View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by AMS Tesi di Laurea

ALMA MATER STUDIORUM – UNIVERSITA' DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

ELABORATO IN
SYSTEMS INTEGRATION

Kubernetes

Candidato:
Umberto Baldini

Relatore:
Vittorio Ghini

Anno Accademico 2019/20

Indice

1 INTRODUZIONE 7

2 CONTESTO 9

2.1 INFRASTRUTTURA CLOUD, CONCETTI CHIAVE 9

2.2 CONTAINER, MECCANISMI ALLA BASE 11

2.3 LE TECNOLOGIE PRESE IN ESAME 11

2.4 ASTRAZIONE DELL'INFRASTRUTTURA 13

3 ARCHITETTURA CLUSTER 15

3.1 PIANO DI CONTROLLO 15

3.2 ADDONS 17

3.3 STATO DEL NODO 17

3.3.1 Indirizzi 18

3.3.2 Condizioni 18

3.3.3 Capacità e allocabilità 19

3.3.4 Info 20

3.4 MANAGEMENT 20

3.4.1 Heartbeats 21

3.4.2 Zone di disponibilità 21

3.5 AUTO-REGISTRAZIONE DEI NODI 22

3.5.1 Amministrazione manuale del nodo 23

3.6 CAPACITÀ DEI NODI 23

3.6.1 Topologia dei Nodi 24

3.7 COMUNICAZIONE MASTER-NODE 24

3.7.1 Da cluster a master 24

3.7.2 Da master a cluster 25

4 LAVORARE CON OGGETTI KUBERNETES 27

4.1 LE API KUBERNETES 27

4.1.1 Oggetti Kubernetes 28

4.1.2 Gestione oggetti Kubernetes . 29

4.1.3 Nomi e ID degli Oggetti . . . 30

4.1.4 Etichette e selettori 31

4.1.5 Annotazioni 31

4.1.6 Selezionatori di campi 32

4.2 NAMESPACES 32

5 IMMAGINI DEI CONTAINER 33

5.1 GESTIONE IMMAGINI 33

5.2 CONTAINER ENVIRONMENT . . 33

5.2.1 Info sui Container 34

5.2.2 Info sul Cluster 34

5.3 CLASSE RUNTIME 34

5.3.1 Motivazione 34

5.4 CONTAINER LIFETIME HOOKS . 35

6 I POD 37

6.1 CICLO DI VITA DEI POD 38

6.1.1 Fasi 38

6.1.2 Condizioni 39

6.1.3 Probes (sonde) 40

6.1.4 Stato del container 40

6.1.5 Restart policy 41

6.1.6 Lifetime 41

6.2 INIT CONTAINER 41

6.2.1 Differenze dai container normali 41

6.2.2 Utilizzo 42

6.2.3 POD Presets 42

6.3 VINCOLI SULL'ESPANSIONE DELLA TOPOLOGIA DEI POD . . 42

6.4 INTERRUZIONI NELL'ESECUZIONE 43

6.5 CONTAINER EFFIMERI 44

7 I CONTROLLER 45

7.1 CONTROLLER PATTERN 45

7.2 CONCETTI SUL CLOUD CONTROLLER MANAGER 46

7.3 REPLICA SET 47

7.3.1 Quando usare i ReplicaSet . . 48

7.4 DEPLOYMENT 48

7.4.1 Casi d'uso 48

7.4.2 Esempio 49

7.4.3 Rollback 50

7.4.4 Scalare un Deployment . . . 51

7.4.5 Scrivere le specifiche di un Deployment 51

7.4.6 RollingUpdate 51

7.5 STATEFULL SET 52

7.5.1 Limitazioni 52

7.6 DAEMON SET 53

7.7 GARBAGE COLLECTION 53

7.7.1 Oggetti proprietari e dipendenti 53

7.7.2 Policy di cancellazione degli oggetti dipendenti 54

7.8 TTL CONTROLLER PER RISORSE FINITE 54

7.9 I JOB 54

7.10 CRONJOB 55

7.10.1 Limitazioni 55

8 SERVIZI, LOAD BALANCING E NETWORKING 57

8.1 SERVIZI 57

8.1.1 Motivazioni 57

8.1.2 Definire un servizio 58

8.1.3 Servizi senza selettori 58

8.1.4 IP Virtuali e proxy di servizi 59

8.1.5 Tipi di modalità proxy 59

8.1.6 Servizi Headless 61

8.1.7 Cercare i servizi 61

8.1.8 Pubblicare servizi 62
8.1.9 Scalare 62
8.2 TOPOLOGIA DEI SERVIZI 63
8.2.1 Utilizzo 63
8.2.2 Limiti 63
8.3 ENDPOINTSLICE 64
8.3.1 Risorse degli EndpointSlice . 64
8.3.2 Controller EndpointSlice . . . 64
8.4 DNS PER SERVIZI E POD 65
8.4.1 Servizi 65
8.4.2 POD 66
8.5 CONNETTERE APPLICAZIONI CON I SERVIZI 66
8.5.1 Rendere sicuro il servizio . . 67
8.5.2 Pubblicare il servizio 67
8.6 OGGETTO INGRESSO 67
8.6.1 Regole di ingresso 68
8.6.2 Classi di Ingressi 68
8.6.3 Fanout 68
8.6.4 LoadBalancing 69
8.7 CONTROLLER DI INGRESSO . . 69
8.8 POLITICHE DI RETE 69
8.8.1 POD isolati e non 69
8.9 AGGIUNGERE HOSTALIASES AI POD 70
8.10 IPv4/IPv6 STACK DUALE 70

9 MEMORIA DI MASSA (STORAGE) 71

9.1 VOLUMI 71
9.1.1 Propagazione del montaggio . 72
9.2 VOLUMI PERSISTENTI 73
9.2.1 Fornire Volumi persistenti . . 73
9.2.2 Binding 73
9.2.3 Oggetto Storage in Utilizzo Protetto 74
9.2.4 Riutilizzo 74
9.2.5 Modalità di accesso 74
9.3 SNAPSHOT DI VOLUMI 75
9.3.1 PersistentVolumeClaim come protezione dell'origine di uno snapshot 75
9.3.2 Cancellazione 75
9.3.3 Creare un Volume da uno snapshot 76
9.4 CLASSI DI STORAGE 76
9.4.1 Permettere espansione del Volume 76
9.5 CLASSI DI SNAPSHOT DI VOLUMI 76
9.6 FORNIRE VOLUMI DINAMICAMENTE 77
9.6.1 Comportamento di deafult . . 77
9.7 LIMITI DI VOLUMI SPECIFICI PER NODO 77

10 LE POLICY 79

10.1 IMPORRE DEI LIMITI 79
10.2 QUOTE DI RISORSE 80
10.3 POLICY DI SICUREZZA DEI POD 80

11 SCHEDULER KUBERNETES 81

11.0.1 Kube-Scheduler 81
11.1 FRAMEWORK DI SCHEDULAZIONE 82

12 ESEMPI PRATICI **83**

12.1 HARDWARE & SOFTWARE 83

12.1.1 installazione 84

12.1.2 LENS IDE 85

12.2 ESEMPI 86

12.2.1 Applicazione Guestbook stateless 86

12.2.2 Sito web Wordpress con database MySQL 91

12.2.3 Verifica 95

Capitolo 1

INTRODUZIONE

Il soggetto della mia tesi è **Kubernetes**, un software in grado di attuare l'**orchestrazione** di grandi quantità di **container**.
L'obbiettivo del software è di rendere disponibili ed affidabili i servizi in rete, mascherando tutte le strutture utilizzate e l'implementazione degli stessi all'interno del **cloud**.

Dopo aver introdotto il contesto in cui Kubernetes si applica, passerò a descriverne il funzionamento dividendolo in parti.
Verrà mostrato come le tante astrazioni disponibili collaborino tra loro dando un impressione di **caos ordinato**, in cui tutti gli attori hanno un ruolo ed un'elenco di regole che puntualmente seguiranno, errore umano permettendo.

Per comprendere il funzionamento userò un **approccio top-down**; l'architettura del sistema darà una prima impressione delle capacità di Kubernetes.
Passerò poi alle astrazioni specifiche di Kubernetes implementate come oggetti, alla gestione dei container, della loro organizzazione in POD e come questi vivano, parlerò dell'importanza fondamentale dei controller, dell'utilizzo di servizi nella gestione del networking, dell'utilizzo di astrazioni per lo storage di dati e citerò meccanismi più avanzati come le policy di assegnazione di risorse hardware.
Infine proporrò un esempio pratico, testato sul mio portatile, per mostrare la facilità di introduzione all'utilizzo di Kubernetes.

Le potenzialità mostrate in questo testo si applicano ad un contesto di gestione dei server molto intenso, con tantissimi utenti ed applicazioni atti-

ve. Non rientra nella comune attività del programmatore, il quale collabora tipicamente utilizzando Docker (o altri container runtime) sulla propria macchina ed implementando immagini di container pronte per il deployment.

Capitolo 2

CONTESTO

Il cloud mette a disposizione un numero enorme di servizi molto eterogenei, dietro questa nuvola astratta esiste una pletora di problematiche, alcune di queste: la disponibilità 24/7, la scalabilità delle richieste e la sicurezza.

La **gestione** di più sistemi software su più macchine potrebbe essere **semplificata** imponendo una serie di regole il più possibile generale, che valga per sistemi e macchine anche molto diversi tra loro.

2.1 INFRASTRUTTURA CLOUD, CONCETTI CHIAVE

La **virtualizzazione** di un sistema operativo (SO) è una tecnica che permette di eseguire un SO "ospite" nello stesso modo di una qualunque applicazione, all'interno di un altro SO "ospitante", installato su di una macchina fisica. Si parla in questo caso di Macchina Virtuale.

Un **container** è un'astrazione che definisce un contenitore per applicazioni software, alle quali fornisce tutto il necessario (codice, esecuzione runtime, strumenti/librerie di sistema ed impostazioni) virtualizzando le risorse fisiche fornite dal sistema operativo sottostante.

Entrambi i metodi permettono l'esecuzione di un numero di MV/container in parallelo sulla stessa macchina dipendente solo dalle risorse rese dispo-

nibili dalla macchina fisica, in questo modo forniscono un ambiente coerente per lo sviluppo e l'esecuzione di applicazioni a prescindere dalla macchina ospitante. Si riesce poi a **separare il contesto** della macchina fisica: nel caso in cui le MV/container dovessero essere soggette a fault di sistema o attacchi informatici, non avranno modo di coinvolgere anche la macchina ospitante, creando un ulteriore livello di sicurezza

.

La **differenza** tra MV e container risiede nell'utilizzo delle risorse fisiche della macchina fisica; le MV virtualizzano l'hardware mentre i container il SO. Un container installa solamente la parte necessaria del SO in quanto condivide il Kernel con il SO ospitante e non impegna la CPU se non quando lo richiede l'applicazione interna.

Un **container è quindi più leggero**, più veloce all'avvio, inoltre è replicabile a runtime, il deployment di container è automatizzabile e altamente configurabile. Questo permette di gestire volumi di richieste molto variabili, instanziando solamente il servizio richiesto e non l'intero stack di cui fa parte. Inoltre il container è immutabile per design: non si può cambiare il codice mentre è in esecuzione. Per aggiungere dei cambiamenti bisogna ripartire dal progetto e produrre una nuova immagine del container.

Nella pratica una tecnica di virtualizzazione non esclude l'altra: un server fisico può ospitare più MV, che a loro volta possono ospitare cluster di container.

Architettura a micro-servizi: progettazione di un sistema software complesso scomposto in parti minime, considerate servizi, offerti ognuno da un software indipendente che mette tipicamente a disposizione una o più interfacce. Questo metodo rende un sistema enorme umanamente affrontabile, facilitando le varie fasi di sviluppo, deployment e mantenimento del software.

Ma **perché** si ospita in un'ambiente virtuale un'applicazione? Per offrire a quest'ultima un **contesto isolato e personalizzato**, fornendo le librerie e dipendenze per singolo servizio, cosi da far eseguire anche applicazioni molto eterogene sulla stessa macchina fisica e per la maggior sicurezza dei sistemi, garantita dalla separazione dei contesti.

2.2 CONTAINER, MECCANISMI ALLA BASE

Un container è un insieme di processi visibile dalla macchina host, che incapsulano un contesto simile a quello di una MV, con la differenza di condividere il kernel con l'host senza essere abilitati a modificarlo. Per fare questo, sfrutta delle funzionalità di Linux:

- **cgroup**: per la misurazione e limitazione di risorse assegnate ad un processo (file system, risorse, rete), limitando cio che possono usare.
- **namespace**: forniscono ai processi una visione manipolata del sistema, ergo limita cio che possono vedere.
- **chroot**: per la modifica della directory di riferimento dei processi in esecuzione.
- **copy-on-write storage**: crea un nuovo container istantaneamente, abbattendo i tempi di boot.

Considerazione: tutti gli utenti sono costantemente all'interno di un container, dato che utilizzano i cgroup ed il namespace della root.

2.3 LE TECNOLOGIE PRESE IN ESAME

Docker è attualmente la piattaforma più utilizzata per la creazione di **Container Linux**; è un progetto atto ad automatizzare il deployment di applicazioni contenute nei container eliminando problemi di compatibilità; se il server/computer è in grado di ospitare docker allora può ospitare qualunque applicazione.
Sviluppato con il linguaggio GO, nato nel 2013 è compatibile con gli OS principali: MacOs, Windows e il nativo GNU/Linux.

Kubernetes: deriva da Borg, un progetto interno a Google per la **gestione di cluster di container**, ovvero agglomerati più o meno grandi di container che comunicano tra loro.
Il nome deriva dal greco e significa timoniere o pilota. Venne pubblicato come progetto open source nel gennaio del 2015, per aumentarne la velocità

e qualità di sviluppo.
Utilizza un **container runtime** per creare i container e raggrupparli in POD, il più utilizzato è Docker, alcune alternative: CRI-O, containerd, frakti.
I suoi **compiti/obbiettivi principali** sono: automatizzare il deployment, distribuire il carico di lavoro, gestire il routing, aggiornare il contenuto dei container in accordo con le loro immagini, mantenere i processi continuamente attivi e funzionanti, gestione dello storage (memoria di massa) locale e nel cloud, rollout e rollback automatici, orchestrazione del deployment sulle macchine (bin packed problem) e protezione nell'uso e memorizzazione di informazioni sensibili (password, chiavi SSH). Kubernetes fornisce all'utente la capacità di dichiarare **regole globali** per automatizzare tutte queste operazioni.
La versione attuale, presa in esame nel documento, è la 1.18 e presenta ancora molte feature in fase alfa e beta. Lo sviluppo è continuo e sicuramente c'è ancora molto margine di espansione delle funzionalità.

Kubernetes lavora con questi componenti logici:

POD: l'elemento più piccolo nello stack fornito da Kubernetes, contiene uno o più container, Kubernetes si relaziona solo ai POD. Gruppi di POD vengono poi posizionati su di una singola macchina (fisica o virtuale), definita **"nodo"**. Gruppi di nodi formano un **cluster**, ognuno dei quali viene sorvegliato da un nodo definito **master**.

Servizio: un'applicazione utile ad altre applicazioni e residente su più POD, viene resa disponibile come servizio in rete.

Controller: oggetto Kubernetes responsabile della gestione di altri oggetti, sono specializzati nel compito loro assegnato.

Volume: astrazione di una directory resa disponibile ai container di uno o più POD. Come questa directory sia ospitata, quale sistema la renda disponibile oppure il suo contesto sono determinati dal tipo di volume.

Spazi dei nomi (namespaces): astrazione di cluster virtuali multipli situati nello stesso cluster fisico, denominati appunto namespaces. Sono utilizzati per gestire molti utenti che appartengano a diversi gruppi o progetti.

Job: creano uno o più POD ed assicurano che un certo numero di questi termineranno con successo, completando il compito assegnato.

2.4 ASTRAZIONE DELL'INFRASTRUTTURA

L'**infrastruttura** che Kubernetes andrà a controllare sarà **astratta**, rendendo ogni server identico all'altro, come ogni container nato dalla stessa immagine.
Il rischio di mancanza del servizio sarà distribuito su più cluster presenti su più nodi, a seconda delle richieste dell'utente, rendendo il sistema più affidabile.

Inoltre, Kubernetes è **resiliente**: controlla ciclicamente lo stato dei processi attivi con quello descritto dalle immagini da cui sono stati istanziati e, nel caso in cui dovesse rilevare differenze, riavvierebbe il processo.
La stessa gestione si applica agli **aggiornamenti**: nel momento in cui viene inserita una nuova versione dell'immagine di partenza di un dato container, Kubernetes inizia a confrontarli e a sostituirli uno alla volta.

Capitolo 3

ARCHITETTURA CLUSTER

Un **nodo** è una **macchina in produzione** in Kubernetes, può essere una macchina virtuale oppure fisica. Ogni nodo contiene i servizi necessari per eseguire uno o più POD ed è gestita dal nodo master.
I servizi su un nodo minion (non-master) includono il runtime del container, **Kubelet** e **Kube-proxy**.

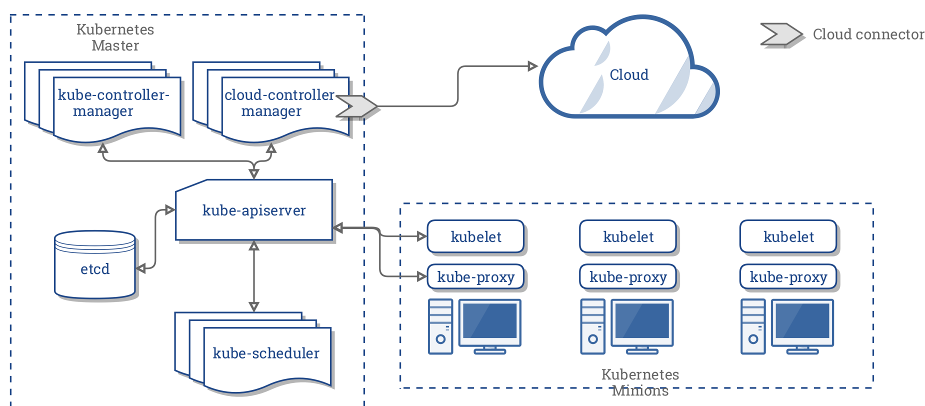

Figura 3.1: Architettura di un cluster Kubernetes

3.1 PIANO DI CONTROLLO

Consiste di un insieme di processi in esecuzione nei nodi del cluster:

- Il **Master Kubernetes** è un nodo che ospita il database di kubernetes ed alcuni processi fondamentali:

  - **kube-apiserver**: espone l'API kubernetes ed è il front-end del Piano di Controllo. Progettato per scalare orizzontalmente, offre le API Kubernetes. Si possono istanziare diverse entità di kube-apiserver per bilanciare il traffico tra loro.
  - **kube-controller-manager**: esegue i processi controller. Ogni controller è un processo separato ma per ridurne la complessità, sono tutti compilati insieme in un unico binario ed eseguono in un unico processo.
  - **kube-scheduler**: rileva la creazione di nuovi POD, se non sono ancora stati assegnati ad un nodo gliene sceglie uno. I fattori di scelta per le decisioni di scheduling includono le risorse individuali e collettive richieste, limitazioni hardware/software/policy, localizzazione dei dati, interferenze tra carichi di lavoro e scadenze.
  - **cloud-controller-manager**: esegue dei controller che interagiscono con il cloud provider sottostante. Fornisce le dipendenze necessarie per ogni cloud provider a Kubernetes, nelle release future il codice specifico per ogni cloud vendor verrà mantenuto dal cloud vendor stesso.
  - **node-controller**: esegue controlli ciclici sui nodi per assegnare lo stato ready o spegnere tutti i POD di un nodo, assegna blocchi CIDR ai nuovi, mantiene la lista interna di macchine disponibili aggiornata.
  - **etcd:** base di dati esterna a Kubernetes, mantiene delle coppie chiave-valore che descrivono i dati del cluster. È necessario schedulare un backup di questi dati.

- Mentre ogni nodo **non-master** nel cluster contiene 2 processi:

  - **Kubelet**: un agente, si assicura che il container stia eseguendo in un POD. Prende in input un set di specifiche ottenute in vari modi e si assicura che i container descritti si attengano alle specifiche.
  - **Kube-proxy**: un proxy di rete che implementa parte del Servizio di Kubernetes. Gestisce le regole di rete del nodo; permettendo la comunicazione all'interno e all'esterno del cluster.
    Kube-proxy utilizza il packet filtering layer del SO, se esiste ed è disponibile, altrimenti instrada il traffico.

Questi processi mantengono una **registrazione di tutti gli oggetti Kubernetes** nel sistema ed eseguono controlli in loop per gestirne lo stato. In ogni momento, il Piano di Controllo sarà in grado di rispondere ai cambiamenti nel cluster, prendendo decisioni globali.
Il piano di Controllo può essere eseguito su qualsiasi macchina del cluster.

3.2 ADDONS

Gli addons utilizzano le risorse di Kubernetes per implementare **funzionalità** nel cluster, non obbligatorie ma spesso molto utili, eccone alcune:

- **DNS:** quasi obbligatorio in un cluster. In aggiunta ai DNS server del contesto, viene incluso automaticamente nelle ricerche dei container.

- **Web UI (Dashboard):** UI web-based per il general purpose, permette agli utenti di gestire e verificare le applicazioni in esecuzione nel cluster, ma anche il cluster stesso.

- **Container Resource Monitoring:** registra dati generici a intervalli regolari sui container in un database centrale, offre una UI per visionare i dati.

- **Cluster-level Logging:** responsabile per il salvataggio dei log dei container in un volume centralizzato che offre un interfaccia per il browsing.

3.3 STATO DEL NODO

Il **Node Status** contiene queste informazioni: **indirizzi**, **condizioni**, **capacità** ed **allocabilità**, **info**. Lo status e altri dettagli sul nodo possono essere mostrati utilizzando questi comandi:

```
kubectl describe node <node-name>
```

3.3.1 Indirizzi

l'uso di questi campi varia a seconda del cloud provider o se si usa una configurazione bare metal.

- `HostName`: l'hostname riportato nel kernel del nodo.Può essere sovrascritto col comando: `kubelet -hostname-override <hostname>`
- `ExternalIP`: l'indirizzo IP che identifica il nodo dall'esterno del cluster
- `InternalIP`: l'indirizzo IP del nodo utilizzato all'interno del cluster

3.3.2 Condizioni

descrive lo stato di tutti i nodi in esecuzione. Alcuni esempi:

- **Ready**: true se il nodo è pronto ad accettare dei POD, false se non lo è e unknown se il controller del nodo non è stato aggiornato nell'ultimo `node-monitor-grace-period` (ogni 40 secondi, di default).
- **MemoryPressure**: vero se la memoria principale disponibile del nodo è sotto un certo limite, altrimenti falso
- **PIDPressure**: vero se ci sono troppi processi in esecuzione nel nodo, altrimenti falso
- **DiskPressure**: vero se lo spazio libero su disco è poco, altrimenti falso
- **NetworkUnavailable**: vero se la rete del nodo non è configurata, altrimenti falso

La condizione del nodo è rappresentata come un oggetto JSON. Per esempio, questa risposta descrive un nodo correttamente configurato e pronto ad ospitare dei POD (definito **healthy**):

```
"conditions": [{
        "type": "Ready",
        "status": "True",
        "reason": "KubeletReady",
        "message": "kubelet is posting ready status",
```

```
                "lastHeartbeatTime": "2019-06-05T18:38:35Z",
                "lastTransitionTime": "2019-06-05T11:41:27Z"
        }]
```

Se lo stato della condizione type è diverso da Ready per un tempo maggiore del `pod-eviction-timeout` (configurabile), un argomento viene passato al kube-controller-manager e tuti i Pod sul nodo vengono schedulati per la **cancellazione** dal controller del nodo.
Il timeout di default per l'espulsione è di 5 minuti. In alcuni casi quando il nodo non è raggiungibile il server API non riesce a comunicare con il kubelet sul nodo, per cui la decisione di eliminare il nodo non può essere comunicata. Fino alla cancellazione questi POD continueranno a essere in esecuzione su di una partizione del nodo. Prima della versione 1.5 di Kubernetes venivano automaticamente eliminati.

I **POD** in esecuzione sul **nodo irrangiungibile** sono rappresentati con uno **stato Terminating** oppure **Unknown**.

Nel caso in cui Kubernetes non possa dedurre dall'infrastruttura sottostante se un nodo abbia lasciato permanentemente un cluster, l'amministratore del cluster potrebbe essere costretto a cancellare l'oggetto nodo manualmente. Questa operazione causa la cancellazione di tutti gli oggetti POD dal nodo, liberando i loro nomi.

Il workflow del controller del nodo automaticamente crea dei metadati che rappresentano delle condizioni; quando lo scheduler assegna un POD ad un nodo, lo scheduler tiene conto di questi metadati.

3.3.3 Capacità e allocabilità

Le risorse disponibili sul nodo: CPU, memoria e il massimo numero di POD che possono essere schedulati sul nodo.

La somma delle parti dello status indica le risorse totali del nodo (**capacity**) comprese quelle ancora disponibili per POD (**allocatable**).

3.3.4 Info

Informazioni **generali** sui nodi, ad esempio versione del kernel, di Kubernetes (kubelet e kube-proxy), di Docker (se utilizzato) e nome del SO. Queste informazioni sono raccolte da kubelet nel nodo.

3.4 MANAGEMENT

Diversamente da POD e servizi, un nodo non è creato da Kubernetes, ma esternamente dai cloud provider, oppure esiste in un pool di macchine fisiche o virtuali.
Kubernetes rappresenta il **nodo** con un **oggetto** corrispondente e ne controlla la validità subito dopo la creazione. Per esempio, questo oggetto rappresenta un nodo "**healthy**" (sano: pronto per ospitare POD):

```
{
        "kind": "Node",
        "apiVersion": "v1",
        "metadata": {
                "name": "10.240.79.157",
                "labels": {
                        "name": "my-first-k8s-node"
                }
        }
}
```

Il controllo sulla "salute" (stato) del nodo viene eseguito da Kubernetes basandosi sul campo **metadata.name**; se il nome è valido nel sottodominio DNS, allora tutti i servizi necessari sono attivi, quindi il nodo è healthy, altrimenti viene ignorato.
Al momento, esistono 3 componenti che interagiscono con l'interfaccia dei nodi di Kubernetes: **controller del nodo, kubelet, e kubectl**.

Nota: Kubernetes non elimina l'oggetto del nodo, se non healthy, ma continua a controllarne lo stato. É necessario cancellare esplicitamente il nodo per fermare questi controlli.

3.4.1 Heartbeats

Segnale inviato dai nodi Kubernetes al **node-controller**, aiutano a determinarne la **disponibilità**. Ne esistono 2 tipi: aggiornamento del **NodeStatus** e dell'oggetto **Lease**.
Ogni nodo è associato ad ud oggetto Lease nel namespace **kube-node-lease**.
Il Lease è una risorsa molto leggera, che migliora le performance del segnale heartbeat allo scalare del cluster.

Kubelet è responsabile per creare e aggiornare il NodeStatus e dell'oggetto Lease; lo fa quando c'è un cambiamento di stato o se non ci sono stati aggiornamenti per un certo intervallo (default: 5 minuti).
Kubelet crea e aggiorna l'oggetto Lease ogni 10 secondi di default. Gli aggiornamenti di Lease e NodeStatus sono indipendenti tra loro.

3.4.2 Zone di disponibilità

Una zona di disponibilità è un **luogo fisico** dove sono ospitati i nodi/macchine, spesso appartengono al cloud provider che li gestisce.

Dalla versione 1.4 di Kubernetes, il controller del nodo controlla lo stato di tutti i nodi di un cluster prima di prendere decisioni riguardo alla terminazione dei POD. In molti casi, i **controller limitano il numero di terminazioni** ad un certo rateo (default: 0.1, quindi un nodo ogni 10 secondi).

La **policy di terminazione cambia** a secondo della zona di disponibilità in cui si trova il nodo unhealthy:
il controller prima verifica la percentuale di nodi unhealthy nella zona (NodeReady con valore ConditionUnknown o ConditionFalse) nello stesso momento.
Se questa frazione del totale è almeno `-unhealthy-zone-threshold` (default 0.55) allora il rateo di terminazione è ridotto a `-secondary-node-eviction-rate` (default: 0.01), se il cluster ha meno di `-large-cluster-size-threshold` (default: 50) nodi non avverrà la terminazione.

Si fa un analisi per zone di disponibilità perchè una di queste potrebbe essere partizionata dal master mentre le altre rimangono connesse. Se il cluster non è diviso tra più zone di disponibilità del cloud provider, allora ne

esiste una soltanto che ospita l'intero cluster.

Una buona ragione per espandere un cluster su più zone di disponibilità è la **resilienza al guasto** di una intera zona: le altre possono continuare ad offrire i servizi online. In ogni caso, se tutti i nodi di una zona sono non-healthy allora la terminazione procede normalmente.
Il caso peggiore è rappresentato da **tutte le zone** completamente **non-healthy**, il controller dei nodi, allora, presume che ci siano **problemi di connettività** del master e ferma il processo di terminazione fino alla ripresa della connettività.

Da Kubernetes 1.6, il NodeController è anche responsabile per la **terminazione dei POD** che vivono su nodi definiti,con un metadato, come `NoExecute`. Dalla 1.8 può essere incaricato per creare questi **metadati**, detti **taint** (macchia, oppure marchio) che rappresentano la **condizione del nodo**. È una feature in fase alpha in questa versione.

3.5 AUTO-REGISTRAZIONE DEI NODI

Se il flag `-register-node` ha valore true, kubelet cercherà di registrare se stesso all'API server. Questo è il comportamento preferibile, utilizzato in molte distro.
Per auto registrarsi, kubelet viene avviato con queste opzioni:

- `-kubeconfig`: Path alle credenziali di autenticazione per l'API server
- `-cloud-provider`: Modalità di dialogo dipendente dal cloud provider, per leggere metadati che riguardano il nodo stesso
- `-register-node`: Registrazione automatica all'API server
- `-register-with-taints`: Registrazione automatica all'API server con una certa lista di taint (separati da virgole: <chiave>=<valore>:<effetto>). Non si registra se `-register-node` è false
- `-node-ip`: Indirizzo IP del nodo
- `-node-labels`: Etichetta da aggiungere quando si registra il nodo nel cluster

- `-node-status-update-frequency`: Specifica quanto spesso kubelet invierà lo stato del nodo al master

3.5.1 Amministrazione manuale del nodo

Un amministratore del cluster può **creare e modificare gli oggetti nodo**.

Per creare manualmente, deve settare `-register-node=false`, mentre può modificare le risorse del nodo a prescindere dal flag, tra le quali le etichette che specificano impostazioni.

Le etichette dei nodi possono essere usate insieme ai selettori di nodi sui POD per controllare lo scheduling, ad esempio per costringere un POD ad essere istanziabile soltanto in un sottoinsieme di nodi.

Rendere un nodo non schedulabile non crea modifiche sui POD già presenti sul nodo. È un passo utile per operazioni quali il reboot del nodo. Il comando per rendere un nodo non schedulabile:

```
kubectl cordon NODENAME
```

Nota: i POD creati dal controller `DaemonSet` saltano i controlli dello scheduler Kubernetes e non rispettano la non schedulabilità di un nodo: si presume che il processo appartenga alla macchina anche se sta per riavviarsi. Una conseguenza della non schedulabilità è la rimozione del nodo da tutte le liste utilizzate dai processi `LoadBalancer`,**eliminando** così la g**estione del carico di lavoro** per il nodo.

3.6 CAPACITÀ DEI NODI

La capacità di un nodo (**numero di CPU e quantità di memoria**) è descritta nel suo oggetto, normalmente, i nodi si registrano e riportano la loro capacità quando viene creato un oggetto nodo.
Con l'amministrazione manuale, questo dato non viene impostato autonomamente.

Lo scheduler Kubernetes si assicura che esistano abbastanza risorse per tutti i POD sul nodo verificando che la somma delle richieste dei container

sul nodo non sia maggiore della sua capacità, includendo tutti i container istanziati da kubelets, ma non quelli direttamente istanziati dal container runtime per qualsiasi processo al di fuori dei container.

3.6.1 Topologia dei Nodi

Una feature in fase alfa in Kubernetes 1.17. Se si abilita la feature TopologyManager, allora kubelet può utilizzare i consigli sulla topologia quando prende decisioni riguardo all'assegnamento di risorse.

3.7 COMUNICAZIONE MASTER-NODE

Se gli utenti implementano anche la gestione della rete, possono ottenere la capacità di passare per reti non sicure (es. pubbliche), utilizzando una comunicazione tra IP pubblici su cloud provider differenti.

3.7.1 Da cluster a master

Tutti i path per la comunicazione da un cluster al master terminano all'**API server**, nessun altro componente del master espone servizi per processi remoti. Normalmente, l'API server è configurato per ascoltare le connessioni sulla porta standard HTTPS (443), quindi sicura, con una o più forme di autenticazione per client abilitate (specialmente se vengono utilizzate richieste anonime di service account token).

I nodi dovrebbero essere forniti con il **certificato pubblico di root** per il cluster (oltre alle credenziali), in modo da connettersi in sicurezza con l'API server. Per esempio, su di un deployment Google Kubernetes Engine, le credenziali del client fornite a kubelet sono nella forma di un certificato client.

I POD che vogliono connettersi all'API server possono farlo in sicurezza utilizzando il service account; Kubernetes fornirà automaticamente il certificato root e un token valido nel POD alla sua istanziazione. Il servizio Kubernetes (in tutti i namespace) è configurato con un indirizzo IP virtuale che redireziona (via kube-proxy) su HTTPS all'API server.

Anche i componenti del master comunicano con l'API server del cluster in maniera sicura.

3.7.2 Da master a cluster

Esistono 2 tipi primari di percorsi di comunicazione dal master (API server) al cluster: il primo va dall'API server al processo kubelet che esegue su ogni nodo del cluster, il secondo arriva ad ogni nodo, POD o servizio attraverso la funzionalità proxy dell'API server.

Le comunicazioni **da apiserver a kubelet** sono utilizzate per:

- Recuperare i log dai POD
- Collegarsi grazie a kubectl a POD in esecuzione
- Fornire la funzionalità port-forwarding a kubelet

L'API server non controlla il certificato del client, il che rende la connessione debole agli attacchi man-in-the-middle, nonchè impossibile l'utilizzo di reti non fidate e/o pubbliche.
È necessario utilizzare il flag `-kubelet-certificate-authority` per fornire all'API server un certificato root, per la verifica di quello del client.

Le comunicazioni **da API server ai nodi, pod e servizi** utilizzano connessioni HTTP di default, non utilizzano autenticazione ne criptazione del contenuto. Utilizzare una connessione HTTPS in questo caso non permette la verifica dei certificati ne l'aggiunta di credenziali, risulta quindi non sicura.

I **tunnel SSH** sono attualmente deprecati, ma idealmente una valida alternativa.

Capitolo 4

LAVORARE CON OGGETTI KUBERNETES

Si utilizzano le **API di oggetti Kubernetes** per descrivere lo **stato desiderato del cluster**: quali applicazioni vogliamo istanziare, quali immagini utilizzare, il numero di repliche, quali reti e memorie di massa (storage) rendere disponibili e altro.
Si utilizza, per fornire queste descrizioni, tipicamente l'interfaccia a linea di comando (CLI) chiamata **kubectl**. Questa comunica con il nodo master, mentre le API Kubernetes servono per comunicare con il cluster.
Una volta impostato lo stato desiderato, il Piano di Controllo di Kubernetes lavora per rendere identico lo stato corrente grazie al `Pod Lifecycle Event Generator` (**PLEG**, generatore di eventi del ciclo di vita dei POD).

4.1 LE API KUBERNETES

Il meccanismo fondamentale per lo schema di **configurazione dichiarativa** del sistema è rappresentato dalle API Kubernetes, utilizzate dai suoi stessi componenti per interagire.
Il tool a riga di comandi kubectl può essere usato per creare, aggiornare ed eliminare le API di oggetti.
Kubernetes salva lo stato degli oggetti **serializzandoli** ed inserendo il testo ottenuto nel database etcd. Le API di Kubernetes accettano dati in formato **JSON** per la serializzazione, ma è disponibile anche uno standard **protobuf**,

personalizzato nel caso di comunicazioni intracluster, molto spesso però si utilizzano file **.yaml** per la descrizione degli oggetti.

Sono supportate molte versione delle API, accedibili tramite URL differenti. Questo permette una chiara rappresentazione delle risorse del sistema e di comportamento, oltre alla compatibilità a lungo termine per le applicazioni oppure alle API in fase alpha.

4.1.1 Oggetti Kubernetes

Sono entità persistenti nel sistema Kubernetes, il quale lavora per garantirne l'esistenza. **Gli oggetti descrivono lo stato del sistema:**

- quali applicazioni containerizzate sono in esecuzione e su quali nodi
- le risorse disponibili per le applicazioni
- le policy sul comportamento di queste applicazioni, ad esempio quelle di riavvio, fault-tolerance e aggiornamento

Ogni oggetto Kubernetes contiene 2 oggetti incapsulati che ne gestiscono la **configurazione**:

- **Spec**: descrive lo stato desiderato dell'oggetto
- **Status**: descrive lo stato attuale dell'oggetto, fornito e aggiornato da Kubernetes

Creare un oggetto Kubernets: si fa una richiesta API includendo i dati descrittivi in un file .yaml. Esempio: application/deployment.yaml

```
# per versioni precedenti 1.9.0 utilizzare apps/v1beta2
apiVersion: apps/v1
kind: Deployment
metadata:
	name: nginx-deployment
spec:
	selector:
		matchLabels:
```

```
                      app: nginx
# impone al deployment di avere 2 copie di questo container
       replicas: 2
       template:
              metadata:
                      labels:
                              app: nginx
              spec:
                      containers:
                      - name: nginx
                              image: nginx:1.7.9
                              ports:
                              - containerPort: 80
```

comando kubectl per fare il **deployment**:

```
       kubectl apply -f file.yaml --record
```

output:

```
       deployment.apps/nginx-deployment created
```

Campi obbligatori nel file .yaml:

- **apiVersion** – versione dell'API Kubernetes
- **kind** – tipo dell'oggetto
- **metadata** – dati che aiutano ad identificare gli oggetti, include il nome, UID e un namespace opzionale
- **spec** – stato desiderato dell'oggetto

4.1.2 Gestione oggetti Kubernetes

Un oggetto Kubernetes dovrebbe essere gestito utilizzando soltanto una tecnica tra queste:

- **Imperative commands:** un utente può comunicare direttamente con un oggetto in esecuzione nel cluster, inserendo comandi come argomenti o flags per kubectl. Esempio: `kubectl run nginx -image nginx`

- **Imperative Object configuration:** nel comando kubectl è necessario specificare l'operazione (create, replace ecc), i flag opzionali e almeno un file in formato YAML o JSON.

  Creare l'oggetto definito nel file:

  ```
  kubectl create -f nginx.yaml
  ```

  Cancellare l'oggetto definito in due file di configurazione:

  ```
  kubectl delete -f nginx.yaml -f redis.yaml
  ```

  Aggiornare l'oggetto sovrascrivendo la definizione dell'oggetto istanziato:

  ```
  kubectl replace -f nginx.yaml
  ```

- **Declarative object configuration:** un utente crea un file di configurazione locale, senza specificare le operazioni attuabili sul file. Creazione, aggiornamento e cancellazione sono rilevate automaticamente da Kubernetes per ogni oggetto. Così è possibile assegnare operazioni diverse per oggetti diversi, in una logica a directory.

Esempio: Processare tutti i file di configurazione di oggetti nella cartella config, istanziando gli oggetti. Con il comando 'diff' vengono mostrati i cambiamenti che verranno applicati.

```
kubectl diff -f configs/
kubectl apply -f configs/
```

4.1.3 Nomi e ID degli Oggetti

Ogni oggetto nel cluster ha un **nome** che risulta **unico per quel tipo di risorsa**, l'oggetto viene identificato grazie a degli URL di risorse, mentre l'**UID** deve essere **unico in tutto il cluster**.

Ci sono 4 tipi di vincoli sui nomi delle risorse:

- **nomi di sottodomini DNS**

- **etichette DNS**
- **nomi di PATH:** il nome non può essere '.', '..' e non può contenere '/' o '%'.
- **UID**: Una stringa generata da Kubernetes che identifica univocamente l'oggetto. Permette di distinguere tra occorrenze nel tempo di entità simili. Gli UID Kubernetes sono universalmente unici (UUID).

4.1.4 Etichette e selettori

Coppie chiave/valore che descrivono gli oggetti; **identificano** attributi degli oggetti che sono rilevanti e significativi per gli utenti, ma non implicano direttamente la semantica del sistema Kubernetes.
Le etichette possono essere utilizzate per organizzare e selezionare sottoinsiemi di oggetti, possono essere assegnate agli oggetti al momento della creazione oppure aggiunte e tolte in qualsiasi momento.
Ogni chiave deve essere unica per ogni oggetto. Sintassi:

```
"metadata": {
        "labels": {
                "key1" : "value1",
                "key2" : "value2"
        }
}
```

Una **chiave** valida ha **2 parti**: il **nome** è obbligatorio ed è preceduto da un '/'. Il **prefisso** opzionale, se specificato deve essere un sottodominio DNS: una serie di etichette DNS separate da punti '.', non più lunghe di 253 caratteri in totale, seguite da uno slash '/'.
Se il prefisso non viene messo, l'etichetta si presume privata dell'utente.

4.1.5 Annotazioni

Coppie **chiave/valore** che aggiungono **metadati arbitrari** agli oggetti. Tools e librerie possono recuperare questi metadati e mostrarli all'utente.

4.1.6 Selezionatori di campi

Filtri di risorse, permettono di selezionare risorse di Kubernetes sulla base del valore di uno o più campi. **Esempio**:

```
metadata.name = my-service
metadata.namespace != default
status.phase = Pending
kubectl get pods --field-selector status.phase = Running
```

4.2 NAMESPACES

Kubernetes supporta **multipli cluster virtuali** che siano sostenuti dallo stesso cluster fisico, chiamati **namespaces**. Sono progettati per gestire molti utenti in più team, o progetti.

I namespaces forniscono uno **scope**: i nomi delle risorse devono essere unici all'interno di un namespace, ma non in namespace diversi.
I namespace non possono essere incapsulati uno dentro l'altro, ogni risorsa Kubernetes può esistere soltanto in un namespace.
Si utilizzano le etichette per distinguere le risorse all'interno dello stesso namespace.

Quando si crea un servizio, viene creata una corrispondente entry DNS, nella forma:

```
<service-name>.<namespace-name>.svc.cluster.local
```

Questo significa che se un container utilizza <service-name>, finirà per utilizzare **il servizio del suo namespace locale**. Questo è utile per utilizzare la stessa configurazione su più namespace, ad esempio in ambiente di sviluppo, staging e produzione.
Per raggiungere risorse in un namespace diverso bisogna utilizzare il fully qualified domain name (FQDN).

Capitolo 5

IMMAGINI DEI CONTAINER

5.1 GESTIONE IMMAGINI

Una volta creata l'immagine Docker del container, è necessario farne una **push in un registro** prima di usarla in Kubernetes.
È possibile utilizzare **registri privati**, che richiedano **credenziali** per accedere alle immagini, ogni cloud vendor ha una diversa procedura di recupero delle stesse.
Le proprietà dell'immagine hanno la stessa sintassi dei comandi Docker, inclusi proprietà ed etichette.

La politica di default per il reperimento, quindi anche aggiornamento, delle immagini è `IfNotPresent` (se non presente) ed è configurabile.

5.2 CONTAINER ENVIRONMENT

Il container environment fornisce risorse importanti ai container:

- un **filesystem**, che è la combinazione di un immagine con uno o più volumi
- informazioni riguardo al **container stesso**
- informazioni riguardo ad **altri oggetti** nel cluster

5.2.1 Info sui Container

Il **hostname** di un container è il nome del POD in cui il container esegue. Il nome del POD ed il suo namespace sono disponibili come **variabili d'ambiente** grazie ad un'API "discendente".
Variabili d'ambiente create dagli utenti nella definizione del POD, sono disponibili anche al Container, come tutte le variabili d'ambiente specificate staticamente in un immagine Docker.

5.2.2 Info sul Cluster

Quando un container viene creato, gli viene fornita una **lista di tutti i servizi disponibili** in quel momento, come variabili di ambiente. Per un servizio (es. foo) mappato in un container (es. bar), esistono 2 variabili:

- FOO_SERVICE_HOST=<l'host su cui esegue il servizio>
- FOO_SERVICE_PORT=<la porta su cui risponde il servizio>

I servizi hanno IP dedicati e sono **disponibili al container via DNS**, se abilitato.

5.3 CLASSE RUNTIME

È una feature per la selezione della configurazione runtime di un container, utilizzata per istanziarlo in un POD.

5.3.1 Motivazione

Si puo impostare una Classe Runtime differente per ogni POD per creare un bilanciamento tra sicurezza e prestazioni. Per esempio, un container che ha bisogno di un alto livello di sicurezza, si può schedulare in quei POD che utilizzano la **virtualizzazione hardware** ottenendo un **maggiore overhead** ed un **maggiore isolamento**.

Oppure, per istanziare lo **stesso container** ma con **settaggi differenti** sullo stesso POD.

5.4 CONTAINER LIFETIME HOOKS

I container gestiti da Kubelet possono utilizzare il `Container Lifecycle Hook framework` per eseguire codice in base ad **eventi generati** dalla gestione del **loro ciclo vitale**. Esistono due hook per i container:

- **PostStart**: che esegue immediatamente dopo la creazione del container
- **PreStop**: Chiamato immediatamente prima che un container termini a causa di una chiamata API, oppure di un evento di gestione.

Un hook viene sfuttato con l'implementazione di un handler dedicato, ne esistono 2 tipi per container:

- **Exec**: esegue un comando specifico (es. preStop.sh), all'interno del contesto del container, utilizzandone le risorse.
- **HTTP**: esegue una richiesta HTTP in uno specifico punto di uscita di un container

Le chiamate agli hook handler sono **sincrone** all'interno del POD di contesto, questo significa che per un `PostStart` hook, l'**ENTRYPOINT del container** (punto/momento di partenza del codice) ed **evento di hook** vengono inizializzati in maniera **asincrona**.
In ogni caso, prima di entrare nello stato di esecuzione (running) il container deve **attendere la terminazione** dell'hook handler.

Similmente, per un `PreStop` hook, se l'handler esegue all'infinito, il POD rimane in uno stato di terminazione e viene quindi eliminato dopo un certo periodo, definito dalla proprietà `terminationGracePeriodSeconds`.

Se un hook **handler fallisce**, il **container termina**, anche senza aver mai cominciato ad eseguire. Nel caso, lo stesso handler invierà in broadcast un evento (PopStart: FailedPostStartHook, PreStop: FailedPreStopHook) visibile tramite kubectl.
Gli handler dovrebbero essere sempre il più leggeri possibile.

Capitolo 6

I POD

Un POD è l'unità più piccola e semplice del object model di Kubernetes; rappresenta un insieme di **processi in esecuzione** nel cluster.

Un POD incapsula uno o più container, un'applicazione, memoria di massa, una rete IP, dei settaggi su come il container dovrebbe essere gestito e offre un solo host logico per tutti i container. Rappresenta quindi un'**unità di deployment**.

Il contesto condiviso offerto dal POD corrisponde all'insieme dei namespace Linux, ai cgroups e ad altri elementi isolati; le stesse cose che isolano un container.

I POD sono utilizzati in 2 modi principalmente:

- Eseguono **un solo container**: si può pensare al POD, in questo caso, come ad un **wrapper** per il container, dove Kubernetes gestisce il POD e non direttamente il container.
- Eseguono **più container**: questi tipicamente lavorano insieme (tightly coupled) e hanno bisogno di **condividere risorse**. Ad esempio, alcuni container potrebbero fornire all'esterno dei file ospitati in un volume condiviso, mentre un altro ne aggiorna o modifica i dati.

Ogni POD può eseguire una sola istanza di una certa applicazione e può scalare orizzontalmente utilizzando delle sue **repliche**. POD replicati vengono **creati e gestiti come un gruppo** da un **controller**.

I container di un POD sono automaticamente **schedulati e localizzati in gruppi** nello stesso nodo, possono condividere risorse e dipendenze, comunicano tra loro e si coordinano anche sulla terminazione.
Ogni POD riceve un indirizzo IP ed ogni container all'interno ne utilizza lo **stesso namespace di rete**, quindi anche IP e porte disponibili. I container utilizzano l'interfaccia di loopback **localhost per comunicare tra loro**, mentre per comunicare all'esterno è necessario gestire l'utilizzo delle risorse di rete condivise.

I POD non sono creati per durare; non sopravvivono ad un errore di scheduling, del nodo oppure alla mancanza di risorse. Anche per questo è sempre meglio utilizzare i controller per istanziare i POD e non crearli direttamente.

Utilizzo

Potenzialmente un POD può ospitare uno stack verticale di applicazioni (es. LAMP), ma normalmente si istanziano applicazioni a microservizi, quali:

- sistema di gestione di contenuti, manager di cache locale, gestore di file
- log e gestione backup, compressione, rotazione, snapshot
- pubblicatore di eventi, monitoraggio, polling di dati
- server proxy, bridge, adattatori di rete
- controller, manager, configuratore, gestione degli aggiornamenti

6.1 CICLO DI VITA DEI POD

6.1.1 Fasi

Un oggetto status di un POD contiene il campo `phase`, un astrazione di alto livello sulla fase in cui si trova rispetto al suo ciclo di vita, intesa come **stato globale** del POD. Possibili fasi (valori assegnabili a phase):

- **Pending**: POD accettato dal sistema Kubernetes ma senza container istanziati. Può essere il caso in cui si attende la schedulazione, oppure mentre vengono scaricate le immagini dei container.
- **Running**: POD stanziato su di un nodo, tutti i container sono stati creati e ne esiste almeno uno in esecuzione, avvio o riavvio.
- **Succeded**: tutti i container di un POD hanno terminato l'esecuzione con successo e non saranno riavviati.
- **Failed**: tutti i container di un POD hanno terminato l'esecuzione, almeno uno di questi ha però fallito. Quindi, o esce dall'esecuzione ritornando uno stato diverso da 1 o è stato terminato dal sistema.
- **Unknown**: per qualche ragione lo stato del POD non è ottenibile, tipicamente il problema è dovuto ad un errore nelle comunicazioni con l'host del POD.

6.1.2 Condizioni

Lo status di un POD contiene anche un array chiamato `PodCondition`, composto di 6 campi, ognuno dei quali rappresenta il risultato di test effettuati sul POD.

- `lastProbetime`: timestamp dell'ultima verifica
- `lastTransitionTime`: timestamp dell'ultima transizione da uno stato all'altro
- `message`: contiene dettagli riguardo alla transizione
- `reason`: descrive la condizione dell'ultima transizione con una parola
- `status`: una stringa, può valere "True", "False" oppure "Unknown"
- `type`: stringa che contiene uno tra questi valori:
  - `PodScheduled`
  - `Ready`: POD in grado di ricevere richieste, dovrebbe venire aggiunto alle liste utilizzate dai meccanismi di load balancing
  - `Initialized`
  - `ContainersReady`

6.1.3 Probes (sonde)

Un probe è una attività diagnostica performata periodicamente dal kubelet di un container. Ogni probe può avere uno di questi risultati: **Success**, **Failed** oppure **Unknown** (fallimento della diagnostica). Kubelet può opzionalmente eseguire e reagire a 3 tipi di probe sui container:

- `livenessProbe`: indica se il container si trova in esecuzione, in caso di fallimento kubelet termina il container, che si riavvia.
- `readinessProbe`: pronto per ricevere richieste. Se fallisce, il controller lo rende irragiungibile eliminando il suo IP dagli altri container.
- `startupProbe`: indica se l'applicazione all'interno di un container è avviata. Se fornito dal container, disabilita tutti gli altri probe fino alla sua terminazione. Se fallisce, kubelet termina il container che si riavvia.

Un probe utilizza un handler implementato dal container, di cui esistono 3 tipi:

- `ExecAction`: esegue uno specifico comando all'interno del container. Se restituisce 0 il probe viene considerato superato.
- `TCPSocketAction`: esegue un controllo TCP sull'IP del container su una porta specifica. Se la porta è aperta, probe superato.
- `HTTPGetAction`: esegue una richiesta HTTP GET sull'IP del container su una specifica porta, con un dato path. Se lo status della risposta è superiore o uguale a 200 ed inferiore a 400 allora il probe è superato.

6.1.4 Stato del container

Quando un POD viene assegnato ad un nodo dallo scheduler, kubelet inizia a creare i container utilizzando il container runtime. Un container si trova sempre in uno di questi 3 **stati**: Waiting, Running e Terminated. Per verificare lo stato si può utilizzare kubectl per leggere la proprietà `POD_NAME`.
Un container, se non è in running o terminated, è in waiting, stato nel quale continuerà a eseguire operazioni. Insieme allo stato esiste una propietà `reason` che aggiunge informazioni riguardo alla causa.

6.1.5 Restart policy

Per i container di un POD si può impostare un comportamento al riavvio grazie alla restart policy, specificando in quali **casi**: `Always`, `OnFailure` oppure `Never`. Al riavvio, nel caso, viene aggiunto un ritardo nell'istanziazione che cresce esponenzialmente (10s, 20s, 40s..) e raggiunge un massimo di 5 minuti, resettato dopo 10 minuti dall'ultima esecuzione che ha avuto successo.

6.1.6 Lifetime

In generale la durata di un POD non viene decisa prima, rimangono finchè non sono esplicitamente rimossi. Il piano di controllo ripulisce i POD terminati e recupera le risorse.

6.2 INIT CONTAINER

Sono container specializzati per l'inizializzazione, questi eseguono prima di quelli dell'applicazione nel POD e solitamente contengono degli script per l'installazione di dipendenze o altri software; **preparano il contesto**.

Gli init container eseguono sempre fino al completamento e tutti devono terminare con successo prima che venga eseguito il prossimo.
Se un init container fallisce , Kubernetes riavvia ripetutamente il POD fino a che non termina con successo. Questo comportamento si può impedire con la `restartPolicy` **Never**.

6.2.1 Differenze dai container normali

Rispetto ai container normali viene gestita diversamente la richiesta di risorse, inoltre non supportano le probe di Readiness. Se si specificano più init container per un POD, Kubelet li esegue sequenzialmente. Quando tutti gli init container hanno terminato con successo, Kubelet inizializza l'applicazione.

6.2.2 Utilizzo

Alcuni esempi:

- Possono contenere delle librerie o dei tool non presenti nei container delle applicazioni, se utili durante il setup.
- Il costruttore ed il deployer dell'immagine dell'applicazione possono lavorare indipendentemente
- Accesso diverso del file system: possono accedere a segreti, durante il setup, non accessibili all'applicazione.
- Possibilità di verificare delle precondizioni.
- Si possono utilizzare per eseguire controlli di sicurezza, in contesti separati, limitando gli attacchi.

6.2.3 POD Presets

I POD Presets sono una risorsa API per l'aggiunta di requisiti runtime addizionali in un PO D al momento della creazione. Questo meccanismo permette agli autori di POD di non fornire tutte le informazioni necessarie, quindi di non dover conoscere tutti i dettagli riguardo al servizio.

Kubernetes fornisce un controller di ammissione (`PodPreset`) che, se abilitato, applica delle regole alle richieste di creazione dei POD. Vengono utilizzati i selettori di etichette per scegliere i POD a cui applicare le regole.

6.3 VINCOLI SULL'ESPANSIONE DELLA TOPOLOGIA DEI POD

Funzionalità in fase **beta** sulla versione 1.18 di Kubernetes.

Funzione di creazione di vincoli per il deployment dei POD del cluster attraverso molte zone soggette ad errori, quali regioni, zone, nodi e altre topologie user-defined. Le etichette dei nodi li identificano rispetto al dominio in cui si trovano.

Supponiamo di avere un cluster di 4 nodi con queste etichette:

```
NAME STATUS ROLES AGE VERSION LABELS
node1 Ready <none> 4m26s v1.16.0 node=node1,zone=zoneA
node2 Ready <none> 3m58s v1.16.0 node=node2,zone=zoneA
node3 Ready <none> 3m17s v1.16.0 node=node3,zone=zoneB
node4 Ready <none> 2m43s v1.16.0 node=node4,zone=zoneB
```

Allora il cluster sarà logicamente inteso come:

```
        zoneA: node1, node2
        zoneB: node3, node4
```

Il campo `pod.spec.topologySpreadConstraints` è stato introdotto nella versione 1.16, utilizzo:

```
apiVersion: v1
kind: Pod
metadata:
        name: mypod
spec:
        topologySpreadConstraints:
        - maxSkew: <integer>
                topologyKey: <string>
                whenUnsatisfiable: <string>
                labelSelector: <object>
```

Si possono definire multipli vincoli per istruire il kube-scheduler su come posizionare ogni POD in arrivo in relazione all'esistenza di altri POD nel cluster.

6.4 INTERRUZIONI NELL'ESECUZIONE

Un POD non scompare finchè non viene cancellato da un utente, oppure nel caso di errori hardware o software inevitabili (incluso l'esaurimento di risorse). Questi ultimi vengono chiamati interruzioni **involontarie**, i primi **volontarie**. Cosa si può fare per mitigare le interruzioni:

- Assicurare le risorse necessarie

- **Replicare** le applicazioni
- **Distribuire** le repliche su più macchine fisiche e su più zone

Deve essere previsto un "**budget di interruzioni**" per coprire sia quelle volontarie, fatte principalmente per manutenzione o aggiornamenti, che quelle accidentali o involontarie.

Con l'utilizzo del budget di interruzioni torna molto utile eseguire una separazione delle responsabilità tra il **proprietario del cluster** e il **proprietario dell'applicazione**. Sopratutto quando ci sono molti team che condividono un cluster ed esiste una naturale specializzazione nei ruoli e quando i tool e i servizi di terze parti sono utilizzate per automatizzare la gestione del cluster.
Il Pod `Disruption Budget` supporta questa separazione di ruoli fornendo un interfaccia dedicata.

6.5 CONTAINER EFFIMERI

Funzionalità in fase pre-alpha, i Container Effimeri eseguono temporaneamente in un POD già esistente per ispezionare i servizi, principalmente, oppure l'analisi di **debug**.

Questi container mancano di garanzie sulle risorse assegnate, non verranno mai riavviati in automatico e quindi non sono appropriati per costruire applicazioni. Vengono descritti come container normali, ma molti campi non sono riconosciuti oppure vietati: non possono avere porte (no campi: `port`, `livenessProbe`, `readinessProbe`), dato le risorse immutabili `resources` non è permesso.

Capitolo 7

I CONTROLLER

In Kubernetes, i controller agiscono come **cicli di controllo infiniti**; verificano continuamente lo stato del cluster, causando cambiamenti se necessario. Ogni controller ha il compito di rendere lo stato attuale il **più simile possibile** a quello desiderato.

Nota: nel capitolo sugli oggetti Kubernetes, si spiega che i controller vengono compilati insieme come un unico eseguibile e istanziati sul kube-controller-manager, processo che fa parte del piano di controllo ospitato in un nodo definito master.

7.1 CONTROLLER PATTERN

Un controller traccia almeno un tipo di risorsa Kubernetes, questi oggetti hanno un campo spec che rappresenta lo stato desiderato. Il controller può eseguire il **cambiamento** necessario da solo; più spesso però, invierà un **messaggio all'API server**.

I **controller built-in** tipicamente modificano lo stato interagendo con l'API server del cluster ed offrono importanti **funzionalità core** di Kubernetes (Es. deployment e job), mentre i controller all'esterno del piano di controllo estendono Kubernetes. I controller inoltre, **aggiornano gli oggetti che configurano**.

I controller che interagiscono con lo stato esterno ricevono lo stato deside-

rato dall'API server, poi comunicano direttamente con il sistema esterno per avvicinare lo stato attuale a quello desiderato (es. aggiunta e quindi utilizzo di un nuovo nodo).

Il meccanismo di stato attuale e desiderato permette a Kubernetes di gestire un **cambiamento costante** nel sistema. In qualunque momento, il cluster potrebbe essere costretto a modificarsi ed i controller se ne occuperebbero immediatamente.
Questo significa che potenzialmente il cluster non raggiungerà **mai uno stato stabile**, il che, fintanto che i controller sono attivi, **non è un problema**.

È più facile gestire **tanti controller semplici** piuttosto che uno, monolitico e interconnesso con molti controlli logici. Ci possono esser molti controller che creano o aggiornano lo **stesso tipo di oggetti**, ma si assicurano tutti di modificare solo gli oggetti collegati a loro stessi.

Il piano di controllo è **resiliente** per cui se un controller built-in va in crash, un'altra parte del piano di controllo si occupa del suo lavoro.

7.2 CONCETTI SUL CLOUD CONTROLLER MANAGER

Il meccanismo **CCM** è stato creato per permettere al codice del cloud provider e a Kubernetes di evolversi indipendente uno dall'altro. Il CCM è in esecuzione a fianco di altri componenti del master (Es. controller manager, API server e scheduler), se viene eseguito come addon però, esegue al di sopra/fuori di Kubernetes.

Il CCM permette ai nuovi cloud provider di **integrare Kubernetes facilmente** utilizzando plugin. Siamo in un momento di passaggio, in cui gli sviluppatori di Kubernetes stanno cercando di portare i vari cloud provider verso questo modello di integrazione, basato su CCM.

Nella figura, Kubernetes ed il cloud provider sono integrati attraverso **molti componenti**: kubelet su ogni nodo, controller manager e API server. Il CCM semplifica la logica dipendente dal cloud, raccogliendola in un **singolo punto di integrazione**.

Il cloud controller manager si prende carico di alcune funzionalità del

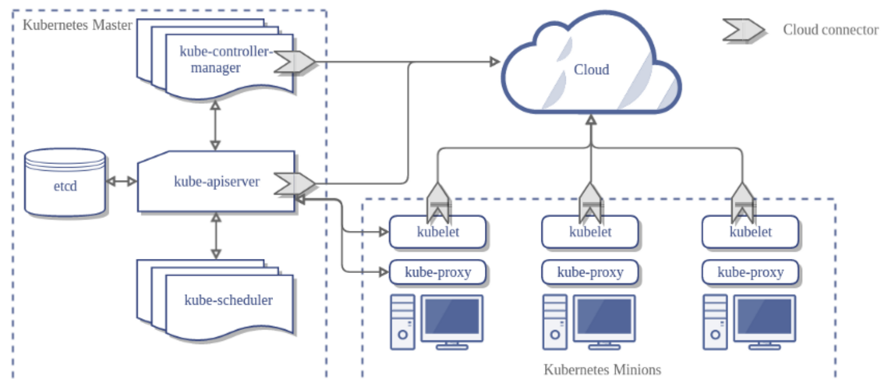

Figura 7.1: Architettura di un cluster Kubernetes **senza** il CCM

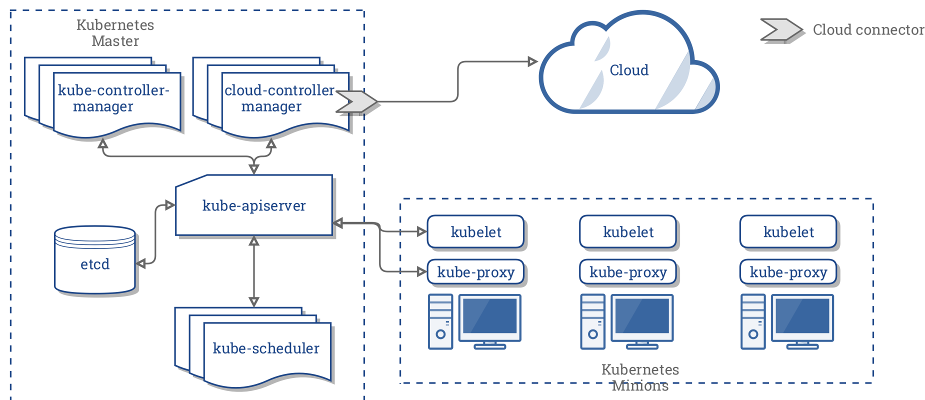

Figura 7.2: Architettura di un cluster Kubernetes **con** il CCM

controller manager di Kubernetes (KCM), in particolare quei controller che sono particolarmente dipendenti dal cloud: **Node**, **Route**, **Service**.
Il Volume controller **non è parte** del CCM sia per la sua complessità ma anche per astrarre dagli storage specifici dei cloud vendor.

7.3 REPLICA SET

L'obbiettivo di un controller `ReplicaSet` è di **mantenere un insieme di POD** di replica stabili e in **esecuzione** per tutto il tempo necessario, viene anche utilizzato per garantire la disponibilità di un numero specifico di POD

identici.

Un ReplicaSet possiede campi che descrivono quali POD può acquisire, il numero di repliche totali che deve gestire ed un pod template dei POD che deve creare.
Il ReplicaSet è collegato ai suoi POD attraverso un link che questi ultimi possiedono nel campo `metadata.ownerReferences`, che, in generale, specifica quale risorsa possiede l'oggetto corrente. È grazie a questo link che i ReplicaSet sanno in che stato sono i loro POD.

I ReplicaSet identificano i nuovi POD da acquisire grazie ai selettori; se un POD non possiede un `OwnerReference` (oppure questo non è un controller) e rientra nella ricerca, verrà immediatamente acquisito.

7.3.1 Quando usare i ReplicaSet

Esiste un oggetto di livello più alto che fa lo stesso lavoro: il **Deployment**. Normalmente è sufficiente, ma nel caso si voglia aggiungere una personalizzazione dell'orchestrazione sugli aggiornamenti è raccomandato il ReplicaSet.

Una combinazione di Deployment e ReplicaSet è preferibile rispetto alla scelta progettuale precedente di Kubernetes, il **Replica Controller**.

7.4 DEPLOYMENT

Il `Deployment` fornisce aggiornamenti dichiarativi per i POD ed i ReplicaSet, è sufficiente descrivere lo stato desiderato nella logica di Kubernetes.
Una volta schedulati, i POD diventano parte dello stato desiderato per un processo kubelet.
Un Deployment è in grado, oltre ad istanziare ReplicaSet, di cancellare oggetti uguali già esistenti e di adottarne tutte le risorse con nuovi Deployments.

7.4.1 Casi d'uso

- Per un lancio di più ReplicaSet.

- Dichiarare il nuovo stato dei POD aggiornando il `PodTemplateSpec` del Deployment. Viene creato un nuovo ReplicaSet e i POD vengono spostati dal vecchio al nuovo gradualmente. Ogni ReplicaSet aggiorna lo stato nel Deployment.
- Tornare indietro ad una precedente versione (rollback), se lo stato corrente non è stabile.
- Scalare il Deployment per rispondere al carico di lavoro
- Mettere in pausa il Deployment per applicare molte correzioni del PodTemplateSpec e ripartire dall'ultima versione dei POD.
- Utilizzare lo stato del Deployment come indicatore di lancio di POD bloccato.
- Eliminare un vecchio ReplicaSet di cui non c'è più bisogno

7.4.2 Esempio

```
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
```

Nell'esempio viene descritto un deployment chiamato `nginx-deployment`, questo deve istanziare 3 POD replicati (campo `replicas`) mentre il campo `selector` definisce l'etichetta con cui selezionare i POD (`app: nginx`)

Mentre il cluster è attivo ed in esecuzione, si può istanziare il deployment con questo comando:

```
kubectl apply -f <path-to>/nginx-deployment.yaml
```

7.4.3 Rollback

Un deployment non stabile ha dei comportamenti indesiderati, ad esempio va in crash in loop, riavviandosi continuamente. Per default, viene salvato in memoria uno **storico di tutti i lanci di POD** cosi da poter ritornare ad una **versione precedente** in qualsiasi momento.

Controllare lo storico:

```
kubectl rollout history deployment.v1.apps/nginx-deployment
```

Output di esempio:

```
deployments "nginx-deployment"
REVISION CHANGE-CAUSE
       1 kubectl apply --filename=<path>/nginx-deployment.yaml \
                              --record=true
       2 kubectl set image deployment.v1.apps/nginx-deployment \
                             nginx=nginx:1.16.1 --record=true
       3 kubectl set image deployment.v1.apps/nginx-deployment \
                             nginx=nginx:1.161 --record=true
```

L'ultimo aggiornamento è quello nella terza riga, dove la versione è stata scritta male, 1.161 invece di 1.61.1, quindi per tornare alla revisione 2, che è quella con la versione corretta:

```
kubectl rollout undo deployment.v1.apps/nginx-deployment
```

Oppure, per tornare ad una versione specifica:

```
kubectl rollout undo deployment.v1.apps/nginx-deployment \
       --to-revision=2
```

7.4.4 Scalare un Deployment

```
kubectl scale deployment.v1.apps/nginx-deployment --replicas=10
```

Oltre ad impostare staticamente il numero di replica, si può utilizzare una funzione di autoscaling, dando il numero minimo e massimo di queste, il controller si baserà sull'utilizzo della CPU fatto dai POD esistenti per decidere il numero effettivo di repliche.

```
kubectl autoscale deployment.v1.apps/nginx-deployment \
       --min=10 --max=15 --cpu-percent=80
```

7.4.5 Scrivere le specifiche di un Deployment

Come tutti i file di configurazione di Kubernetes, un Deployment deve avere i campi `apiVersion`, `kind`, `metadata` ed una sezione `.spec`. Il nome del Deployment deve essere valido nel sottodominio DNS.

I campi `.spec.template` e `.spec.selector` sono gli unici richiesti in `.spec`.
Il primo ospita il template di un POD, possiede lo stesso schema senza `apiVersion` e `kind`. Il numero di repliche è opzionale, di default è 1.
Il secondo, `.spec.selector`, indica l'etichetta affibbiata al POD, utilizzata nelle ricerche da parte dei controller. Questo campo deve essere identico a `.spec.template.metadata.label` o verrà rifiutato dalle API.

7.4.6 RollingUpdate

Per utilizzare questa funzione è necessario impostare questo campo:
`.spec.strategy.type==RollingUpdate`. Si può impostare il numero di `maxUnavailable` e `maxSurge`.

- **max unavailable**: specifica il numero massimo di POD che possono essere non disponibili durante un processo di aggiornamento, descritto da un numero assoluto o una percentuale.

- **max surge**: il numero massimo di POD creabili con cui si può sforare il numero di POD desiderati. Anche in questo caso, numero assoluto o percentuale dei POD desiderati.

7.5 STATEFULL SET

Uno `StatefullSet` è un oggetto dotato di **API per il carico di lavoro**, utilizzato per gestire applicazioni statefull.
Gestisce il deployment e lo scalare di un set di POD, inoltre fornisce garanzie sull'ordinamento e unicità degli stessi. I POD devono essere basati sullo stesso container, lo StatefullSet mantiene un'identità per ognuno di essi, quindi anche se molto simili non sono intercambiabili.

Statefull set si utilizza se l'applicazione richiede:

- identificatori di rete unici e stabili
- storage stabile e persistente
- deployment e scaling ordinato
- rolling updates automatizzati ed ordinati

Per **stabile** si intende **persistente** rispetto al ri-scheduling dei POD. Se l'applicazione non richiede queste proprietà, si istanziano repliche stateless con oggetti quali Deployment e ReplicaSet.

7.5.1 Limitazioni

- lo storage di un certo POD deve essere offerto da un oggetto `PersistentVolume Provisioner` oppure pre-fornito dall'admin
- cancellare o scalare uno StatefullSet non cancellerà i volumi associati
- StatefullSet richiede un **Servizio Headless** per la gestione dell'identità dei POD in rete, servizio da creare a parte.
- StatefullSet non garantisce la terminazione dei POD quando viene cancellato. Per ottenere una terminazione ordinata dei POD collegati, è possibile scalare lo StatefullSet a 0 prima di cancellarlo.

- L'utilizzo di Rolling Update con la gestione di default (`OrderedReady`) potrebbe portare ad un errore sullo stato che richieda l'intervento manuale dell'admin.

7.6 DAEMON SET

Il suo compito è assicurarsi che sui nodi designati siano presenti precise copie di POD contenenti dei daemon (programmi in background). Eliminare un DaemonSet cancellerà anche i POD che ha creato. Compiti tipici dei daemon:

- `cluster storage` (es. `glusterd`, `ceph`)
- gestione dei log su tutti i nodi (es. `fluentd`, `filebeat`)
- monitoraggio su tutti i nodi (es. `Prometheus Node Exporter`, `Flowmill`. `collectd` ecc.)

In un caso semplice, un `DaemonSet` coprirebbe tutti i nodi gestendo tutti i tipi di daemon. In un setup più complesso si potrebbero utilizzare più DaemonSet per ogni tipo di daemon, ma con differenti flag e/o differenti risorse da hardware diversi.

7.7 GARBAGE COLLECTION

7.7.1 Oggetti proprietari e dipendenti

Alcuni oggetti Kubernetes sono proprietari di altri oggetti, ad esempio un ReplicaSet possiede dei POD. Si parla in questo caso di **dipendenza** verso il proprietario, che viene indicato negli oggetti posseduti da un campo `metadata.ownerReferences`.

Nella maggior parte dei casi Kubernetes gestisce il riferimento al proprietario autonomamente, specialmente quando un controller istanzia dei POD, rimane possibile specificare manualmente il valore di `ownerReferences`.

7.7.2 Policy di cancellazione degli oggetti dipendenti

Si può specificare se estendere la cancellazione agli oggetti dipendenti quando il proprietario viene eliminato, altrimenti gli oggetti rimangono orfani. Questo meccanismo è chiamato `cascading deletion` (**cancellazione a cascata**) e ne esistono 2 modelli:

- **foreground**: l'oggetto alla radice entra in uno stato detto "deletion in progress", il garbage collector inizia a cancellare i suoi oggetti dipendenti, quando ha cancellato tutti quelli che bloccano il proprietario (`ownerReference.blockOwnerDeletion`=true, campo settato in automatico) cancella anche il proprietario.
- **background**: il garbage collector cancella immediatamente il proprietario e poi gli oggetti dipendenti, che diventano quindi temporaneamente orfani.

7.8 TTL CONTROLLER PER RISORSE FINITE

Questo controller è ancora in **fase alpha**, si occupa di limitare il tempo di vita di un oggetto risorsa che ha terminato l'esecuzione assegnandogli un Time To Live (TTL). Per ora gestisce solamente i Job, ma può essere esteso per gestire altre risorse, come i POD.

Si utilizza per ripulire i Job terminati automaticamente impostando il campo `.spec.ttlSecondsAfterFinished`. Naturalmente questo campo può essere modificato a runtime, quindi la scadenza di un oggetto può essere estesa.

7.9 I JOB

Un Job crea uno o più POD e si **assicura** che un certo numero di questi termini con **successo**. Eliminare un Job cancellerà tutti i POD che ha creato.

Dato che un POD può fallire per molte ragioni, un'**applicazione** istanziata con un Job deve prevedere la **gestione del riavvio** in un POD nuovo; attività quali la gestione dei file temporanei, dei lock a runtime, output incompleti e procedure lasciate a metà da istanze precedenti.

Per definire un numero massimo di tentativi si imposta il campo `.spec.backoffLimit`, di default il suo valore è 6. Questo per impedire loop basati su errori di configurazione e situazioni stagnanti.

Al completamento del Job, il controller ed i suoi POD rimangono esistenti, cosi da mantenere i log. Sta all'utente cancellare i Job terminati, utilizzando kubectl.

I Job sono utilizzati per rendere affidabili POD in esecuzione parallela, con **flussi di lavoro indipendenti**: ad esempio l'invio di email, rendering di frame, file da tradurre/compilare, serie di query da eseguire ecc.

7.10 CRONJOB

Un controller che crea Job sulla base di uno scheduler basato sul tempo (UTC).

7.10.1 Limitazioni

Un CronJob crea circa un oggetto Job per ogni esecuzione schedulata, esistono delle circostante per cui vengono creati 2 oggetti oppure nessuno, anche se rare. Perciò, i Job dovrebbero essere idempotenti.

Il CronJob controller verifica, per ogni CronJob, quante schedulazioni ha saltato dall'ultima esecuzione, se sono più di 100 non avvia il Job e fa il log con errore:

```
Cannot determine if job needs to be started. Too many missed
start time (> 100). Set or decrease
 .spec.startingDeadlineSeconds or check clock skew.
```

Il controller CronJob è responsabile solamente della creazione dei Job schedulati, è compito del Job gestire i POD definiti nel suo manifest.

Capitolo 8

SERVIZI, LOAD BALANCING E NETWORKING

8.1 SERVIZI

Un'astrazione per esporre un'**applicazione** in esecuzione su di un insieme di POD come **servizio di rete**.
Kubernetes fornisce ad ogni POD un IP diverso e a tutti un DNS comune, inoltre gestisce il carico di lavoro distribuito.

8.1.1 Motivazioni

Un POD ottiene un **IP diverso** quando viene riavviato, quindi, per fare in modo che un insieme di POD di "backend" possano offrire funzionalità a POD di "frontend", si utilizzano i servizi per **disaccoppiarli**.

Un servizio abbina ad un insieme di POD una policy di accesso, questo pattern è chiamato **micro-servizio**.

8.1.2 Definire un servizio

In Kubernetes, un servizio è un **oggetto REST**, che può essere pubblicato sull'API server per crearne un istanza.
Ad esempio, se esistono un insieme di POD che ascoltano sulla porta TCP 9376 a sono etichettati `app=MyApp`:

```
apiVersion: v1
kind: Service
metadata:
        name: my-service
spec:
        selector:
                app: MyApp
        ports:
                - protocol: TCP
                port: 80
                targetPort: 9376
```

Questa specifica crea un nuovo oggetto servizio chiamato "`my-service`", sulla porta TCP 9376 di tutti i POD etichettati "`app=MyApp`".
Il Controller del servizio esegue un polling continuo alla ricerca di POD con questa etichetta, per poi pubblicare tutti gli aggiornamenti in un oggetto `Endpoint` sempre chiamato "`my-service`".

Si possono definire più porte per servizio, differenti protocolli di comunicazione e assegnare nomi alle porte dei POD.

8.1.3 Servizi senza selettori

Un servizio generalmente astrae l'accesso ad un POD, ma possono astrarre anche altri tipi di backend:

- un cluster di database esterno in produzione ed uno diverso nell'ambiente di test
- un servizio che punta ad un altro situato in un Namespace differente, o su un altro cluster

- mentre viene migrata un'applicazione su Kubernetes, si esegue solamente una parte del backend

L'alternativa ai selettori è definire manualmente un oggetto `Endpoint`, da cui si accede al servizio, grazie al riferimento `IP:Porta`.

8.1.4 IP Virtuali e proxy di servizi

Il processo `kube-proxy` presente in ogni nodo, è responsabile per l'implementazione di un **IP virtuale** per servizi diversi da `ExternalName` (servizi su altri cluster, puntati da un DNS).
Si tenga presente che un cluster Kubernetes è pensato per scalare enormemente e che le interazioni fra microservizi sono automatizzate, perciò brevi ma numerose.

Perchè si utilizzano **i proxy** per il traffico interno?

- i tentativi di utilizzo di un DNS con risoluzione round-robin non rispettavano i TTL dei record, l'utilizzo di cache risulta in un ritorno di **nomi già scaduti**
- alcune applicazioni fanno **caching** del risultato DNS
- anche se app e librerie implementassero ricerche DNS appropriate, il bassissimo TTL dei record del DNS provocherebbe dei **carichi di lavoro enormi** sul DNS server

8.1.5 Tipi di modalità proxy

- User space in proxy mode

  In questa modalità, kube-proxy osserva il master Kubernetes per l'aggiunta o rimozione di oggetti, servizi ed Endpoint. Per ogni servizio apre una porta (scelta casualmente) sul nodo locale. Tutte le **connessioni a** questa "**porta proxy**" sono **inoltrate** ad uno dei POD di servizio backend (via Endpoint).

  Inoltre, il proxy user-space instaura delle **regole per le iptables**: queste catturano il traffico per il cluster IP:porta (virtuale) del servizio e

lo inoltrano sulla porta del proxy, che lo inoltra al POD di backend. Per default, si sceglie un POD backend secondo un algoritmo round-robin.

- IPtables in proxy mode

  Il kube-proxy osserva il piano di controllo di Kubernetes per l'aggiunta e rimozione di oggetti servizi ed Endpoint. Per ogni servizio installa regole di iptable, che catturano il traffico verso il cluster IP:porta del servizio e lo ridirige ad uno dei backend dei servizi. Per ogni oggetto Endpoint, installa delle regole iptable per selezionare un POD backend, scelto casualmente.

  L'utilizzo di iptables per gestire il traffico produce un minor hoverhead, perchè il traffico è gestito da netfilter di Linux senza il bisogno di cambiare da userspace a kernel space. Questo approccio è anche più affidabile.

  Se kube-proxy esegue in **modalità iptable** ed il primo POD selezionato non risponde, la **connessione termina**. In modalitò **userspace** invece, kube-proxy rileva la connessione fallimentare del primo POD e **ne sceglie automaticamente un altro**.
  Si può utilizzare il `readiness probe` di un POD per verificare che sia attivo, così da mostrare al kube-proxy in modalità iptable solamente i POD del backend che risultano in stato "healthy", evitando traffico inutile.

- IPVS in modalità proxy

  In modalità `IP Virtual Server`, il kube-proxy osserva i servizi ed gli Endpoint di Kubernetes, chiama l'interfaccia `netlink` per creare regole IPVS sincronizzate con i servizi e gli Endpoint di Kubernetes periodicamente. Questo loop di controllo si assicura che lo status IPVS sia uguale a quello desiderato. Sarà IPVS a redirigire il traffico ad un POD di backend.

  La modalità proxy IPVS utilizza una hash table come struttura dati sottostante e lavora in kernel space. Significa che il kube-proxy in IPVS redirige il traffico con **minore latenza** rispetto al kube-proxy in modalità iptable, con **performance molto migliori** quando le regole proxy vengono sincronizzate. Rispetto alle altre modalità, IPVS supporto anche un **maggiore throghput** di rete.

  Inoltre, ci sono più algoritmi di gestione disponibili per bilanciare il traffico: round-robin, connessioni minime, hashing delle destinazioni, hashing delle sorgenti, minor ritardo atteso, mai in coda.

Nota: la modalità IPVS va abilitatà sul nodo prima di avviare il kube-proxy.

In questi modelli proxy, il limite del traffico sull'IP:porta del servizio è dirottato ad un backend appropriato senza che il client sappia nulla su Kubernetes, i servizi o i POD.
Per assicurare il dirottamento di un certo client allo stesso POD backend ad ogni connessione, si può selezionare l'affinità di sessione basata sull'IP del client impostando il servizio `service.spec.sessionAffinity=ClientIP` (default "none").

8.1.6 Servizi Headless

Se non sono necessari meccanismi di load-balancing ed un singolo servizio IP, si utilizza un servizio headless. Impostando il campo `.spec.clusterIP=none` si evita di allocare un cluster IP, kube-proxy quindi **non gestisce i servizi** e non viene effettuato neanche il proxy di traffico.
Si utilizza per interfacciarsi con altri meccanismi di scoperta di servizi, senza essere legati all'implementazione di Kubernetes.

8.1.7 Cercare i servizi

Kubernetes supporta 2 modalità per la ricerca di servizi:

- **variabili d'ambiente**: quando un POD viene assegnato ad un nodo, kubelets crea una serie di variabili d'ambiente per ogni servizio attivo. In questo caso il servizio dovrà esistere prima del client, che altrimenti non troverà le variabili di riferimento.

- **DNS**: consigliato tramite add-on, un server DNS cluster-aware controlla l'API Kubernetes alla ricerca di nuovi servizi e crea un insieme di record DNS per ognuno. Se il DNS è abilitato su tutto il cluster allora tutti i POD dovrebbero utilizzarlo automaticamente. Questo sistema è l'unico utilizzabile per raggiungere servizi `ExternalName`, cioè esterni ad un cluster.

8.1.8 Pubblicare servizi

Kubernetes `ServiceTypes` pemette di specificare il tipo di servizio, il default è `ClusterIP`;

- `ClusterIP`: espone il servizio con un IP interno al cluster, unico posto dove sarà disponibile.
- `NodePort`: espone il servizio sull'IP di ogni nodo su di una porta statica (`NodePort`). Un servizio `ClusterIP` verrà creato automaticamente, sarà puntato dalla route del servizio `NodePort`. Si potrà contattare il servizio richiedendo <`NodeIp`>:<`NodePort`> dall'**esterno** del cluster.
- `LoadBalancer`: espone il servizio esternamente utilizzando un load balancer del cloud provider. `NodePort` e `ClusterIP` sono creati e puntati dal load balancer esterno automaticamente.
- `ExernalName`: mappa il servizio con i contenuti del campo `ExternalName`, ritornando un record `CNAME` con il suo valore. Nessun proxy impostato.

8.1.9 Scalare

L'utilizzo di **IP virtuali** (**VIP**) funziona in ambiente piccoli e medi,ma per scalare verso cluster molto grandi **non è sufficiente**. L'utilizzo di spazi di utenti proxy nasconde l'indirizzo IP sorgente di un pacchetto che accede ad un servizio; questo rende **funzionalità di firewall di rete impossibili**.

La modalità iptables proxy **non oscura** gli IP sorgenti all'interno del cluster.

Evitare collisioni significa non scegliere i numeri di porta se questa scelta può collidere con le scelte di altri. Se 2 sviluppatori di applicazioni utilizzassero la stessa porta in un nodo, la loro **applicazione** rimarrebbe **isolata**.

Per evitare queste situazioni, Kubernetes assegna ad **ogni servizio un diverso IP**. Per assicurare l'unicità dell'indirizzo, un allocatore interno aggiorna atomicamente una mappa di allocazione globale in etcd (base di dati) prima di creare ogni servizio.

Nel piano di controllo, un controller in background è responsabile per la creazione della mappa, inoltre controlla se esistono assegnamenti invalidi ed il recupero di indirizzi IP non più utilizzati.

Gli indirizzi IP di servizi, a differenza di quelli dei POD, non vengono restituiti da un singolo host. Invece, il kube-proxy utilizza le iptable per definire **IP virtuali** che sono transparentemente redirezionati al bisogno.

8.2 TOPOLOGIA DEI SERVIZI

La topologia dei servizi permette ad un servizio di gestire il traffico basandosi sulla topologia dei nodi del cluster. Per esempio, quando si effettua instradamento, si può specificare di preferire un backend sullo stesso nodo del client, o nella stessa zona di disponibilità.

8.2.1 Utilizzo

Se questa feature è abilitata nel cluster, si controlla il routing del traffico dei servizi impostando il campo `topologyKeys` nelle specifiche del servizio. Questo campo è una lista di preferenze ordinata le cui entry sono le etichette di nodi, utilizzate nello stesso ordine per scegliere gli endpoint quando si accede ad un servizio.

Se non si raggiunge una connessione, una volta processate tutte le entry, il traffico viene respinto esattamente come se non ci fossero istanze del backend in esecuzione.

8.2.2 Limiti

La topologia dei servizi non è compatibile con `externalTrafficPolicy=Local`, quindi un servizio non può utilizzare entrambe le feature.
Le **etichette** utilizzabili sono **limitate** a hostname, zona e regione attualmente, ma verranno aumentate.
Massimo 16 entry, di cui "*" deve essere l'ultimo.

8.3 ENDPOINTSLICE

Forniscono un metodo per tracciare gli endpoint di una rete all'interno di un cluster. Sono un'**alternativa agli endpoint maggiormente scalabile ed estensibile**.

8.3.1 Risorse degli EndpointSlice

Un EndpointSlice contiene **entry univoche servizio:porta** che puntano ad un insieme di endpoint in rete. Il controller EndpointSlice crea automaticamente questi riferimenti per un servizio quando viene specificato un selettore.

Per default, gli EndpointSlice gestiti da controller omonimi non possiedono più di 100 Endpoint ognuno. Sono supportati 3 tipi di indirizzi: IPv4, IPv6 e Fully Qualified Domain Name (FQDN).

Ogni endpoint all'interno di un EndpointSlice può contenere informazioni rilevanti sulla topologia, utili per indicare dove si trovi l'endpoint (hostname, zona e regione), tutti dati recuperati dal controller ed inseriti nelle sue etichette corrispondenti.

8.3.2 Controller EndpointSlice

Per default questo è l'oggetto che istanzia, gestisce e possiede un oggetto EndpointSlice. Possiede un insieme di porte da utilizzare, osserva i POD ed i servizi per rimanere aggiornato, gestisce EnpointSlice per tutti i servizi che specificano un selettore. questo per rappresentare l'indirizzo IP dei POD che fanno match con il selettore del servizio.

Il controller cerca di utilizzare il più possibile le porte disponibili, ma non ne bilancia l'utilizzo, applica una logica abbastanza semplice: itera gli EndpointSlice rimuovendo/aggiornando gli endpoint, itera di nuovo gli EndpointSlice modificati e aggiunge i nuovi necessari, se rimangono altri endpoint da aggiungere controlla se c'è posto negli EndpointSlice non modificati oppure ne crea di nuovi.

Il terzo passaggio limita l'aggiornamento degli EndpointSlice se perfettamente pieni. Con un kube-proxy in esecuzione su ogni nodo che osservano gli EndpointSlice, ogni modifica ad uno di questi diventa costoso da trasmettere a tutti i nodi nel cluster, potrebbe portare a non sfruttare a pieno gli EndpointSlice.

8.4 DNS PER SERVIZI E POD

Il DNS di Kubernetes è composto da un POD ed un servizio sul cluster, inoltre configura il kubelet per indicare il DNS ai singoli container.

Tutti i servizi definiti nel cluster (incluso lo stesso DNS server) possiedono un nome all'interno del dominio DNS. Quando un POD chiede la risoluzione di un nome al server, include anche il proprio namespace e dominio, per cui la ricerca sarà limitata a questi contesti.

8.4.1 Servizi

I servizi non headless ricevono un record DNS di tipo A oppure AAAA (da intendersi come IPv4 o IPv6), a seconda del gruppo IP del servizio, con un nome nella forma `my-svc.my-namespace.svc.cluster-domain.example`. La risoluzione del nome riporta l'IP del cluster del servizio.
Servizi headless (senza cluster) ricevono un record con le stesse regole, ma viene risolto restituendo l'insieme di IP dei POD selezionati dal servizio.

I record **SRV** vengono creati per le porte che hanno un nome e sono parte di servizi headless, nella forma:

```
_my-port-name._my-port-protocol.my-svc
       .my-namespace.svc.cluster-domain.example
```

.

8.4.2 POD

Quando un POD viene creato, il suo hostname è `metadata.name`, è possibile specificarne uno diverso ed anche un sottodominio opzionale.
Se esiste un servizio headless nello stesso namespace di un POD e con lo stesso nome di sottodominio, il DNS restituirà record A o AAAA per il fully qualified hostname del POD.

La politica DNS dei POD può essere impostata per ognuno di essi, aggiungendo un valore al campo **dnsPolicy**, ne esistono alcune versione standard supportate:

- `Default`: il POD eredità la configurazione DNS del nodo ospitante.
- `ClusterFirst`: le query DNS in cui manca il suffisso del dominio del cluster, vengono propagate al server più in alto nella gerarchia fornito dal nodo.
- `ClusterFirstWithHostNet`: se utilizzano la `hostNetwork` è obbligatorio
- `None`: ignora la configurazione DNS dell'ambiente Kubernetes.

Nota: la politicà di default non è `Default` ma `ClusterFirst`.

Se il `dnsPolicy` è impostato a `None` è necessario specificare un valore per `dnsConfig` tra:

- `nameservers`: lista di indirizzi IP di server DNS nel POD, massimo 3
- `searchers`: lista di domini DNS di ricerca per risoluzione degli hostname nel POD.
- `options`: lista di oggetti opzionale.

8.5 CONNETTERE APPLICAZIONI CON I SERVIZI

Docker utilizza la rete privata dell'host per far comunicare i container tra loro, il che è possibile solamente se si trovano sulla stessa macchina.

Kubernetes assegna ad ogni POD un IP privato all'interno del cluster, per cui tutti i POD possono raggiungere gli altri sulla stessa macchina e, senza bisogno di un NAT, nella rete.

8.5.1 Rendere sicuro il servizio

Per rendere sicuro il canale di comunicazione, prima di esporre il servizio su internet, è necessario:

- ottenere **certificati TLS/SSL** firmati per HTTPS
- **configurare il server** dell'applicazione per utilizzarli
- ottenere un **segreto** che renda i certificati accessibili ai POD

Al momento, gli Ingressi supportano solamente la **porta 443** come terminazione TLS. Se si specificano host differenti nella configurazione, vengono multiplexati sulla stessa porta grazie all'estensione SNI TLS (se il controller lo supporta).

8.5.2 Pubblicare il servizio

Kubernetes supporta 2 modalità di esposizione verso l'esterno: **NodePorts** e **LoadBalancer**.
La prima si riferisce ad una porta sul nodo, che deve quindi avere un IP pubblico dal quale essere raggiunto.

8.6 OGGETTO INGRESSO

Un oggetto API che gestisce accessi esterni ad un servizio nel cluster, tipicamente utilizzando HTTP. Può fornire funzionalità di load balancing, terminazioni SSL e virtual-hosting basato sui nomi (es. Apache).

Nell'oggetto `Ingresso` si definiscono regole di routing utili a gestire il traffico da un apposito controller.

Un `Ingresso` non espone porte e protocolli arbitrari: esporre protocolli di comunicazione diversi da HTTP/HTTPS richiede un servizio di tipo NodePort o LoadBalancer.

8.6.1 Regole di ingresso

Ogni regola HTTP contiene queste informazioni:

- un host opzionale
- una lista di path, ognuno dei quali ha un backend associato definito con un `serviceName` e `servicePort`.
- un backend è una combinazione di servizi e nomi di porte, ai quali vengono inviate le richieste HTTP/S ricevute

Si può configurare un backend di default per le richieste che non fanno match con nessuna entry nella lista di backend.

8.6.2 Classi di Ingressi

Un Ingresso può essere implementato da differenti controller, con diverse configurazioni. Ogni Ingresso dovrebbe specificare una `IngressClass` che contiene configurazioni addizionali, incluso il nome del controller che dovrebbe implementarne la classe.

8.6.3 Fanout

Una configurazione di routing fanout porta il traffico da un **singolo IP a più servizi**, basandosi sulla HTTP URI richiesta. Un Ingresso permette di mantenere minimo il numero di load balancers.

8.6.4 LoadBalancing

La politica di bilanciamento viene scelta nella configurazione del controller di un oggetto Ingresso e si applica a tutti gli Ingressi. Non offre le funzionalità avanzate disponibili nel load balancer di un servizio.

8.7 CONTROLLER DI INGRESSO

Questo oggetto gestisce l'oggetto Ingresso, che non funziona da solo. A differenza di altri tipi di controller non viene avviato automaticamente con il cluster.
Attualmente Kubernetes supporta e mantiene controller di Ingresso di tipo **GCE** e **nginx**.

Si può istanziare un qualunque numero di controller di Ingresso in un cluster, definendo quale classe implementa nel campo `ingress.class`, altrimenti è possibile che il cloud provider utilizzi un suo controller di default.

8.8 POLITICHE DI RETE

Una politica di rete specifica come un gruppo di POD possono comunicare tra loro e con gli endpoint di rete, utilizzano etichette per selezionare i POD e definire le regole che specificano il tipo di traffico permesso. Un plugin fornisce le implementazioni di NetworkPolicy, ma devono essere eseguite da un controller per avere effetto.

8.8.1 POD isolati e non

Di default un POD non è isolato, accetta cioè traffico da qualunque sorgente, ma una NetworkPolicy può cambiare il suo comportamento e fargli rifiutare il traffico non permesso dalle regole.
Le NetworkPolicy non vanno in conflitto, si sommano; se un POD è selezionato da più policy, il traffico viene ristretto dall'insieme delle regole di tutte le policy e l'ordine di applicazione non importa.

Di default tutto il traffico, in ingresso e in uscita, è permesso verso e da qualunque POD nel namespace.

8.9 AGGIUNGERE HOSTALIASES AI POD

Aggiungere entry al file /etc/hosts di un POD provoca una sovrascrittura della risoluzione dell'hostname a livello del POD, quando DNS e altre opzioni non sono applicabili. Le entry vanno aggiunte nelle specifiche del POD, poichè kubelet gestisce il file e potrebbe sovrascriverlo altrimenti.

Per default il file hosts include riferimenti IPv4 e IPv6 a localhost ed al suo stesso IP.

8.10 IPv4/IPv6 STACK DUALE

Questa feature permette l'allocazione di entrambe le versioni di indirizzi per POD e servizi.

Un servizio può ricevere solamente un tipo di versione, ma si può configurare una replica diversamente per coprirli entrambi.

Capitolo 9

MEMORIA DI MASSA (STORAGE)

9.1 VOLUMI

I file su disco dei container sono **effimeri** e vengono **cancellati alla terminazione**, inoltre alcuni vanno condivisi tra più container.
Anche Docker possiede un astrazione di Volume per nascondere una directory su disco o in un altro container, ma le funzionalità sono limitate rispetto alla soluzione di Kubernetes.

Un **Volume** è una directory accessibile dai container di un POD, come questo avvenga, l'hardware che lo ospita od il contenuto sono determinati dal tipo di volume.

Un Volume in Kubernetes ha un tempo di durata pari a quello del POD che lo ospita, perciò i dati contenuti **vivono più a lungo** dei container all'interno del POD e vengono mantenuti dopo il loro riavvio.
Un POD può utilizzarne molti contemporaneamente e di diversi tipi.

Nelle specifiche di un POD è possibile descrivere quale Volume montare nel campo `.spec.volumes` ed obbligatoriamente il path di mount in `.spec.container[*].volumeMounts`.
Un processo in un container vede un **filesystem** composto dall'immagine Docker (come root) e dai Volumi (montati in un path relativo alla root). I Volumi non possono avere hard link per, o essere montati in, altri Volumi.

Tipi di **Volumi specifici per cloud provider** e disponibili in rete: awsElasticBlockStore (Amazon AWS), azureDisk, azureFile, cephfs, cinder, csi, fc (canale in fibra), flexVolume, flocker, gcePersistentDisk (Google Compute Engine), gitRepo (deprecato), glusterfs, iscsi, nfs, portworxVolume, quobyte, rbd, scaleIO, storageos, vsphereVolume.

Classi di Volumi di Kubernetes:

- **configMap**: per inserire dati di configurazione in un POD.
- **downwardAPI**: per rendere disponibili API meno recenti alle applicazioni.
- **emptyDir**: viene creata, vuota, quando un POD viene assegnato ad un nodo, esiste per lo stesso tempo del POD e viene cancellata alla terminazione. Fisicamente sono ospitate sull'hardware del nodo.
- **hostPath**: directory presente sul filesystem dell'host.
- **local**: per utilizzare un disco, partizione o directory locale. Vengono utilizzate come volumi persistenti, ma sono disponibili anche all'host e non adatte ad alcune applicazioni.
- **persistentVolumeClaim**: un interfaccia per "pretendere" durabilità da un Volume senza conoscerne i particolari.
- **projected**: mappa molti Volumi in un'unica directory.
- **secret**: Volume contenente informazioni sensibili (es. password, certificati TLS), si possono montare come file.

Per rendere disponibile una parte specifica del Volume è possibile impostare la proprietà `volumeMounts.subPath` e designare un sub-path al posto della root del suddetto.

9.1.1 Propagazione del montaggio

Propagare il montaggio di un Volume può renderlo disponibile a tutti i container di un POD o ad altri POD, questa proprietà viene impostata definendo il campo `Container.volumeMounts.mountPropagation` con uno dei seguenti valori: `none`, `HostToContainer`, `Bidirectional`.

9.2 VOLUMI PERSISTENTI

Sono rappresentati con 2 astrazioni API:
PersistentVolume: è una porzione di memoria di massa nel cluster fornita tramite classi di Volumi, rispetto ai quali hanno un ciclo di vita indipendente dai singoli POD ed incapsulano i dettagli di implementazione rendendo tutti i PersistentVolume uguali (a parte per la quantità di memoria disponibile).
PersistentVolumeClaim: è una richiesta di memoria da parte dell'utente, consuma risorse di tipo PersistentVolume, possono specificare la quantità di spazio necessaria e modalità di accesso.

I POD utilizzano i claim come Volumi, sta al POD eseguire l'operazione di mount al suo interno.

9.2.1 Fornire Volumi persistenti

Un administrator del cluster può fornire `PersistentVolume` in maniera statica; questi Volumi possiedono i dettagli dello storage effettivo, disponibile agli utenti nel cluster e sono richiamabili dall'API Kubernetes.

Quando un utente crea un `PersistentVolumeClaim` che non fa match con nessun Volume statico esistente, il cluster può **crearlo dinamicamente**; deve venire specificata la `StorageClass` e l'administrator deve averla configurata per l'utilizzo.

L'administrator del cluster abilità l'offerta dinamica di Volumi basata sulle classi istanziando un controller di ammissione chiamato `DefaultStorageClass` sull'API Server.

9.2.2 Binding

Una volta creato un bind tra offerta e richiesta, questo collegamento è esclusivo: è una mappatura 1-a-1 che utilizza un oggetto `ClaimRef` per implementare un collegamento bidirezionale. Altrimenti, i `PersistentVolumeClaim` rimarranno slegati fino a quando non verrà creato un `PersistentVolume`.
Se il `PersistentVolume` è fornito dinamicamente per un `PersistentVolumeClaim`, verrà sempre legato alla stessa richiesta.

9.2.3 Oggetto Storage in Utilizzo Protetto

Questa feature assicura che i `PersistentVolumeClaim` in uso da parte di un POD non siano rimossi dal sistema, causando una perdita di dati.
Se un utente elimina un `PersistentVolumeClaim` in uso, questo non viene eliminato immediatamente, ma l'operazione viene rimandata finchè il Volume non viene più utilizzato dal POD. Comportamento simile per i `PersistentVolume` che vengono rimossi alla cancellazione del bind.

9.2.4 Riutilizzo

Si può impostare una politica di riciclo dei `PersistentVolume`, scelta tra quelle supportate:

- **retain** (tenere): permette la richiesta manuale della risorsa, il `PersistentVolume` continua ad esistere ed è considerato "rilasciato", ma non è subito disponibile perchè possiede ancora i dati della precedente bind. Un amministratore deve prima svuotarlo e cancellare l'associazione manualmente.

- **cancellazione**: alla cancellazione del `PersistentVolumeClaim` viene eliminato anche il `PersistentVolume` in Kubernetes e la memoria associata nella struttura esterna. Volumi creati dinamicamente ereditano questa policy di default.

9.2.5 Modalità di accesso

Un `PersistentVolume` può essere montato su un host in tutte le modalità supportate dal resource provider, ad esempio NFS supporta multipli client in read/write. Le modalità sono: `ReadWriteOnce`, `ReadOnlyMany` e `ReadWriteMany`.

Nota: si può montare un Volume con una sola modalità alla volta.

9.3 SNAPSHOT DI VOLUMI

Un `VolumeSnapshotContent` è una risorsa disponibile nel cluster, come i `PersistentVolume`, viene richiesta da un utente tramite `VolumeSnapshots` similmente a un `PersistentVolumeClaim`, è possibile specificare attributi da abbinare allo snapshot.
Dettagli di cui tenere conto:

- gli oggetti API `VolumeSnapshot`, `VolumeSnapshotContent` e `VolumeSnapshotClass` sono CRD (codice custom per risorse Kubernetes senza implementazione di un server dedicato) e non fanno parte del core dell'API.
- Kubernetes fornisce un controller di snapshot per il deploy nel piano di controllo e un container di supporto chiama csi-snapshotter da istanziare insieme ai driver CSI.
- il controller e gli oggetti dipendono dalla distribuzione Kubernetes.

Tipi di Volume snapshot: **Pre-provisioned** dall'amministratore del cluster e disponibili nell'API Kubernetes per l'utilizzo, oppure **Dinamicamente** richiedendo la creazione dello snapshot di un `PersistentVolumeClaim`.

9.3.1 PersistentVolumeClaim come protezione dell'origine di uno snapshot

L'obbiettivo è assicurare che un oggetto `PersistentVolumeClaim` non venga rimosso durante la creazione di un suo snapshot.
Il volume persistente risulta **in uso** durante la creazione dello snapshot, per cui non verrà cancellato fino al suo rilascio.

9.3.2 Cancellazione

Cancellando l'oggetto `VolumeSnapshot` si innesca la `DeletionPolicy` selezionata alla creazione: con `Delete` lo snapshot del Volume verrà eliminato, con `Retain` sia lo snapshot che l'oggetto `VolumeSnapshotContent` continueranno ad esistere.

9.3.3 Creare un Volume da uno snapshot

Si può creare un nuovo Volume pre-popolato da dati partendo da uno snapshot, specificandone il nome nel campo `dataSource` dell'oggetto `PersistentVolumeClaim`.

9.4 CLASSI DI STORAGE

Ogni `StorageClass` contiene i campi `provisioner` (fornitore), `parameters` (specifici per fornitore) e `reclaimPolicy`, utilizzati quando un Volume persistente di una certa classe deve essere fornito dinamicamente.

Il nome dell'oggetto di una `StorageClass` è utilizzato dall'utente per poterlo richiedere. L'amministratore imposta il nome ed altri parametri, che non possono essere cambiati una volta creato.

Policy di richiesta supportate: `Delete` (default) e `Retain`.

9.4.1 Permettere espansione del Volume

Feature configurabile, permette la modifica della dimensione del Volume modificando il corrispondente oggetto `PersistentVolumeClaim`.

9.5 CLASSI DI SNAPSHOT DI VOLUMI

Forniscono una descrizione dello snapshot al momento della creazione di uno snapshot di Volume.

Ogni `VolumeSnapshotClass` contiene i campi `driver`, `deletionPolicy`, `parameters`. Il nome di un oggetto `VolumeSnapshotClass` viene utilizzato per richiederlo, il nome ed i parametri sono impostati dall'amministratore e non possono essere cambiati una volta creato.

9.6 FORNIRE VOLUMI DINAMICAMENTE

L'implementazione di questa feature è basata sull'oggetto dell'API `StorageClass`. Un amministratore del cluster definisce tutti gli oggetti `StorageClass` di cui ha bisogno, ognuno dei quali specifica un Volume plugin (detto provisioner) che fornirà un Volume ed il set di parametri da passare al momento della creazione.

Un **prerequisito** è la creazione da parte dell'amministratore di 1 o più oggetti `StorageClass` per gli utenti.
Questo classi definiscono il provisioner utilizzato e i parametri da passare alla creazione.

L'utente **utilizza** il Volume dinamico includendo la `StorageClass` nella loro `PersistentVolumeClaim`, nel campo `storageClassName`.

9.6.1 Comportamento di deafult

Si può abilitare un comportamento di default tale che **tutte le richieste** di storage dinamico, dove non viene specificata una `StorageClass`, vengono **soddisfatte**. L'amministratore, per rendere possibile questo comportamento, deve etichettare una `StorageClass` come default e assicurarsi che il controller di ammissione `DefaultStorageClass` sia abilitato sull'API server.

Il meccanismo funziona perchè è il controller a modificare il campo `PersistentVolumeClaim.storageClassName` con il nome del default storage.
Può esistere un solo `StorageClass` di default per cluster.

9.7 LIMITI DI VOLUMI SPECIFICI PER NODO

Se il numero di Volumi (definito dal provider) per nodo viene superato, i POD potrebbero rimanere bloccati in attesa della loro creazione.

Capitolo 10

LE POLICY

Di default, i container lavorano senza **restrizioni sulle risorse** di computazione in un cluster Kubernetes. Gli amministratori utilizzano l'astrazione della **quota** per diminuire il consumo di risorse, questo vale all'interno del namespace.

10.1 IMPORRE DEI LIMITI

Esiste un `LimitRange` per namespace; è un controller di ammissione che gestisce i limiti e i ratei per tutti i POD e container che non specificano le risorse necessarie. All'interno del namespace non è possibile creare POD/container che richiedano risorse eccedenti ai limiti imposti. Può imporre limiti su:

- un minimo ed un massimo nelle risorse di computazione per POD o container
- un minimo ed un massimo per le richieste di spazio di storage per `PersistentVolumeClaim`
- istituire un rateo tra richieste e limiti per una risorsa
- imposta un limite di richieste di default per risorsa di computazione e le propaga automaticamente ai container

10.2 QUOTE DI RISORSE

Con un oggetto `ResourceQuota` si possono limitare i consumi di risorse di computazione aggregate e la quantità di oggetti istanziabili per tipo.

L'amministratore crea un oggetto `ResourceQuota` per namespace, a cui corrisponde un team, che lavora e crea oggetti (POD, controller, Volumi ecc) al suo interno. Le risorse vengono monitorate e quando la creazione o l'aggiornamento dell'ennesima eccede le quote, la richiesta fallisce con risposta HTTP `403 FORBIDDEN` insieme ad un messaggio più esplicativo.

10.3 POLICY DI SICUREZZA DEI POD

L'amministratore del cluster può istanziare un oggetto chiamato `PodSecurityPolicy` per definire un insieme di condizioni da imporre ai POD, che altrimenti non saranno instanziati. Alcuni esempi:

- esecuzione di container con privilegi
- utilizzo del namespace dell'host
- utilizzo di rete e porte dell'host
- utilizzo dei tipi dei volumi
- utilizzo del filesystem dell'host

Capitolo 11

SCHEDULER KUBERNETES

Uno scheduler **controlla la creazione di nuovi POD** non ancora assegnati a dei nodi; per ognuno di questi, lo scheduler si occupa di trovargli il nodo migliore.

11.0.1 Kube-Scheduler

`Kube-Scheduler` è l'oggetto che di default si occupa della schedulazione, esegue come parte del piano di controllo ed è progettato per essere sostituito da uno scheduler scritto da un amministratore.

Lo scheduler lavora filtrando i nodi sulla base dei requisiti del POD in esame; un nodo adatto è chiamato "utilizzabile" (**feasible**). Se non si trovano nodi adatti, il POD rimane non schedulato finchè lo scheduler non trova una soluzione.

Una volta trovato un insieme di nodi adatti, lo scheduler esegue su questi delle funzioni di test, che restituiscono un punteggio di "utilizzabilità" necessario per fare una classifica. Il nodo migliore verrà poi notificato all'API server con un'operazione di binding.

Ci sono due modi supportati per configurare il filtraggio dei nodi e l'assegnamento del punteggio:

- `Scheduling Policies` permette di scegliere i "predicati" per il filtrag-

gio e le "piorità" nel punteggio

- `Scheduling Profiles` permette di configurare dei Plugin che implementano diversi momenti nella schedulazione, tra cui: `Queuesort`, `Filter`, `Score`, `Bind`, `Reserve` e `Permit`.

11.1 FRAMEWORK DI SCHEDULAZIONE

Il framework di schedulazione aggiunge un'API che **permette di creare features** nella forma di plugin, mantenendo il core dello scheduler semplice e mantenibile.
Queste estensioni vengono registrate per essere invocate in uno o più punti, alcuni di questi possono cambiare le decisioni di scheduling mentre altri sono solamente informativi.
Ogni tentativo di schedulazione di un POD è diviso in 2 fasi:

- **Scheduling Cycle**: selezione del nodo, eseguiti serialmente
- **Binding Cycle**: applicazione della decisione, eseguiti concorrenzialmente

Insieme sono definite come "contesto di scheduling".

In caso di errore, il ciclo viene interrotto ed il POD ritorna nella lista di attesa, pronto per essere schedulato.

Capitolo 12

ESEMPI PRATICI

In questo capitolo mostro come la comunità open source abbia reso l'introduzione a Kubernetes quasi una passeggiata, offrendo varie versioni di Kubernetes dalle dimensioni ridotte che possono girare su comuni pc portatili senza requisiti hardware eccessivi, inoltre esiste una GUI gratuita per il monitoraggio del cluster in maniera più comprensibile ed articolata, anche grazie all'utilizzo di grafici.

Esiste anche l'opzione di utilizzare un tutorial presente sul sito `Kubernetes.io`, dove è presente un terminale per l'utilizzo di una macchina virtuale (o un container?) dove i software necessari sono già installati.
Inoltre questi esempi e molti altri sono tutti disponibili all'interno del sito.

12.1 HARDWARE & SOFTWARE

Hardware: pc portatile dotato di 8gb RAM, CPU intel i5 7300HQ, scheda grafica nvidia GTX 1050, SSD 512 GB.
Software: completamente open source: ubuntu, docker, microk8s e lens IDE. Sono compatibili anche con windows, ma con maggiore overhead (consigliato multipass o minikube: gestori di macchine virtuali linux, non molto diverse da virtual box).

La scelta di scartare minikube, che è consigliato dal sito ufficiale di Kubernetes, è dovuta alla semplicità di installazione ed al maggior isolamento, inoltre ho ritenuto poco rilevante una possibile instabilità nel caso degli esem-

pi.
Microk8s installato su ubuntu risulta molto leggero ed efficiente, nonostante sia in grado di utilizzare più nodi e di scalare molto.

12.1.1 installazione

con kubernetes 1.18

```
sudo snap install microk8s --classic --channel=1.18/stable
```

Aggiunta dell'utente al gruppo di microk8s per l'utilizzo di comandi con privilegi di amministratore

```
sudo usermod -a -G microk8s \$USER
sudo chown -f -R \$USER ~/.kube
```

login dell'utente per utilizzare comandi kubectl senza preporre `sudo`

```
su - \$USERs
```

Ottenere info su microk8s

```
microk8s status
microk8s kubectl get nodes
microk8s kubectl get services
microk8s kubectl create deployment
microk8s kubectl get pods
```

Creare alias del comando kubectl, semplifica utilizzo

```
alias kubectl='microk8s kubectl'
```

Per salvare l'alias, aggiungere la stessa riga al file `~/.bashrc`.
Addons abilitati nel cluster con comando '`microk8s enable`':
prometheus: per l'invio di statistiche all'IDE Lens
dns: per la risoluzione di nomi da parte dei POD, container ecc
storage: per l'istanziazione di Volumi

12.1.2 LENS IDE

Applicazione stand-alone e real time per il monitoraggio dello stato del cluster, possiede un terminale bash integrato per lavorare sul sistema, testato su cluster con 25k POD, tutt'ora in continuo sviluppo.
Permette una gestione multicluster, multi workspace, con visualizzazione grafica delle statistiche e relativo storico grazie all'addon prometheus. Esegue un ciclo continuo di verifica dello stato del cluster comunicando grazie al suo kubectl built-in e alle Kubernetes RBAC.

RBAC: Role-Based Access Control, è un approccio utilizzato per restringere l'accesso degli utenti alle applicazioni, sul sistema e sulla rete attraverso la concessione di autorizzazioni e permessi.

Lens è stato originariamente sviluppato da Kontena Inc. Ad oggi, tutti i diritti appartengono a Lakend Labs, un collettivo di persone che si occupano dello sviluppo.

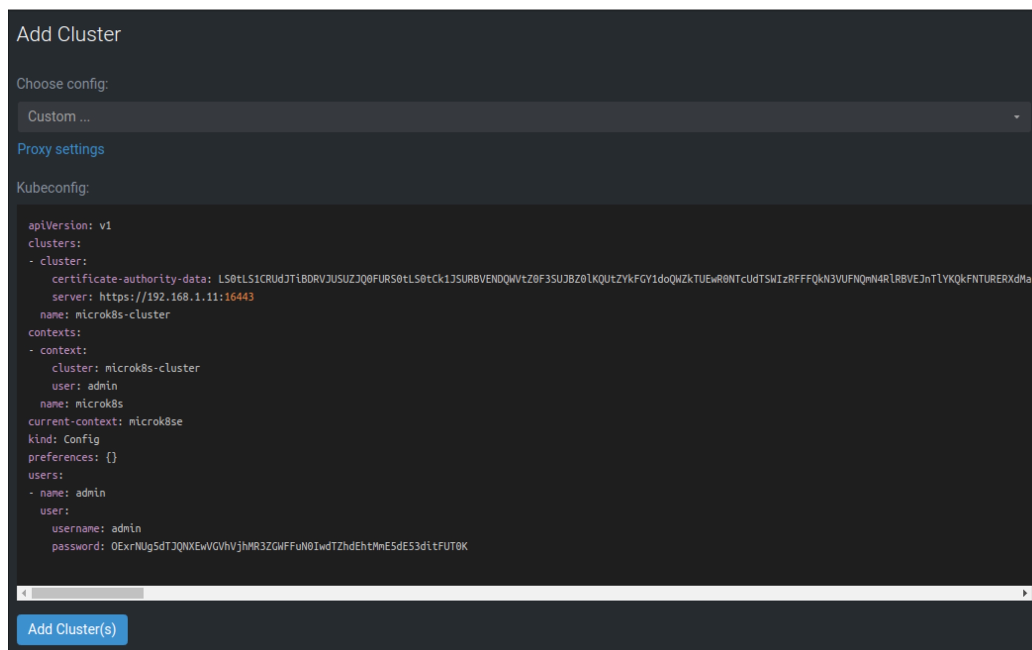

Figura 12.1: utilizzo di kubeconfig per accedere al cluster con Lens IDE

12.2 ESEMPI

12.2.1 Applicazione Guestbook stateless

Questo esempio si compone di un backend Redis, per la gestione delle entry, e di un frontend php based. In dettaglio:

- singola istanza master Redis
- multiple istanze di slave Redis, per gestire le letture
- multiple istanze del frontend
- un servizio per ogni tipo di istanza, per indirizzare il traffico

Un servizio aggiuntivo di load balancing, non incluso nell'esempio, potrebbe esporre l'applicazione su un IP pubblico e renderla raggiungibile, ma il suo funzionamento è legato all'integrazione con un cloud provider.
In locale l'alternativa è esporre una porta della macchina in uso e collegarla a quella del servizio del frontend già presente nel cluster.

E' necessario un DNS all'interno del cluster per il funzionamento degli slave redis:

```
microk8s enable dns
```

Ogni file .yaml va salvato in una directory, il suo contenuto viene aggiunto allo stato desiderato grazie al comando

```
kubectl apply -f <nomefile.yaml>
```

1. redis master replica:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-master
  labels:
    app: redis
```

```
spec:
  selector:
        matchLabels:
          app: redis
          role: master
          tier: backend
  replicas: 1
  template:
        metadata:
          labels:
                app: redis
                role: master
                tier: backend
        spec:
          containers:
          - name: master
                image: k8s.gcr.io/redis
                  resources:
                        requests:
                          cpu: 100m
                          memory: 100Mi
                  ports:
                  - containerPort: 6379
```

2. servizio del master Redis:

```
apiVersion: v1
kind: Service
metadata:
  name: redis-master
  labels:
        app: redis
        role: master
        tier: backend
spec:
  ports:
  - port: 6379
        targetPort: 6379
  selector:
        app: redis
        role: master
```

```
        tier: backend
```

3. deployment degli slave Redis:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-slave
  labels:
        app: redis
spec:
  selector:
        matchLabels:
          app: redis
          role: slave
          tier: backend
  replicas: 2
  template:
        metadata:
          labels:
                app: redis
                role: slave
                tier: backend
        spec:
          containers:
          - name: slave
                image: gcr.io/google_samples/gb-redisslave
                resources:
                  requests:
                        cpu: 100m
                        memory: 100Mi
                env:
                - name: GET_HOSTS_FROM
                  value: dns
                ports:
                - containerPort: 6379
```

4. servizio per gli slave Redis:

```
apiVersion: v1
kind: Service
```

```
metadata:
  name: redis-slave
  labels:
        app: redis
        role: slave
        tier: backend
spec:
  ports:
  - port: 6379
  selector:
        app: redis
        role: slave
        tier: backend
```

5. deployment frontend php

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  labels:
        app: guestbook
spec:
  selector:
        matchLabels:
          app: guestbook
          tier: frontend
  replicas: 3
  template:
        metadata:
          labels:
                app: guestbook
                tier: frontend
        spec:
          containers:
          - name: php-redis
                image: gcr.io/google-samples/gb-frontend:v4
                resources:
                  requests:
                        cpu: 100m
                        memory: 100Mi
```

```
            env:
            - name: GET_HOSTS_FROM
              value: dns
            ports:
            - containerPort: 80
```

6. servizio del frontend

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
        app: guestbook
        tier: frontend
spec:
  type: NodePort
  ports:
  - port: 80
  selector:
        app: guestbook
        tier: frontend
```

7. accedere al servizio:

```
        kubectl get service frontend
```

   Visitare l'indirizzo IP con qualunque browser.

8. ripulire tutto: la cancellazione di Deployment e Servizi elimina automaticamente anche tutti i POD correlati.

```
        kubectl delete deployment -l app=redis
        kubectl delete service -l app=redis
        kubectl delete deployment -l app=guestbook
        kubectl delete service -l app=guestbook
```

12.2.2 Sito web Wordpress con database MySQL

Entrambe le applicazioni sono **statefull** e utilizzano `PersistentVolumes` e `PersistentVolumeClaims` per mantenere i dati, questi verranno creati durante il deployment.

Quando nasce un `PersistentVolumeClaims` viene fornito dinamicamente un `PersistentVolume` in base alla configurazione della `StorageClass`. Nel caso di un installazione locale, si utilizzerà di default una logica `hostPath`; i dati saranno salvati nella directory /**tmp** del nodo.
Quindi, se il POD termina e viene schedulato altrove i dati vengono persi.

Verranno creati:

1. Un generatore di `Secret`: questo mantiene la **password** del database ed il riferimento ai file di deployment a cui fornirla.
2. Una singola istanza del database **MySQL**: questo monta un `PersistentVolume` in `/var/lib/mysql` e imposta la variabile globale `MYSQL_ROOT_PASSWORD` con il valore fornito dal segreto.
3. Una singola istanza di **WordPress**: questo monta un `PersistentVolume` in `/var/www/html` per i file del CMS. La variabile d'ambiente `WORDPRESS_DB_HOST` imposta il nome del servizio MySQL definito prima, servizio utilizzato come punto d'accesso al database. Mentre l'altra variabile `WORDPRESS_DB_PASSWORD` viene impostata dal Segreto.

Inserendo i 3 file qui sotto riportati nella stessa directory, si utilizza il comando

```
        kubectl apply -g ./
```

per applicare il contenuto dei file al cluster e istanziare tutte le parti dell'applicazione. Con i seguenti comandi è possibile verificare da linea di comando l'esistenza delle 3 parti:

```
kubectl get secrets
kubectl get pvc  //per i volumi
kubectl get pods
kubectl get services wordpress   //mostra l'URL del cluster
kubectl delete -k ./   //elimina l'applicazione dal cluster
```

File per il deployment:

1. kustomization.yaml

```
secretGenerator:
- name: mysql-pass
       literals:
       - password=abcd1234
resources:
       - mysql-deployment.yaml
       - wordpress-deployment.yaml
```

2. mysql-deployment.yaml

```
apiVersion: v1
kind: Service
metadata:
       name: wordpress-mysql
       labels:
              app: wordpress
spec:
       ports:
              - port: 3306
       selector:
              app: wordpress
              tier: mysql
       clusterIP: None
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
       name: mysql-pv-claim
       labels:
              app: wordpress
spec:
       accessModes:
              - ReadWriteOnce
       resources:
              requests:
                     storage: 20Gi
---
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
       name: wordpress-mysql
       labels:
              app: wordpress
spec:
       selector:
              matchLabels:
                     app: wordpress
                     tier: mysql
       strategy:
              type: Recreate
       template:
              metadata:
                     labels:
                            app: wordpress
                            tier: mysql
              spec:
                     containers:
                     - image: mysql:5.6
                            name: mysql
                            env:
                            - name: MYSQL_ROOT_PASSWORD
                                   valueFrom:
                                      secretKeyRef:
                                        name: mysql-pass
                                        key: password
                            ports:
                            - containerPort: 3306
                                   name: mysql
                       volumeMounts:
                       - name: mysql-persistent-storage
                          mountPath: /var/lib/mysql
                volumes:
                - name: mysql-persistent-storage
                       persistentVolumeClaim:
                             claimName: mysql-pv-claim
```

3. wordpress-deployment.yaml

```
apiVersion: v1
kind: Service
metadata:
	name: wordpress
	labels:
		app: wordpress
spec:
	ports:
		- port: 80
	selector:
		app: wordpress
		tier: frontend
	type: LoadBalancer
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
	name: wp-pv-claim
	labels:
		app: wordpress
spec:
	accessModes:
		- ReadWriteOnce
	resources:
		requests:
			storage: 20Gi
---
apiVersion: apps/v1
kind: Deployment
metadata:
	name: wordpress
	labels:
		app: wordpress
spec:
	selector:
		matchLabels:
			app: wordpress
			tier: frontend
	strategy:
		type: Recreate
	template:
```

```
            metadata:
                    labels:
                            app: wordpress
                            tier: frontend
    spec:
            containers:
            - image: wordpress:4.8-apache
                    name: wordpress
                    env:
                    - name: WORDPRESS_DB_HOST
                            value: wordpress-mysql
                    - name: WORDPRESS_DB_PASSWORD
                            valueFrom:
                                    secretKeyRef:
                                            name: mysql-pass
                                            key: password
                    ports:
                    - containerPort: 80
                            name: wordpress
                    volumeMounts:
                    - name: wordpress-persistent-storage
                            mountPath: /var/www/html
            volumes:
            - name: wordpress-persistent-storage
                    persistentVolumeClaim:
                            claimName: wp-pv-claim
```

12.2.3 Verifica

microk8s

Services | Endpoints | Ingresses | Network Policies

Services — 14 items

Name	Namespace	Type	Cluster IP	Ports	External IP	Selector	Age	Status
alertmanager-main	monitoring	ClusterIP	10.152.183.202	9093:web/TCP	-	alertmanager=main app=al...	2h	Active
alertmanager-operated	monitoring	ClusterIP	None	9093/TCP, 9094/TCP, 9094/U...	-	app=alertmanager	2h	Active
grafana	monitoring	ClusterIP	10.152.183.217	3000:http/TCP	-	app=grafana	2h	Active
kube-dns	kube-system	ClusterIP	10.152.183.10	53/UDP, 53/TCP, 9153/TCP	-	k8s-app=kube-dns	2h	Active
kube-state-metrics	monitoring	ClusterIP	None	8443:https-main/TCP, 9443:h...	-	app=kube-state-metrics	2h	Active
kubelet	kube-system	ClusterIP	None	10250/TCP	-		2h	Active
kubernetes	default	ClusterIP	10.152.183.1	443:16443/TCP	-		1d	Active
node-exporter	monitoring	ClusterIP	None	9100:https/TCP	-	app=node-exporter	2h	Active
prometheus-adapter	monitoring	ClusterIP	10.152.183.94	443:6443/TCP	-	name=prometheus-adapter	2h	Active
prometheus-k8s	monitoring	ClusterIP	10.152.183.145	9090:web/TCP	-	app=prometheus promethe...	2h	Active
prometheus-operated	monitoring	ClusterIP	None	9090:web/TCP	-	app=prometheus	2h	Active
prometheus-operator	monitoring	ClusterIP	None	8080:http/TCP	-	app.kubernetes.io/compone...	2h	Active
wordpress	default	LoadBalancer	10.152.183.177	80:32444/TCP	-	app=wordpress tier=fronte...	4m	Pending
wordpress-mysql	default	ClusterIP	None	3306/TCP	-	app=wordpress tier=mysql	4m	Active

Figura 12.2: istanze nel cluster dopo deployment

Service: wordpress

Created	5m ago (2020-06-21T14:56:42Z)
Name	wordpress
Namespace	default
Labels	app=wordpress
Selector	app=wordpress tier=frontend
Type	LoadBalancer
Session Affinity	None

Connection

Cluster IP	10.152.183.177
Ports	80:32444/TCP

Endpoint

Name	Endpoints
wordpress	10.1.27.28:80

Events

Figura 12.3: dettaglio del POD del CMS

Bibliografia

[1] Ci/cd - redhat. https://www.redhat.com/it/topics/devops/what-is-ci-cd.

[2] Cloud computing. https://it.wikipedia.org/wiki/Cloud_computing.

[3] Containers: Google. https://cloud.google.com/containers/?hl=it.

[4] Containers: Ibm. https://www.ibm.com/it-it/cloud/container-service.

[5] Cosa sono i microservizi - amazon. https://aws.amazon.com/it/microservices/.

[6] Cosa sono i microservizi - redhat. https://www.redhat.com/it/topics/microservices/what-are-microservices.

[7] Kubernetes: descrizione di amazon aws. https://aws.amazon.com/it/kubernetes/.

[8] Kubernetes: descrizione di google. https://cloud.google.com/kubernetes/?hl=it.

[9] Kubernetes: descrizione di redhat. https://www.redhat.com/it/topics/containers/what-is-kubernetes.

[10] Kubernetes: sito sviluppatori. https://kubernetes.io/.

[11] Kubernets su github. https://github.blog/2017-08-16-kubernetes-at-github/.

[12] lens ide su github. https://github.com/lensapp/lens.

[13] microk8s sito ufficiale. https://microk8s.io/.

[14] microk8s su github. https://github.com/ubuntu/microk8s.

[15] sito ufficiale docker. https://www.docker.com/.

[16] RedHat. What is docker. https://www.redhat.com/it/topics/containers/what-is-docker.