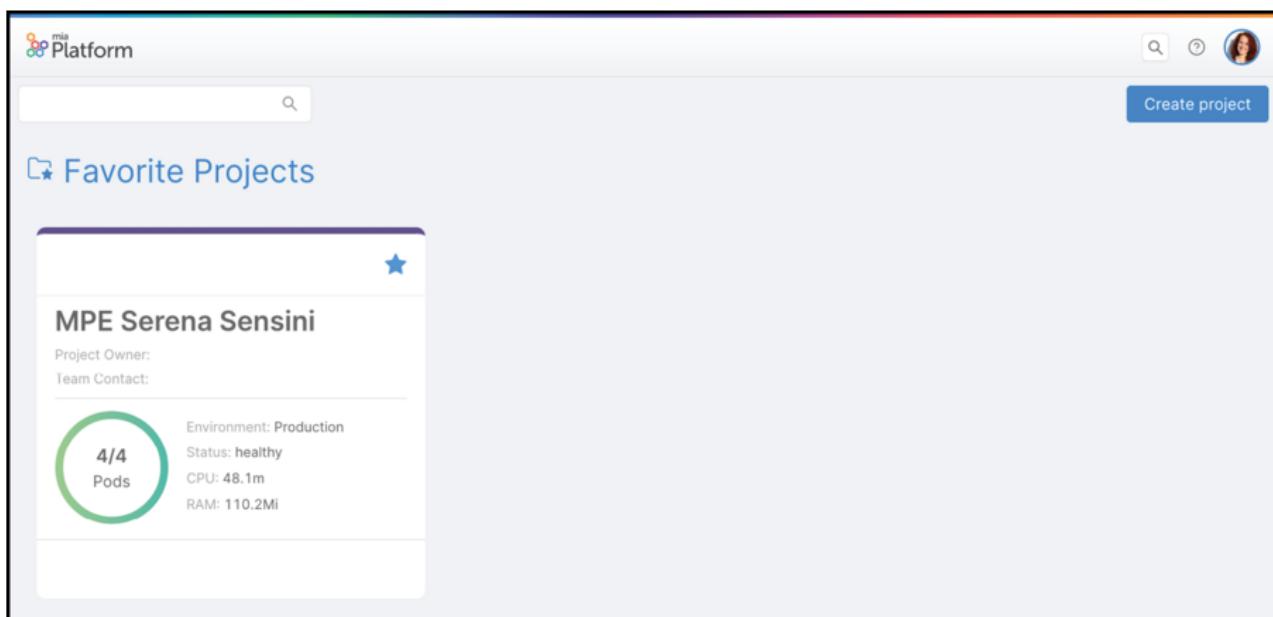


Figura 15.31 Pagina principale della console Mia-Platform.

Nella pagina principale è possibile scegliere quale ambiente utilizzare tramite le impostazioni: questi ambienti possono essere molteplici per rappresentare diverse condizioni di staging del software, ma anche contesti completamente diversi in cui l'applicazione viene eseguita. Da qui è possibile anche collegare la documentazione correlata a un determinato ambiente, così come la dashboard a cui accedere per iniziare a lavorare. Sempre tramite questa sezione, c'è la possibilità di definire delle variabili di ambiente globali: queste sono fondamentali per poter gestire le informazioni comuni al progetto o agli ambienti con cui abbiamo a che fare, come stringhe di connessione verso dei database, piuttosto che la configurazione della verbosità dei log prodotti dagli applicativi. Queste potranno essere riutilizzate in una seconda fase quando andremo a istanziare l'applicazione.

Name	ID	Documentation	CMS	Application	Cluster
Development	DEV	Go to Documentation	Go to CMS	-	internal- Edit
Production	PRODUCTION	Go to Documentation	Go to CMS	-	internal- Edit

Figura 15.32 Ambienti presenti all'interno del progetto.

A proposito di applicazione, torniamo allo scopo di questa sezione: tramite la dashboard principale, accediamo alla sezione relativa ai micro servizi premendo *Ctrl+k*, così da aprire il menu rapido che ci permette una ricerca veloce di quello che cerchiamo.

Project Settings Create, configure and manage your Kubernetes clusters	Design Configure CRUD, Microservices Marketplace, APIs, Proxies, Fast Data and Headless CMS	Deploy Deliver your configurations & code to Prod/No Prod environments	Dashboards Visualize API & Microservices Metrics
Runtime Monitor your Pods and the status of your microservices	Debug Debug microservices with Telepresence	Documentation View the documentation of APIs according to OpenAPI specification	Marketplace In the marketplace, you will find different templates to accelerate software development

Figura 15.33 Console con voci rapide.



Figura 15.34 Menu con ricerca.

A questo punto, abbiamo di fronte un menu molto più completo di quello iniziale, ma che ci mette a portata di mano tutto ciò che ci serve per avviare l'immagine Node.js. In effetti, l'elenco dei micro servizi consente la creazione di una nuova applicazione a partire da un'immagine presente su DockerHub o su altri registry come Nexus (a seconda del tipo di license), o anche di sfruttare una delle opzioni presenti nel marketplace. In effetti, uno dei punti di forza di questa piattaforma è proprio questo catalogo, distribuito in modalità self-service, che contiene diverse tipologie di tecnologie sotto quattro diverse forme: i *template*, ossia dei boilerplate per avere un modello rapido da cui partire per avviare la propria istanza e personalizzarla, gli *example*, per dei casi d'uso veloci e già pronti per l'utilizzo, i *plugin*, ossia immagini pronte e finite da utilizzare, e infine le *application*, che sono un insieme di plugin con lo scopo di fornire un insieme di funzionalità specifico. Un esempio di quest'ultimo caso è il *Payment Gateway*, ossia un gestore dei metodi di pagamento che è possibile installare, configurare e utilizzare come wrapper del processo di pagamento con diversi provider, tra cui Satispay, Unicredit, Scalapay e Stripe.

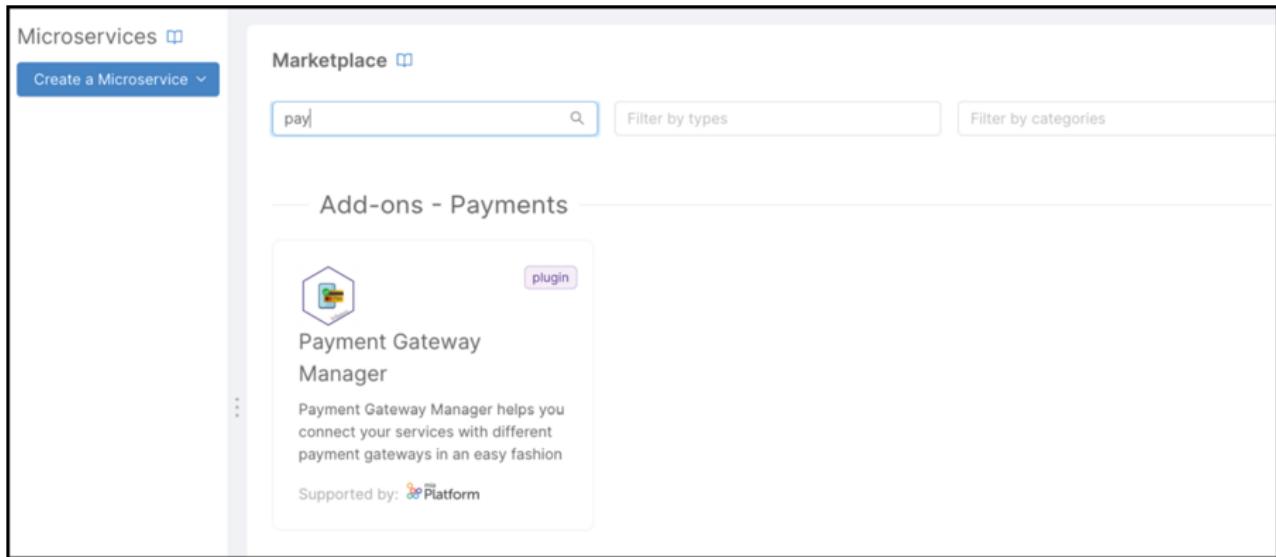


Figura 15.35 Marketplace per la ricerca di applicazioni, template, plugin o example già pronti.

Nel nostro esempio, cerchiamo nel Marketplace l'applicazione di esempio per Node.js e iniziamo a compilare i diversi campi: il nome host interno alla piattaforma, eventualmente la descrizione dell'applicazione, e poi il nome del repository e dell'immagine Docker che verrà creata. In questo caso, è utile notare come, a ogni microservizio, corrisponda la creazione di un repository Git: questo permette di salvare le risorse e la loro definizione su uno strumento esterno, e soprattutto di

versionarle; tutto ciò che andremo a creare avrà una sua “storia”, così da poter lavorare con degli oggetti che evolvono nel tempo e che ci permettono anche di tornare indietro in caso di errori. In alto a destra nel dettaglio del microservizio c’è un link che rimanda al repository Git appena creato, a cui potremo accedere per visualizzare le risorse appena create grazie all’avvio di questo esempio; questo perché, se avessimo bisogno di lavorare con un prodotto che non è un semplice esempio, sarà necessario poter configurare i suoi attributi con uno strumento che è più familiare anche per le persone che sviluppano.

Una volta creato il microservizio, questo sarà presente nell’elenco visto in precedenza e sarà possibile gestirne il dettaglio: già tramite questa visualizzazione, torniamo in una zona di comfort con degli oggetti e delle terminologie che ci sono familiari, come le informazioni circa *request* e *limit* in termini di CPU e memoria, o anche *liveness* e *readiness probe*. Inoltre, all’interno del dettaglio di questo applicativo, abbiamo anche la possibilità di gestire le variabili di ambiente specifiche per quel contesto, il cui valore può anche essere recuperato da quelle globali.

Create a new microservice from Example [⊕](#)

Node.js HelloWorld Microservice Example
Supported by: Platform [View documentation](#)

General details

* Name (Internal Hostname)
nodejs-helloworld-microservice-example

Description
Example of a simple Node.js application.
It contains example of tests too.

* Git repository owner
Clients / External PaaS MPE / Mpe Serena Sensini / Services

* Git repository name
nodejs-helloworld-microservice-example

* Docker Image Name
mpe-serena-sensini/nodejs-helloworld-microservice-example

Create

Figura 15.36 Interfaccia per la creazione di un microservice basato su Node.js.

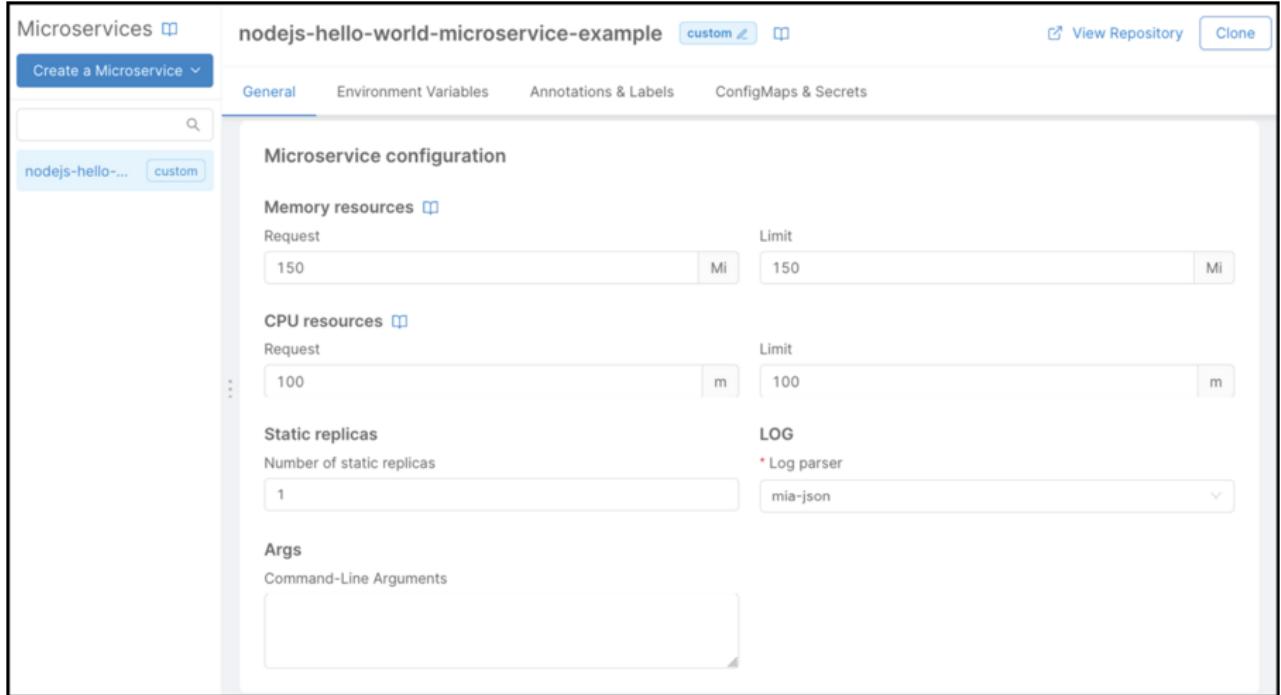


Figura 15.37 Dettaglio delle proprietà relative alle risorse del microservizio creato.

Key	Value	Description
LOG_LEVEL	{{LOG_LEVEL}}	
MICROSERVICE_GATEWAY_SERVICE	microservice-gateway	
TRUSTED_PROXIES	10.0.0.0/8,172.16.0.0/12,192.168.0...	
HTTP_PORT	3000	
USERID_HEADER_KEY	miauserid	
GROUPS_HEADER_KEY	miausergroups	
CLIENTTYPE_HEADER_KEY	client-type	
BACKOFFICE_HEADER_KEY	isbackoffice	
USER_PROPERTIES_HEADER_KEY	miauserproperties	

Figura 15.38 Dettaglio sulle variabili di ambiente configurate per il microservizio.

Ma che cosa rappresenta questo microservizio? Nient'altro che un Pod che viene avviato all'interno del cluster. Salviamo il lavoro fatto finora facendo clic in alto a destra sul nome del branch su cui il lavoro attuale verrà salvato tramite un commit, torniamo nel menu principale e selezioniamo la voce *Deploy*: da qui possiamo recuperare le informazioni relative all'ultimo rilascio sull'ambiente su cui

stiamo lavorando. Questa interfaccia ci permette di selezionare il commit fatto e di avviare l'esecuzione dell'applicazione Node.js. In questa sezione, possiamo anche decidere se eseguire quello che si chiama *Smart deploy*: in altre parole, è possibile richiedere una verifica del delta (della differenza) tra ciò che è eventualmente in esecuzione e ciò che vogliamo eseguire, così da applicare solo le modifiche. Questa funzionalità segue le strategie di Kubernetes sul deploy, e di default utilizza quella che si chiama *BestEffort*: nel mondo Kubernetes, e più in generale dei container, questo vuol dire che se parametri come *request* e *limit* non sono specificati, i container verranno creati con una priorità bassa. Nel caso di Mia-Platform, questo riguarda le modalità con cui queste modifiche tra una versione e l'altra verranno applicate: nel caso di conflitti tra le risorse, si lascerà quanto già presente.

The screenshot shows the deployment interface for a microservice named "myservice".

- Latest deployed version:**
 - Branch: master (commit c81a5873)
 - Completed at: Mar 06, 2023, 6:30 PM
 - Made by: Serena Sensini
 - Type: Smart Deploy
 - Duration: 12 seconds
 - Pipeline: View Log
- Select branch or tag:**
 - master
 - Compare with last successfully deployed version: master ... c81a5873
 - Last 7 commits:
 - master ... 38f07c5a commit
 - master ... 4cd12741 commit
 - master ... 026dda5b Initial commit
- Deploy details:**
 - Smart deploy (radio button selected)
 - Always release services not following semantic versioning (checkbox)
 - Table:

Name (Internal Hostname)	Deploy Outcome	Running version	New version	Last build	More info
myservice (custom)	No Update	latest	latest	success	View build history View changelog

At the bottom right, there are buttons for "Deploying master on Development" and "Smart Deploy (⌘+F)".

Figura 15.39 Deploy del microservizio.

Al deploy di ogni risorsa nel cluster corrisponde una pipeline, che è possibile visualizzare sia cliccando sulla finestra che si aprirà, sia tramite il menu principale (ricorda il trucco del comando *Ctrl+k*) nella voce relativa all'ambiente di esecuzione (alias *Runtime*): in questa sezione troviamo tutte le risorse Kubernetes con cui abbiamo a che fare durante il ciclo di vita dell'applicativo, e sempre da qui è possibile accedere ai log del container, agli eventi del Pod, alle impostazioni e alla documentazione dell'applicazione. Come per una qualsiasi applicazione in esecuzione su un cluster Kubernetes, abbiamo anche la possibilità di gestirne le informazioni di configurazione tramite risorse come ConfigMaps e Secrets, che sono dedicate per quelle specifiche risorse: il lavoro fatto per integrare l'ambiente Kubernetes ed espanderlo all'ennesima potenza con un catalogo di soluzioni pronte all'uso (tramite il Marketplace), oltre alla gestione degli utenti e dei relativi permessi, è quanto di più simile a una piattaforma di gestione di un sistema PaaS ci si possa aspettare.

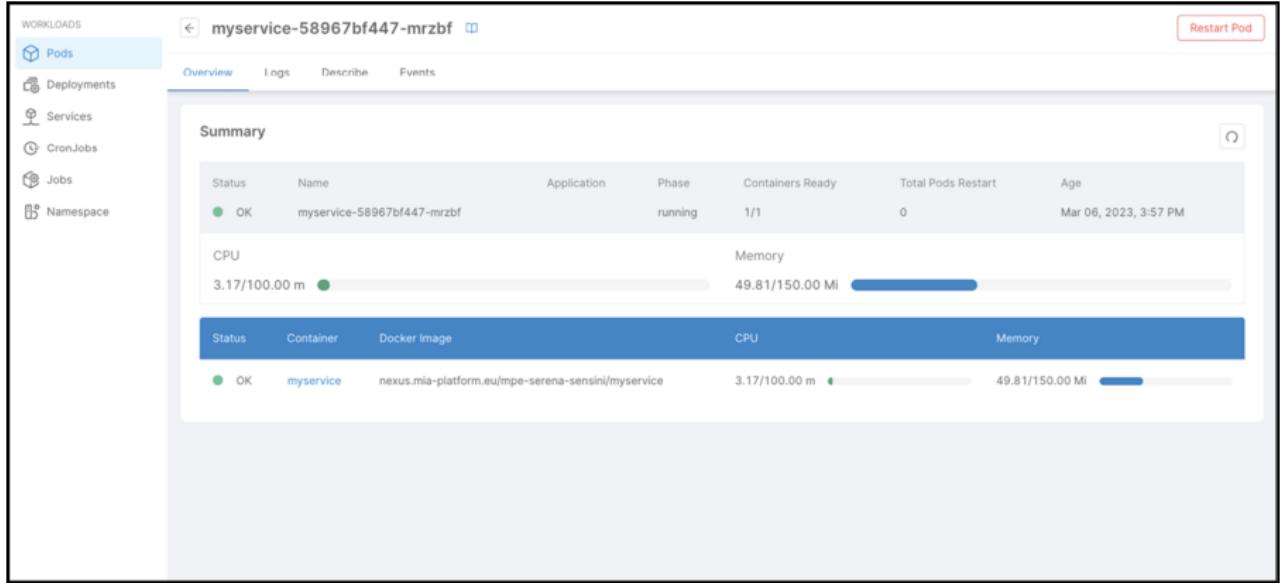


Figura 15.40 Dashboard dedicata alle risorse applicative in esecuzione sul cluster Kubernetes e al relativo dettaglio.

Ora, per completare l'esempio ed esporre l'applicazione, dovremo creare un *API gateway* che ci permetta di rendere raggiungibile il servizio esternamente, insieme a un endpoint: questo componente può essere sempre creato tramite il Marketplace, cercando il microservizio corrispondente e lasciando (per questo esempio) i dati di default. Così come avvenuto per l'esempio di Node.js, verrà creato un Pod e un Service corrispondente, che saranno visibili nella dashboard relativa al *Runtime*.

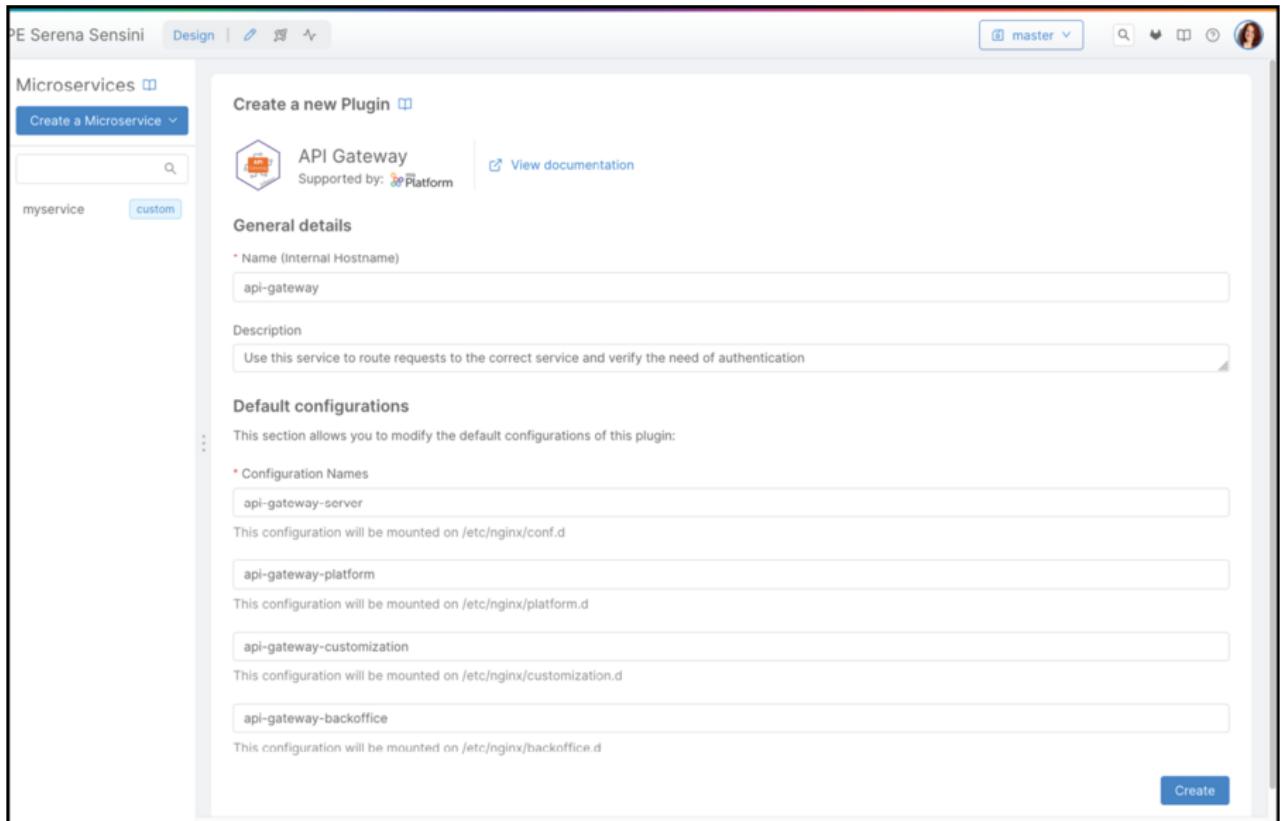


Figura 15.41 Configurazione del plugin API Gateway per la corretta raggiungibilità del servizio creato dall'esterno.

Lo step successivo prevede la creazione di un endpoint che si collega al microservizio creato; tramite la dashboard di *Design*, clicchiamo sulla voce *Endpoints* del menu, ne creiamo uno nuovo e, sempre tramite il pulsante in alto a destra con il riferimento al branch, salviamo il lavoro fatto e procediamo con il deploy.

The screenshot shows a 'Create new endpoint' form. It has two main sections: 'Base path:' containing '/app' and 'Type:' set to 'Microservice'. Below these are 'Microservice:' (set to 'myservice') and 'Description:' (containing 'My service endpoint'). A 'Create' button is at the bottom right.

Figura 15.42 Interfaccia per la definizione di un Endpoint.

L'applicazione è pronta per essere testata, e non solo: grazie alle applicazioni disponibili nella piattaforma, possiamo anche sfruttare strumenti come Swagger per generare della documentazione sulla base del servizio appena creato: come visibile nella figura successiva, creando un'applicazione *API Documentation Aggregator* (sempre disponibile nel Marketplace), potremo mettere in piedi l'infrastruttura che raccoglie e rende visibile la documentazione per tutti i microservizi che abbiamo nel progetto:

The screenshot shows a 'Create a new Application' form for the 'API Documentation Aggregator'. On the left, there's a sidebar with 'Applications' and a 'Create new Application' button. The main area shows the aggregator application with its icon, supported by the Platform, and a 'View documentation' link. The process is at step 1, 'Application Info', with fields for 'Application name:' (set to 'api-documentation-aggregator') and 'Description:' (containing a note about aggregating APIs). A 'Next' button is at the bottom right.

Figura 15.43 Creazione di un'applicazione.

Dopo che i Pod di Swagger saranno in esecuzione (possiamo verificarlo sempre tramite la dashboard di *Runtime*), potremo accedere alla documentazione tramite il link disponibile nella schermata relativa alle impostazioni degli ambienti: questa ci mostrerà la documentazione generata a partire dall'API, dove potremo leggere i dettagli relativi alle chiamate disponibili alle request che possiamo eseguire verso l'applicazione, nonché trovare un box che ci permette di testare le chiamate direttamente nella stessa finestra.

Name	ID	Documentation	CMS	Application	Cluster
Development	DEV	Go to Documentation	Go to CMS	-	internal-paas-mpe-noprod
Production	PRODUCTION	Go to Documentation	Go to CMS	-	internal-paas-mpe-noprod

Figura 15.44 Link per la documentazione recuperabile tramite il riepilogo degli ambienti.

Figura 15.45 Documentazione generata tramite Swagger a partire dalla definizione del microservizio.

Nel caso d'uso appena esaminato abbiamo toccato forse la punta dell'iceberg delle funzionalità di questa piattaforma: infatti, non è da tralasciare la possibilità di visualizzare i dati aggregati del progetto con i quali realizzare dei grafici di rappresentazione delle risorse utilizzate dalle entità in esecuzione a un livello molto specifico di dettaglio attraverso le metriche esposte dalle API dei microservizi. Grazie all'integrazione con strumenti come Grafana, Mia-Platform consente di disporre di dashboard personalizzabili e avanzate per mostrare le informazioni del progetto in maniera visuale, anche grazie alle metriche mostrate attraverso dei grafici che descrivono lo stato delle applicazioni in esecuzione, con i relativi consumi in termini di risorse. Il potenziale è tutto da scoprire, e l'augurio è che nascano (e soprattutto, continuino a crescere) sempre più prodotti con un team che ha una visione così ad ampio raggio sul panorama IT e sulle tecnologie esistenti!

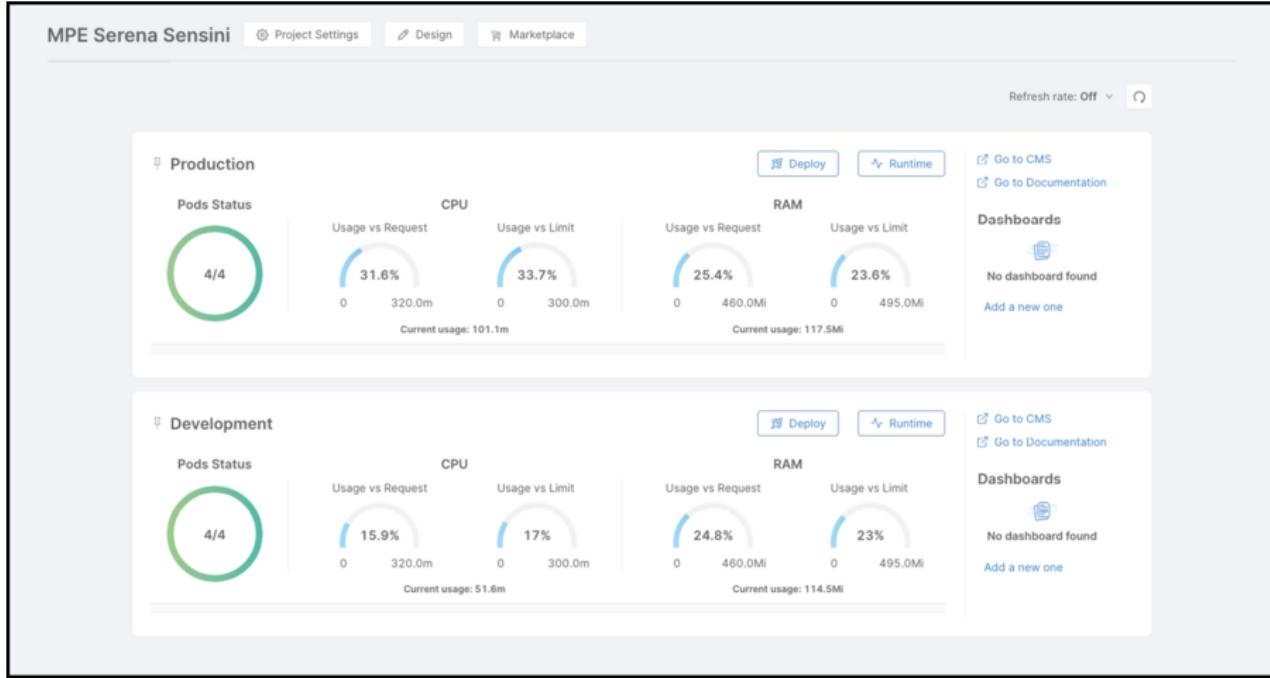


Figura 15.46 Dashboard di riepilogo delle metriche relative ai progetti presenti e alle risorse utilizzate, con i link utili per mostrare informazioni sul deploy degli oggetti e la loro documentazione.

Che cosa abbiamo imparato

- Kubernetes non lavora solo su infrastrutture on-premise, ma viene anche fornito come servizio di molti cloud provider; abbiamo visto quali sono le principali soluzioni offerte da questi, e quando valutarle.
- AWS offre diversi tipo di servizi per eseguire dei container: ECS permette di avviare le proprie istanze in maniera autonoma, lavorando grazie a delle istanze EC2, mentre EKS è quello che permette di gestire un cluster in maniera molto simile a quanto visto in questo manuale.
- GKE è la soluzione offerta da Google, che esiste in due versioni: una completamente gestita, il cui costo è legato al numero di Pod, e una standard, dove gran parte della gestione è demandata all'utente; abbiamo anche visto come creare un cluster e utilizzarlo per collegarci e installare le nostre applicazioni.
- Per il mondo Azure, abbiamo analizzato il funzionamento di AKS e sfruttato il servizio di deploy automatizzato per creare una piccola pipeline che fosse in grado di avviare la nostra applicazione in Python, a partire da un semplice Dockerfile.
- Infine, abbiamo dato uno sguardo a un prodotto tutto italiano, Mia-Platform: una soluzione che permette di fare governance dei propri prodotti attraverso un'unica dashboard, che offre un catalogo di servizi molto vasto, e che ha come motore portante Kubernetes.

Risorse utili

Kubernetes non è solo una tecnologia, ma è un mondo che ha una community molto attiva, e che sta aumentando la sua portata di giorno in giorno. Per questo, attraverso queste poche pagine, raccogliamo alcune risorse utili per certificarsi nel mondo Kubernetes, ma anche per scoprire di più il suo dominio grazie a blog ed eventi del settore che ne aumentano la risonanza. Questo elenco è presente anche su GitHub e può essere aggiornato costantemente con le ultime novità.

Contribuisci

Se conosci qualche evento nella tua zona, o qualche blog che parla di questa tematica in italiano, sfrutta il repository per fare una *pull request* e aggiornare l'elenco delle risorse a disposizione della community!

Certificazioni

Kubernetes sta crescendo in popolarità giorno dopo giorno: un sondaggio condotto dalla Cloud Native Computing Foundation (CNCF) ha riportato che il 96% degli intervistati utilizza o valuta Kubernetes. Esistono nel campo diverse certificazioni sul tema, che possono tornarci utili anche nel mondo del lavoro: nulla è paragonabile all'esperienza, questo è chiaro, ma sicuramente sono una buona palestra per allenare le competenze acquisite. Per questo, vengono riportate quelle più comuni nel panorama, e quali requisiti sono necessari per superarle.

Kubernetes and Cloud Native Associate (KCNA)

KCNA è consigliato a studenti, persone responsabili del settore IT e laureati/e in ingegneria che necessitano di una conoscenza di base dell'ecosistema nativo del cloud. L'esame si concentra su Kubernetes e sugli strumenti correlati nel panorama. Questo esame è molto facile rispetto a quelli che vedremo in seguito e serve proprio a certificare che si è in grado di coprire a grandi linee tutte le tematiche inerenti il contesto Kubernetes e dimostrare quindi una conoscenza generale del suo funzionamento. È un esame con 60 domande a scelta multipla da completare in 90 minuti. Gli argomenti trattati nell'esame KCNA includono:

- Kubernetes Fundamentals
 - Risorse Kubernetes
 - Architettura
 - Kubernetes API
 - Container
 - Scheduling
- Container Orchestration
 - Le basi dell'orchestrazione
 - Ambiente di esecuzione
 - Sicurezza
 - Networking

- Rete Mesh
- Storage
- Cloud Native Architecture
 - Autoscaling
 - Serverless
 - Community e Governance
 - Ruoli e utenti
 - Open Standard
- Monitoraggio
 - Telemetry & monitoring
 - >Prometheus
 - Cost Management
- Cloud Native Application Delivery
 - Applicazione DevOps

Il minimo punteggio per il superamento dell'esame KCNA è del 65%. L'esame dura circa 90 minuti e ha una validità di 3 anni. La documentazione ufficiale è disponibile all'indirizzo

<https://training.linuxfoundation.org/certification/kubernetes-cloud-native-associate/>.

Certified Kubernetes Administrator (CKA)

La certificazione CKA è molto ricercata quando si tratta di DevOps Engineer Jobs. A differenza di altre certificazioni, in cui gli esami hanno domande a scelta multipla, la CKA prevede un esame pratico, in cui è necessario risolvere delle attività attraverso Kubernetes e risolvere i problemi del cluster tramite la riga di comando. Questa certificazione è rivolta alle persone che lavorano in ambito amministrazione di sistema e (in parte) DevOps coinvolte nei progetti che sfruttano Kubernetes. Il programma Certified Kubernetes Administrator garantisce che le persone che superano questa certificazione abbiano le capacità, le conoscenze e le competenze per adempiere a tutte le operazioni che riguardano gli l'amministrazione di un'infrastruttura Kubernetes. L'esame CKA si concentra sui seguenti domini e competenze, ognuno con una percentuale diversa di domande (che varia nel tempo a seconda degli aggiornamenti della documentazione):

- storage;
- troubleshooting;
- risorse applicative e gestione della loro schedulazione;
- architettura, installazione e configurazione del cluster Kubernetes;
- networking.

L'esame dura circa 2 ore e ha una validità di 3 anni. La documentazione ufficiale è disponibile all'indirizzo <https://training.linuxfoundation.org/certification/certified-kubernetes-administrator-cka/>.

Certified Kubernetes Application Developer (CKAD)

Questo esame è rivolto alle persone che sviluppano e che, più in generale, si occupano dell'ambito DevOps; la certificazione CKAD si concentra maggiormente sugli aspetti applicativi di Kubernetes. Se, quindi, ti occupi di sviluppo o di rilascio delle applicazioni e vuoi mostrare la tua capacità di avviare e gestire applicazioni su Kubernetes, questa è la certificazione perfetta. L'esame serve a certificare che i candidati possono progettare, creare e distribuire applicazioni cloud-native per Kubernetes. È anche un

esame pratico basato sulle prestazioni in cui è necessario risolvere le attività Kubernetes relative al rilascio di applicazioni, per cui ci sono diversi domini e competenze da attenzionare:

- progettazione e creazione di applicazioni;
- rilascio dell'applicazione su Kubernetes;
- monitoraggio e manutenzione dell'applicazione;
- configurazione di ambiente dell'applicazione, gestione e sicurezza;
- networking.

L'esame dura circa 2 ore e ha una validità di 3 anni. La documentazione ufficiale è disponibile all'indirizzo <https://training.linuxfoundation.org/certification/certified-kubernetes-application-developer-ckad/>.

Certified Kubernetes Security Specialist (CKS)

L'esame CKS è consigliato alle persone coinvolte nella sicurezza cloud-native. La certificazione CKS si concentra maggiormente sulla sicurezza di Kubernetes: quando si tratta di ambienti containerizzati, la sicurezza gioca un ruolo chiave. Dalla creazione di un'immagine del container alla sua distribuzione su Kubernetes, è necessario applicare le best practice in termini di sicurezza per ridurre la superficie di attacco dell'infrastruttura. Questa certificazione copre gli aspetti di sicurezza dalla creazione di immagini alla loro distribuzione. Avrai bisogno di saper dimostrare l'utilizzo e la conoscenza di strumenti di sicurezza all'interno del contesto CNCF come Falco e Trivy per prepararti a questo esame: infatti, il programma CKS garantisce che una persona che consegua questa certificazione abbia le capacità, le conoscenze e le competenze su un'ampia gamma di best practice per proteggere le applicazioni basate su container e le piattaforme Kubernetes durante la creazione, la distribuzione e la loro esecuzione. Unico prerequisito: per sostenere questo esame è richiesta la certificazione CKA.

L'esame CKS copre i seguenti argomenti:

- configurazione del cluster;
- hardening del cluster;
- hardening del sistema;
- riduzione al minimo le vulnerabilità dei microservizi;
- sicurezza dell'intera architettura e dei flussi di lavoro;
- monitoraggio, gestione dei log e sicurezza dell'ambiente di esecuzione.

L'esame dura circa 2 ore e ha una validità di 2 anni, al contrario delle altre. La documentazione ufficiale è disponibile all'indirizzo <https://training.linuxfoundation.org/certification/certified-kubernetes-security-specialist/>.

Documentazione utile, e non solo

A oggi è possibile trovare diverso materiale in rete sul tema Kubernetes, che va oltre gli argomenti trattati in questo manuale. Per questo, creiamo una sezione utile ad alcune delle risorse più valide in italiano che parlano del tema, e che esplorano il suo dominio attraverso tutorial, video e guide.

Blog

- *TheRedCode*: blog che tratta il mondo della tecnologia a 360 gradi, parlando di intelligenza artificiale, sviluppo e design, ma anche di strumenti come Docker, Kubernetes e OpenShift.
- <https://theredcode.it>

- *MiaMammaUsaLinux*: blog formato da una community di esperti/e che tratta i temi più caldi del momento, e che racconta anche l'evoluzione di Kubernetes e del suo progetto da un punto di vista non solo pratico.
- <https://www.miamammausalinux.org>
- *SparkFabrik*: blog creato a partire dall'azienda stessa, parla di cloud e DevOps e partecipa in modo molto attivo nei principali eventi del settore, tra cui quelli dedicati a Kubernetes e al mondo dell'open source.
- <https://blog.sparkfabrik.com/>
- *amerlin*: blog a cura di Andrea Merlin, tratta diversi casi d'uso su Kubernetes, come la gestione della sua sicurezza.
- <https://amerlin.keantex.com/tag/kubernetes/>
- *VincenzoRacca*: blog che si occupa di sviluppo e dockerizzazione, con un focus su ambiente Java e derivati, sia in inglese che in italiano.
- <https://www.vincenzoracca.com/blog/container/kubernetes>

Canali YouTube

- Canale EmmeCiLab, a cura di Mauro Cicolella, esperto informatico e freelance che parla dell'informatica e del suo "dietro le quinte" a parole chiare e con tutorial estremamente chiari (<https://www.youtube.com/@EmmeCiLab>).
- ItalianCoders DevTalks (<https://www.youtube.com/@ItalianCoders>).
- Michele Ferracin, canale che parla di "cose sull'informatica", in maniera molto semplice (<https://www.youtube.com/@MicheleFerracin/>).

Eventi

- *Kubernetes Community Days*: evento annuale che si svolta in diversi paesi europei, tra cui l'Italia (<https://community.cncf.io/kubernetes-community-days/about-kcd/>).
- *Kubecon*: evento annuale che si tiene in Europa a tema Kubernetes e organizzato dalla CNCF (<https://events.linuxfoundation.org/kubecon-cloudnativecon-europe/>).

Istruzioni di base

Gestione del cluster

Informazioni sul cluster

```
# Informazioni sui nodi master e sui servizi disponibili nel cluster  
kubectl cluster-info  
# Versione di Kubernetes  
kubectl version
```

Configurazione del cluster

```
# Descrizione della configurazione presente nel file kubeconfig, come cluster e utenti  
kubectl config view
```

```
# Elenco degli utenti presenti nel file kubeconfig  
kubectl config view -o jsonpath='{.users[*].name}'
```

```
# Cluster corrente  
kubectl config current-context
```

```
# Elenco dei cluster  
kubectl config get-contexts
```

```
# Selezione del cluster  
kubectl config use-context CLUSTER
```

```
# Configurazione del namespace di lavoro  
kubectl config set-context --current --namespace=NAMESPACE
```

Namespace

Gestione dei namespace

```
# Creazione di un nuovo namespace  
kubectl create namespace NAMESPACE_NAME
```

```
# Descrizione dello stato del namespace  
kubectl describe namespace NAMESPACE_NAME
```

```
# Rimozione di un namespace e delle risorse al suo interno  
kubectl delete namespace NAMESPACE_NAME - Delete a namespace.
```

```
# Modifica della definizione del namespace  
kubectl edit namespace NAMESPACE_NAME
```

```
# Risorse utilizzate dal namespace, in termini di CPU, memoria e storage  
kubectl top namespace NAMESPACE_NAME
```

Risorse

Elenco di tutte le risorse di tutti i namespace

```
kubectl get all --all-namespaces
```

Abbreviazioni

```
# Elenco di tutti i Pod  
kubectl get po
```

```
# Elenco dei ReplicaSets  
kubectl get rs
```

```
# Elenco dei ReplicationControllers  
kubectl get rc
```

```
# Elenco dei Deployment  
kubectl get deploy
```

```
# Elenco degli StatefulSets  
kubectl get sts  
# Elenco dei DaemonSet  
kubectl get ds
```

```
# Elenco dei Service  
kubectl get svc
```

```
# Elenco delle ConfigMaps  
kubectl get cm
```

```
# Elenco dei CronJobs  
kubectl get cj
```

```
# Elenco dei nodi  
kubectl get no
```

```
# Elenco dei namespace  
kubectl get ns
```

```
# Elenco delle PersistentVolumeClaim  
kubectl get pvc
```

```
# Elenco delle ResourceQuotas  
kubectl get quota
```

```
# Elenco dei ServiceAccount  
kubectl get sa
```

```
# Elenco dei Pod e dei Services  
kubectl get po, svc
```

Gestire le risorse

```
# Creazione di una risorsa a partire da un file YAML  
kubectl apply -f FILENAME
```

```
# Modificare lo YAML di definizione  
kubectl edit daemonset DAEMONSET_NAME
```

```
# Cancellare una risorsa  
kubectl delete deploy DEPLOYMENT_NAME
```

```
# Mostrare le informazioni della risorsa  
kubectl describe po POD -n NAMESPACE
```

```
# Informazioni su una risorsa
```

```

kubectl describe pod <pod_name>

# Aprire una shell interattiva sul container di un Pod
kubectl exec -it <pod_name> -c <container_name> /bin/sh

# Mostrare le risorse utilizzate dai Pod
kubectl top pod

# Aggiungere una label ad un Pod
kubectl label pods <pod_name> new-label=<label name>

# Elenco delle label e dei Pod
kubectl get pods --show-labels

# Scalare le repliche di uno StatefulSet
kubectl scale --replicas=3 sts/STATEFULSET_NAME

# Scalare più risorse contemporaneamente
kubectl scale --replicas=5 rc/REPLICATIONCONTROLLER_NAME deploy/DEPLOYMENT_NAME

# Rimuovere uno StatefulSet, senza rimuovere i Pod da esso gestiti
kubectl delete statefulset/[stateful_set_name] --cascade=false

```

Storico delle versioni

```

# Eseguire il rollout di una risorsa
kubectl rollout statefulset STATEFULSET_NAME

# Rollback di una risorsa alla versione precedente
kubectl rollout undo deploy/DEPLOYMENT_NAME

# Mostra le versioni della risorsa
kubectl rollout history deploy/DEPLOYMENT_NAME

# Metti in pausa il rollout
kubectl rollout history deploy/DEPLOYMENT_NAME

# Riavvio del rollout di una risorsa tramite selettore
kubectl rollout restart ds --selector=app=nginx

```

Rete

Gestire la comunicazione verso l'esterno

```

# Esporre una risorsa con un Service
kubectl expose deployment DEPLOYMENT_NAME

# Creare un Service per il Deployment che espone la porta 80 del Service e usa la porta 8000 del container
kubectl expose deployment DEPLOYMENT_NAME --port=80 --target-port=8000

# Creare un port-forwarding per mettere in comunicazione una porta locale con quella del Pod
kubectl port-forward POD_NAME LOCAL_PORT:POD_PORT

# Creare un port-forwarding sulla porta 8443 locale verso la target port del Service chiamata https del Service myservice
kubectl port-forward service/myservice 8443:https

```

Logs

Gestione

```
# Logs delle ultime 8 ore di un Pod  
kubectl logs --since=8h POD_NAME  
  
# Ultime 50 righe dei logs del Pod  
kubectl logs --tail=50 POD_NAME  
  
# Stampa dei log in tempo reale del Pod  
kubectl logs -f POD_NAME  
  
# Log del Pod precedente in stato Failed  
kubectl logs --previous POD_NAME  
  
# Log di tutti i container del Pod  
kubectl logs POD_NAME --all-containers=true
```

Eventi

Gestione

```
# Elenco degli eventi per tutti i namespaces  
kubectl get events --all-namespaces  
  
# Elenco degli eventi filtrati per tipologia  
kubectl get events --field-selector type=Warning  
# Elenco degli eventi in ordine cronologico per il namespace test  
kubectl get events --sort-by=.metadata.creationTimestamp -n test  
  
# Elenco degli eventi di tutte le risorse, tranne i Pod  
kubectl get events --field-selector involvedObject.kind!=Pod  
  
# Elenco degli eventi di uno specifico nodo  
kubectl get events --field-selector involvedObject.kind=Node, involvedObject.name=NODE
```

Nodi

Gestione

```
# Rimuovi nodo  
kubectl delete node NODE_NAME - Delete a node or multiple nodes.  
  
# Mostra le risorse utilizzate  
kubectl top node NODE_NAME  
  
# Mostra i Pod del nodo  
kubectl get pods -o wide | grep NODE_NAME  
  
# Aggiungi un'annotazione al nodo  
kubectl annotate node NODE_NAME  
  
# Aggiungi label al nodo  
kubectl label node NODE_NAME
```

Taint e toleration

```
# Assegnazione di una taint al nodo  
kubectl taint node NODE_NAME
```

Troubleshooting

```
# Configura il nodo come unschedulable
kubectl cordon node NODE_NAME

# Configura il nodo come schedulable
kubectl uncordon node NODE_NAME

# Esecuzione di una drain del nodo per rimuovere i Pod dal nodo e pianificarli su un altro
kubectl drain node NODE_NAME
```

YAML

Nozioni di base

È difficile sfuggirgli se stai lavorando con tecnologie come Kubernetes e OpenShift: YAML, che sta per *Yet Another Markup Language* o *YAML Ain't Markup Language* (a seconda della persona a cui lo chiedi) è un formato basato su testo leggibile dall'essere umano per specificare informazioni sul tipo di configurazione.

Per esempio, in questa breve guida, selezioneremo le definizioni YAML per creare prima un Pod e poi un Deployment.

Quando si definisce un file in Kubernetes, YAML offre una serie di vantaggi, tra cui:

- *semplicità d'uso*: non dovrà più aggiungere tutti i tuoi parametri alla riga di comando;
- *manutenzione*: i file YAML possono essere gestiti tramite software per il controllo del codice sorgente, come un repository GitHub, in modo da poter tenere traccia delle modifiche;
- *flessibilità*: sarai in grado di creare strutture molto più complesse utilizzando YAML rispetto a quanto si possa fare da riga di comando.

YAML è un *superset* di JSON, il che significa che qualsiasi file JSON valido è anche un file YAML valido. Quindi, se la vedi da un'altra prospettiva, se conosci JSON sei già a buon punto, e scriverai il tuo YAML senza troppe difficoltà.

Nel caso in cui tu non lo conosca... fortunatamente, YAML è relativamente facile da imparare. Ci sono solo due tipi di strutture che devi conoscere in YAML:

- Liste
- Mappe

Questo è tutto. Potresti avere mappe di liste e liste di mappe e così via, ma se hai chiare queste due strutture, sei a metà dell'opera. Questo non vuol dire che non ci siano cose più complesse che puoi fare ma, in generale, questo è tutto ciò di cui hai bisogno per iniziare.

Mappe

Iniziamo dando un'occhiata alle mappe in YAML. Le mappe ti consentono di associare coppie nome-valore, il che ovviamente è conveniente quando stai tentando di impostare le informazioni di configurazione. Per esempio, potresti avere un file di configurazione che inizia come il Listato C.1.

Definizione della risorsa pod tramite YAML

```
---  
apiVersion: v1  
kind: Pod
```

La prima riga è un separatore ed è facoltativa, a meno che tu non stia cercando di definire più strutture in un unico file. Di seguito, come puoi vedere, abbiamo due valori, `v1` e `Pod`, mappati su due chiavi, `apiVersion` e `kind`.

Questo genere di definizione è piuttosto semplice e puoi pensarlo in termini di equivalente JSON:

Definizione della risorsa pod tramite JSON

```
{  
    "apiVersion": "v1",  
    "kind": "Pod"  
}
```

Si noti che, nella nostra versione YAML, le virgolette sono facoltative; il processore che elaborerà il file sarà in grado di capire che sta lavorando con una stringa in base alla formattazione.

Puoi anche specificare strutture più complicate creando una chiave che esegue il mapping a un'altra mappa, anziché a una stringa, come nel Listato C.3.

Definizione della risorsa pod tramite YAML

```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: my-site  
  labels:  
    app: web
```

In questo caso abbiamo una chiave, `metadata`, che ha come valore una mappa con altre 2 chiavi, `name` e `labels`. La stessa chiave delle etichette ha una mappa come valore. Puoi annidarli quanto vuoi.

Il processore YAML sa come tutti questi pezzi si relazionano tra loro perché abbiamo indentato le linee. In questo esempio ho usato 2 spazi per la leggibilità, ma il numero di spazi non ha importanza, purché sia almeno 1 e purché tu sia coerente. Per esempio, il nome e le labels sono allo stesso livello di indentazione, quindi il processore sa che fanno entrambi parte della stessa mappa; sa che app è un valore per il selettore labels perché è ulteriormente rientrata.

Consiglio: non utilizzare MAI i tab in un file YAML, ma usa gli spazi. I tab sono stati vietati poiché sono trattati in modo diverso da editor e strumenti diversi. E, poiché l'indentazione è così fondamentale per una corretta interpretazione di YAML, questo problema è troppo complicato anche solo per tentare.

Quindi, se dovessimo tradurre questo in JSON, sarebbe simile al listato C.4.:

Definizione della risorsa pod tramite JSON

```
{  
    "apiVersion": "v1",  
    "kind": "Pod",  
    "metadata": {  
        "name": "my-site",  
        "labels": {  
            "app": "web"  
        }  
    }  
}
```

Liste

Gli elenchi YAML sono letteralmente una sequenza di oggetti. Di seguito un esempio.

Definizione di una lista in formato YAML

```
args:  
- sleep  
- "1000"  
- message  
- "Bring back Lessie!"
```

Come puoi vedere qui, puoi avere praticamente qualsiasi numero di elementi in una lista, che è definito come elementi che iniziano con un trattino (-) con due spazi rispetto all'elemento genitore.

In JSON, sarebbe come nel Listato C.6.

Definizione di una lista in formato JSON

```
{  
    "args": ["sleep", "1000", "message", "Bring back Firefly!"]  
}
```

E, naturalmente, i membri della lista possono anche essere mappe:

Definizione di una lista contenente una mappa

```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: my-site  
  labels:  
    app: web  
spec:  
  containers:  
    - name: front-end  
      image: nginx  
      ports:  
        - containerPort: 80  
          name: backend  
      image: myimage:latest  
      ports:  
        - containerPort: 88
```

Quindi, come puoi vedere qui, abbiamo un elenco di "oggetti" `containers`, ognuno dei quali consiste in un nome, un'immagine e un elenco di porte. Ciascun elemento dell'elenco sotto le porte rappresenta una mappa che elenca `containerPort` e il relativo valore.

Per completezza, diamo un'occhiata rapidamente all'equivalente JSON:

Definizione di una lista contenente una mappa in formato JSON

```
{  
  "apiVersion": "v1",  
  "kind": "Pod",  
  "metadata": {  
    "name": "my-site",  
    "labels": {  
      "app": "web"  
    }  
  },  
  "spec": {  
    "containers": [  
      {  
        "name": "front-end",  
        "image": "nginx",  
        "ports": [  
          {  
            "containerPort": "80"  
          }  
        ]  
      },  
      {  
        "name": "backend",  
        "image": "myimage:latest",  
        "ports": [  
          {  
            "containerPort": "88"  
          }  
        ]  
      }  
    ]  
  }  
}
```

```
        "containerPort": "88"
    }
]
}
}
```

Come puoi vedere, è possibile scrivere e descrivere oggetti piuttosto complessi usando YAML o JSON; per riassumere, abbiamo:

- mappe, che sono gruppi di coppie nome-valore;
- liste, che sono singoli elementi figli di un elemento principale;
- mappe di mappe;
- mappe di liste;
- liste di liste;
- liste di mappe.

Fondamentalmente, qualunque struttura tu voglia mettere insieme, puoi farlo con queste due strutture in diverse combinazioni.

Prefazione

Ringraziamenti

Introduzione

Struttura

A chi è rivolto questo libro

Terminologia

Errori e feedback

Capitolo 1 - Primi passi

Dal container all'orchestratore

Kubernetes: breve storia

Perché Kubernetes

Che cosa abbiamo imparato

Capitolo 2 - Orchestrazione

Modelli

Soluzioni

Che cosa abbiamo imparato

Capitolo 3 - Architettura

Definizione generale

Componenti del control-plane

Componenti del worker

Che cosa abbiamo imparato

Capitolo 4 - Installazione

Che cosa installo?

Installazione tramite Docker Desktop

Installazione tramite CRC

Minikube

Rancher Desktop

Hello world

Che cosa abbiamo imparato

Capitolo 5 - Kubernetes per lo sviluppo

Cluster

Componenti applicativi

Che cosa abbiamo imparato

Capitolo 6 - Configurazione

Risorse

Che cosa abbiamo imparato

Capitolo 7 - Rete

- Come funziona: le basi
- Tipologie di Service
- Ingress
- Che cosa abbiamo imparato

Capitolo 8 - Storage

- Volumi
- PersistentVolumes e PersistentVolumeClaims
- StorageClass
- Che cosa abbiamo imparato

Capitolo 9 - Risorse aggiuntive

- StatefulSet
- DaemonSet
- Job
- CronJob
- Che cosa abbiamo imparato

Capitolo 10 - Autenticazione e autorizzazione

- Utenti
- Ruoli
- Che cosa abbiamo imparato

Capitolo 11 - Templating

- Helm
- Kustomize
- Helm vs Kustomize
- Che cosa abbiamo imparato

Capitolo 12 - Operatori

- Che cosa sono
- Come funziona
- Esempi
- Che cosa abbiamo imparato

Capitolo 13 - Kubernetes: casi d'uso

- Da Docker a Kubernetes
- Da Docker Compose a Kubernetes
- Avviare un cluster MongoDB
- Backup di un database Postgres
- Stack MEAN
- Che cosa abbiamo imparato

Capitolo 14 - Best practice per applicazioni enterprise

- Scaling
- Gestione delle risorse
- Namespace
- ResourceQuota
- Disaster Recovery
- Sostenibilità

Che cosa abbiamo imparato

Capitolo 15 - Kubernetes on cloud

Managed vs self-hosted

AWS

Azure

Google Cloud Platform

Mia-Platform

Che cosa abbiamo imparato

Appendice A - Risorse utili

Certificazioni

Documentazione utile, e non solo

Eventi

Appendice B - Istruzioni di base

Gestione del cluster

Namespace

Risorse

Rete

Logs

Eventi

Nodi

Appendice C - YAML

Nozioni di base

Mappe

Liste